Contents

[11\_ContainerWater.java 22](#_Toc20323675)

[42\_TrappingRainWater.java 23](#_Toc20323676)

[62\_UniquePaths.java 26](#_Toc20323677)

[Sum Target 27](#_Toc20323678)

[1\_TwoSum.java 27](#_Toc20323679)

[167\_TwoSumII.java 28](#_Toc20323680)

[15\_3Sum.java 29](#_Toc20323681)

[16\_3SumClosest.java 30](#_Toc20323682)

[18\_4Sum.java 31](#_Toc20323683)

[Strings 33](#_Toc20323684)

[3\_LongestSubstringWithoutRepeatingCharacters.java 33](#_Toc20323685)

[10\_RegularExpressionMatching.java 35](#_Toc20323686)

[79\_LCS\_LongestCommonSubstring.java 38](#_Toc20323687)

[79\_WordSearch.java 39](#_Toc20323688)

[212\_WordSearchII.java 41](#_Toc20323689)

[14\_LongestCommonPrefix.java 49](#_Toc20323690)

[151\_ReverseWordsinString.java 50](#_Toc20323691)

[186\_ReverseWordsinStringII.java 51](#_Toc20323692)

[271\_EncodeAndDecodeStrings.java 52](#_Toc20323693)

[459\_RepeatedSubstringPattern.java 53](#_Toc20323694)

[557\_ReverseWordsInAStringIII.java 54](#_Toc20323695)

[Stocks 56](#_Toc20323696)

[121\_BestTimetoBuyandSellStock.java 56](#_Toc20323697)

[122\_BestTimetoBuyandSellStockII.java 57](#_Toc20323698)

[123\_BestTimetoBuyandSellStockIII.java 58](#_Toc20323699)

[188\_BestTimetoBuyandSellStockIV.java 59](#_Toc20323700)

[309\_Best Time to Buy and Sell Stock with Cooldown.java 60](#_Toc20323701)

[Palindrome 61](#_Toc20323702)

[125\_ValidPalindrome.java 61](#_Toc20323703)

[131\_PalindromePartitioning.java 62](#_Toc20323704)

[132\_Palindrome.java 65](#_Toc20323705)

[266\_PalindromePermutation.java 68](#_Toc20323706)

[267\_ PalindromePermutationII.java 69](#_Toc20323707)

[336\_PalindromePairs.java 71](#_Toc20323708)

[Parentheses 72](#_Toc20323709)

[20\_ValidParentheses.java 72](#_Toc20323710)

[32\_LongestValidParenthesesSubstring.java 73](#_Toc20323711)

[301\_RemoveInvalidParentheses.java 76](#_Toc20323712)

[423\_ValidParentheses.java 77](#_Toc20323713)

[1021\_RemoveOutermostParentheses.java 78](#_Toc20323714)

[Trie 80](#_Toc20323715)

[208\_Trie.java 80](#_Toc20323716)

[Longest 80](#_Toc20323717)

[76\_LIS\_LongestIncreasingSubsequence.java 80](#_Toc20323718)

[76\_MinimumWindowSubstring.java 81](#_Toc20323719)

[124\_ LongestSequenceOfRepetativeChars.java 83](#_Toc20323720)

[124\_LongestConsecutiveSequence.java 83](#_Toc20323721)

[128\_LongestConsecutiveSequence.java 85](#_Toc20323722)

[159\_LongestSubstringWithAtMostTwoDistinctCharacters.java 85](#_Toc20323723)

[300\_LIS\_LengthOfLongestIncreasingSubsequence.java 87](#_Toc20323724)

[340\_LongestSubstringWithAtMostKDistinctCharacters.java 91](#_Toc20323725)

[395\_LongestSubstringWithAtLeastKRepeatingCharacters.java 92](#_Toc20323726)

[397\_LICS\_LongestContinuousIncreasingSubsequence.java 94](#_Toc20323727)

[516\_LongestPalindromicSubsequence.java 95](#_Toc20323728)

[673\_LCIS\_Number\_LongestContinuousIncreasingSubsequence.java 97](#_Toc20323729)

[674\_LCIS\_LongestContinuousIncreasingSubsequence.java 99](#_Toc20323730)

[1027\_LongestArithmeticSubsequence.java 100](#_Toc20323731)

[LinkedList 101](#_Toc20323732)

[19\_RemoveNth.java 101](#_Toc20323733)

[21\_MergeTwoSortedLists.java 102](#_Toc20323734)

[23\_MergekSortedLists.java 103](#_Toc20323735)

[24\_SwapNodes.java 105](#_Toc20323736)

[83\_RemoveDuplicates.java 105](#_Toc20323737)

[105\_CopyListwithRandomPointer.java 106](#_Toc20323738)

[129\_RehashingKeysForHashTable.java 109](#_Toc20323739)

[141\_DetectLinkedListCycle.java 110](#_Toc20323740)

[142\_LinkedListCycleII.java 111](#_Toc20323741)

[147\_InsertionSortLinkedList.java 113](#_Toc20323742)

[148\_SortLinkedList.java 113](#_Toc20323743)

[160\_IntersectionTwoLinkedLists.java 114](#_Toc20323744)

[167\_AddTwoNumbersLinkedList.java 117](#_Toc20323745)

[170\_RotateLinkedList.java 119](#_Toc20323746)

[203\_RemoveLinkedListElements.java 120](#_Toc20323747)

[206\_ReverseLinkedList.java 121](#_Toc20323748)

[234\_PalindromeLinkedList.java 121](#_Toc20323749)

[237\_DeleteNodeInLinkedList.java 123](#_Toc20323750)

[328\_OddEvenLinkedList.java 124](#_Toc20323751)

[817\_Linked.java 125](#_Toc20323752)

[1030\_NextLargerNode.java 127](#_Toc20323753)

[BinaryTree 128](#_Toc20323754)

[70\_BinaryTreeLevelOrderTraversalII.java 128](#_Toc20323755)

[236\_LowestCommonAncestorBinaryTree.java 130](#_Toc20323756)

[298\_BinaryTreeLongestConsecutiveSequence.java 131](#_Toc20323757)

[448\_InorderSuccessorinBinarySearchTree.java 133](#_Toc20323758)

[545\_BoundaryofBinaryTree.java 134](#_Toc20323759)

[549\_LongestConsecutivePathInBinaryTree.java 136](#_Toc20323760)

[617\_MergeTwoBinaryTrees.java 138](#_Toc20323761)

[623\_AddOneRowToTree.java 140](#_Toc20323762)

[671\_SecondMinimumBinary.java 142](#_Toc20323763)

[1008\_ConstructBinarySearchTreeFromPreorderTraversal.java 143](#_Toc20323764)

[1028\_Recover.java 145](#_Toc20323765)

[100\_SameTree.java 146](#_Toc20323766)

[101\_CheckBinaryTreeSymmetricMirror.java 147](#_Toc20323767)

[103\_BinaryTreeZigzagTraversal.java 148](#_Toc20323768)

[104\_MaximumDepthBinaryTree.java 150](#_Toc20323769)

[105\_ConstructBinaryTreeFromPreorderAndInorderTraversal.java 151](#_Toc20323770)

[106\_ConstructBinaryTreeFromInorderAndPostorderTraversal.java 152](#_Toc20323771)

[106\_ConvertSortedListtoBalancedBST.java 154](#_Toc20323772)

[107\_BinaryTreeLevelOrderTraversal.java 156](#_Toc20323773)

[110\_BalancedBinaryTree.java 157](#_Toc20323774)

[111\_MinimumDepthofBinaryTree.java 159](#_Toc20323775)

[112\_PathSum.java 160](#_Toc20323776)

[113\_PathSumII.java 160](#_Toc20323777)

[114\_FlattenBinaryTree.java 163](#_Toc20323778)

[124\_MaximumPathSum.java 166](#_Toc20323779)

[144\_BinaryTreePreorderTraversal.java 168](#_Toc20323780)

[145\_BinaryTreePostorderTraversa.java 169](#_Toc20323781)

[156\_Binary Tree Upside Down.java 171](#_Toc20323782)

[199\_BinaryTreeRightSideView.java 173](#_Toc20323783)

[226\_InvertBinaryTree.java 174](#_Toc20323784)

[235\_LowestCommonAncestorBinarySearchTree.java 175](#_Toc20323785)

[250\_CountNodesWithSameValue.java 177](#_Toc20323786)

[257\_BinaryTreePaths.java 178](#_Toc20323787)

[285\_InorderSuccessorInBST.java 179](#_Toc20323788)

[272\_ClosestBinarySearchTreeValueII.java 181](#_Toc20323789)

[297\_SerializeandDeserializeBinaryTree.java 182](#_Toc20323790)

[331\_VerifyPreorderSerializationOfaBinaryTree.java 187](#_Toc20323791)

[333\_LargestBSTSubTree.java 189](#_Toc20323792)

[404\_SumOfAllLeftLeaves.java 190](#_Toc20323793)

[726\_CheckFullBinaryTree.java 192](#_Toc20323794)

[Graph 193](#_Toc20323795)

[133\_CloneGraph.java 193](#_Toc20323796)

[178\_GraphValidTree.java 198](#_Toc20323797)

[332\_ReconstructItinerary.java 200](#_Toc20323798)

[DesignQuestions 200](#_Toc20323799)

[146\_LRUCache.java 201](#_Toc20323800)

[155\_MinStack.java 204](#_Toc20323801)

[170\_Two Sum III - Data structure design.java 206](#_Toc20323802)

[208\_Trie.java 207](#_Toc20323803)

[211\_Add and Search Word - Data structure design.java 210](#_Toc20323804)

[225\_Implement Stack using Queues.java 212](#_Toc20323805)

[232\_ImplementQueueUsingStacks.java 214](#_Toc20323806)

[348\_Design\_Tic\_Tac.java 215](#_Toc20323807)

[353\_DesignSnakeGame.java 219](#_Toc20323808)

[362\_DesignHitCounter.java 221](#_Toc20323809)

[379\_Design Phone Directory.java 223](#_Toc20323810)

[419\_Battleships.java 224](#_Toc20323811)

[460\_LFUCache.java 226](#_Toc20323812)

[526\_LoadBalancer.java 228](#_Toc20323813)

[622\_DesignCircularQueue.java 230](#_Toc20323814)

[1001\_Grid.java 232](#_Toc20323815)

[1002\_Find.java 235](#_Toc20323816)

[1003\_Check.java 237](#_Toc20323817)

[1004\_Max.java 238](#_Toc20323818)

[1005\_Maximize.java 239](#_Toc20323819)

[1006\_Clumsy.java 240](#_Toc20323820)

[1007\_Minimum.java 242](#_Toc20323821)

[1012\_Complement.java 244](#_Toc20323822)

[1013\_Pairs.java 246](#_Toc20323823)

[1014\_Capacity.java 247](#_Toc20323824)

[1020\_Partition.java 250](#_Toc20323825)

[1021\_Best.java 251](#_Toc20323826)

[1022\_Smallest.java 252](#_Toc20323827)

[1022\_Sum.java 253](#_Toc20323828)

[1023\_Binary.java 254](#_Toc20323829)

[1023\_Camelcase.java 255](#_Toc20323830)

[1024\_Video.java 256](#_Toc20323831)

[1025\_Divisor.java 258](#_Toc20323832)

[1026\_Maximum.java 259](#_Toc20323833)

[1028\_Convert.java 260](#_Toc20323834)

[1029\_Binary.java 262](#_Toc20323835)

[1029\_Two.java 263](#_Toc20323836)

[1030\_Matrix.java 264](#_Toc20323837)

[1031\_Maximum.java 266](#_Toc20323838)

[1031\_Number.java 267](#_Toc20323839)

[1032\_Stream.java 269](#_Toc20323840)

[1033\_Moving.java 272](#_Toc20323841)

[1034\_Coloring.java 273](#_Toc20323842)

[1035\_Uncrossed.java 275](#_Toc20323843)

[1036\_Escape.java 276](#_Toc20323844)

[1037\_Valid.java 278](#_Toc20323845)

[1038\_Binary.java 278](#_Toc20323846)

[107\_WordBreak.java 281](#_Toc20323847)

[140\_WordBreakII.java 283](#_Toc20323848)

[109\_Triangle.java 287](#_Toc20323849)

[111\_ClimbingStairs.java 288](#_Toc20323850)

[115\_Distinct.java 288](#_Toc20323851)

[116\_JumpGame.java 291](#_Toc20323852)

[116\_Populating.java 292](#_Toc20323853)

[117\_JumpGameII.java 295](#_Toc20323854)

[117\_Populating.java 297](#_Toc20323855)

[118\_Pascal\_Triangle.java 299](#_Toc20323856)

[119\_Pascal'sTriangleII.java 300](#_Toc20323857)

[125\_BackpackII.java 302](#_Toc20323858)

[126\_Word.java 303](#_Toc20323859)

[127\_WordLadder.java 306](#_Toc20323860)

[129\_SumRootLeafNumbers.java 309](#_Toc20323861)

[134\_GasStation.java 311](#_Toc20323862)

[136\_SingleNumber.java 312](#_Toc20323863)

[137\_Single.java 313](#_Toc20323864)

[138\_SubarraySum.java 314](#_Toc20323865)

[139\_SubarraySumClosest.java 316](#_Toc20323866)

[141\_Sqrt.java 317](#_Toc20323867)

[143\_SortColorsII.java 318](#_Toc20323868)

[144\_InterleavingPositiveandNegativeNumbers.java 320](#_Toc20323869)

[149\_Max.java 321](#_Toc20323870)

[150\_EvaluateReversePolishNotation.java 322](#_Toc20323871)

[152\_MaximumProductSubarray.java 324](#_Toc20323872)

[153\_Find Minimum in Rotated Sorted Array.java 325](#_Toc20323873)

[154\_ Find Minimum in Rotated Sorted ArrayII.java 325](#_Toc20323874)

[Intervals 326](#_Toc20323875)

[156\_MergeIntervals.java 326](#_Toc20323876)

[157\_Read.java 328](#_Toc20323877)

[158\_Read.java 329](#_Toc20323878)

[161\_RotateImage.java 331](#_Toc20323879)

[162\_FindPeakElement.java 332](#_Toc20323880)

[163\_Missing.java 334](#_Toc20323881)

[164\_Maximum.java 334](#_Toc20323882)

[165\_Compare.java 335](#_Toc20323883)

[166\_Fraction.java 337](#_Toc20323884)

[Anagrams 339](#_Toc20323885)

[171\_Anagrams.java 339](#_Toc20323886)

[242\_ValidAnagram.java 340](#_Toc20323887)

[249\_Group.java 342](#_Toc20323888)

[438\_FindAllAnagramsinString.java 343](#_Toc20323889)

[Excel 345](#_Toc20323890)

[168\_ColumnExcel.java 345](#_Toc20323891)

[171\_ExcelNumber.java 346](#_Toc20323892)

[169\_MajorityElement.java 347](#_Toc20323893)

[172\_FactorialTrailingZeroes.java 348](#_Toc20323894)

[174\_DungeonGame.java 349](#_Toc20323895)

[179\_LargestNumber.java 351](#_Toc20323896)

[187\_Repeated.java 352](#_Toc20323897)

[189\_RotateArray.java 352](#_Toc20323898)

[190\_ReverseUnsignedInteger.java 353](#_Toc20323899)

[191\_Number.java 354](#_Toc20323900)

[198\_HouseRobber.java 355](#_Toc20323901)

[213\_HouseRobberII.java 356](#_Toc20323902)

[337\_HouseRobberIII.java 357](#_Toc20323903)

[200\_NumberOfIslands.java 359](#_Toc20323904)

[201\_Bitwise.java 362](#_Toc20323905)

[202\_HappyNumber.java 363](#_Toc20323906)

[204\_CountPrimes.java 364](#_Toc20323907)

[205\_IsomorphicStrings.java 365](#_Toc20323908)

[207\_CourseSchedule.java 366](#_Toc20323909)

[210\_CourseScheduleII.java 368](#_Toc20323910)

[214\_Shortest.java 371](#_Toc20323911)

[215\_Kth.java 371](#_Toc20323912)

[216\_Combination.java 373](#_Toc20323913)

[218\_Skyline.java 373](#_Toc20323914)

[219\_Contains.java 377](#_Toc20323915)

[22\_FlattenList.java 377](#_Toc20323916)

[22\_GenerateParentheses.java 378](#_Toc20323917)

[220\_Contains.java 379](#_Toc20323918)

[221\_MaximalSquare.java 381](#_Toc20323919)

[222\_count.java 382](#_Toc20323920)

[223\_RectangleArea.java 383](#_Toc20323921)

[224\_Basic.java 383](#_Toc20323922)

[227\_Basic.java 384](#_Toc20323923)

[228\_Summary.java 387](#_Toc20323924)

[229\_MajorityElementII.java 388](#_Toc20323925)

[230\_KthSmallestBST.java 389](#_Toc20323926)

[231\_PowerTwo.java 391](#_Toc20323927)

[238\_ProductofArrayExceptSelf.java 392](#_Toc20323928)

[239\_SlidingWindowMaximum.java 393](#_Toc20323929)

[240\_SearchII.java 395](#_Toc20323930)

[241\_Different.java 396](#_Toc20323931)

[243\_ShortestI.java 397](#_Toc20323932)

[244\_ShortestII.java 398](#_Toc20323933)

[245\_ShortestIII.java 399](#_Toc20323934)

[246\_Strobogrammatic.java 400](#_Toc20323935)

[247\_Strobogrammatic.java 401](#_Toc20323936)

[251\_Flatten2DVector.java 402](#_Toc20323937)

[254\_Factor.java 404](#_Toc20323938)

[255\_Verify.java 406](#_Toc20323939)

[256\_paint.java 408](#_Toc20323940)

[258\_Add.java 409](#_Toc20323941)

[259\_3SumSmaller.java 410](#_Toc20323942)

[26\_RemoveDuplicatesfromSortedArray.java 411](#_Toc20323943)

[260\_Single.java 412](#_Toc20323944)

[263\_UglyNumber.java 414](#_Toc20323945)

[264\_UglyNumberII.java 415](#_Toc20323946)

[265\_Paint.java 416](#_Toc20323947)

[268\_Missing.java 418](#_Toc20323948)

[269\_AlienDictionary.java 421](#_Toc20323949)

[27\_RemoveElement.java 423](#_Toc20323950)

[270\_ClosestBinarySearchTreeValue.java 424](#_Toc20323951)

[273\_Integer.java 426](#_Toc20323952)

[274\_H-Index.java 427](#_Toc20323953)

[275\_Hindex.java 427](#_Toc20323954)

[276\_Paint.java 428](#_Toc20323955)

[279\_PerfectSquares.java 430](#_Toc20323956)

[28\_ImplementstrStr.java 431](#_Toc20323957)

[280\_WiggleSort.java 431](#_Toc20323958)

[324\_WiggleSortII.java 432](#_Toc20323959)

[281\_Zigzag.java 433](#_Toc20323960)

[282\_Expression.java 435](#_Toc20323961)

[283\_MoveZeroes.java 438](#_Toc20323962)

[284\_Peeking.java 439](#_Toc20323963)

[286\_Walls.java 440](#_Toc20323964)

[287\_FindDuplicateNumber.java 441](#_Toc20323965)

[288\_Unique.java 443](#_Toc20323966)

[289\_GameofLife.java 444](#_Toc20323967)

[29\_Divide.java 446](#_Toc20323968)

[290\_WordPattern.java 448](#_Toc20323969)

[291\_Word.java 449](#_Toc20323970)

[292\_Nim.java 452](#_Toc20323971)

[293\_Flip.java 453](#_Toc20323972)

[294\_Flip.java 453](#_Toc20323973)

[295\_FindMedianDataStream.java 455](#_Toc20323974)

[296\_Best.java 458](#_Toc20323975)

[299\_Bulls.java 461](#_Toc20323976)

[Intervals 463](#_Toc20323977)

[30\_InsertInterval.java 463](#_Toc20323978)

[435\_ Non-overlapping Intervals.java 466](#_Toc20323979)

[436\_ Find Right Interval.java 467](#_Toc20323980)

[252\_MeetingRooms.java 470](#_Toc20323981)

[253\_MeetingRoomsII.java 472](#_Toc20323982)

[30\_Substring.java 475](#_Toc20323983)

[302\_Smallest.java 476](#_Toc20323984)

[303\_Range.java 478](#_Toc20323985)

[304\_Range.java 479](#_Toc20323986)

[31\_NextPermutation.java 481](#_Toc20323987)

[31\_PartitionArray.java 483](#_Toc20323988)

[310\_Minimum.java 484](#_Toc20323989)

[312\_BurstBalloons.java 486](#_Toc20323990)

[313\_SuperUglyNumber.java 489](#_Toc20323991)

[315\_Count.java 490](#_Toc20323992)

[317\_Shortest.java 493](#_Toc20323993)

[318\_Maximum.java 494](#_Toc20323994)

[320\_GeneralizedAbbreviationsOfWord.java 496](#_Toc20323995)

[321\_Create.java 497](#_Toc20323996)

[322\_CoinChange.java 499](#_Toc20323997)

[325\_Maximum.java 501](#_Toc20323998)

[326\_Power.java 501](#_Toc20323999)

[327\_Count.java 502](#_Toc20324000)

[329\_LongestIncreasingPath.java 504](#_Toc20324001)

[33\_SearchinRotatedSortedArray.java 506](#_Toc20324002)

[334\_IncreasingTripletSubsequence.java 510](#_Toc20324003)

[338\_Counting.java 511](#_Toc20324004)

[339\_Nested.java 512](#_Toc20324005)

[34\_Find.java 513](#_Toc20324006)

[342\_Power.java 514](#_Toc20324007)

[343\_Integer.java 515](#_Toc20324008)

[344\_ReverseString.java 516](#_Toc20324009)

[345\_Reverse.java 517](#_Toc20324010)

[346\_Moving.java 518](#_Toc20324011)

[347\_TopKFrequentElements.java 519](#_Toc20324012)

[35\_Search.java 523](#_Toc20324013)

[349\_Intersection.java 524](#_Toc20324014)

[350\_IntersectionII.java 526](#_Toc20324015)

[351\_Android.java 527](#_Toc20324016)

[354\_Russian.java 529](#_Toc20324017)

[356\_Line.java 530](#_Toc20324018)

[357\_Count.java 531](#_Toc20324019)

[358\_Rearrange.java 532](#_Toc20324020)

[359\_DesingLoggerSystem.java 534](#_Toc20324021)

[36\_ValidSudoku.java 536](#_Toc20324022)

[360\_Sort.java 537](#_Toc20324023)

[361\_Bomb.java 538](#_Toc20324024)

[362\_SlidingWindow.java 541](#_Toc20324025)

[362\_SlidingWindowMaximum.java 542](#_Toc20324026)

[363\_Max Sum of Rectangle No Larger Than K.java 544](#_Toc20324027)

[364\_Nested List Weight Sum II.java 545](#_Toc20324028)

[364\_TrappingRainWaterII.java 548](#_Toc20324029)

[365\_Water.java 551](#_Toc20324030)

[367\_Valid.java 554](#_Toc20324031)

[368\_LargestDivisibleSubset.java 555](#_Toc20324032)

[369\_Plus.java 557](#_Toc20324033)

[37\_PartitionArraybyOddandEven.java 560](#_Toc20324034)

[37\_Sudoku.java 560](#_Toc20324035)

[370\_Range Addition.java 562](#_Toc20324036)

[373\_Find.java 565](#_Toc20324037)

[374\_Guess Number Higher or Lower.java 567](#_Toc20324038)

[376\_Wiggle Subsequence.java 568](#_Toc20324039)

[377\_Combination Sum IV.java 571](#_Toc20324040)

[378\_ConvertBSTtoDoublyLinkedList.java 573](#_Toc20324041)

[38\_CountandSay.java 575](#_Toc20324042)

[38\_Search2DMatrixII.java 576](#_Toc20324043)

[380\_Load\_Balancer.java Insert Delete GetRandom O(1) 578](#_Toc20324044)

[381\_Insert Delete GetRandom O(1) - Duplicates allowed.java 579](#_Toc20324045)

[382\_RandomNode.java 581](#_Toc20324046)

[382\_TriangleCount.java 583](#_Toc20324047)

[383\_Ransom.java 584](#_Toc20324048)

[384\_ShuffleArray.java 585](#_Toc20324049)

[385\_Mini.java 586](#_Toc20324050)

[386\_Lexicographical.java 588](#_Toc20324051)

[387\_FirstUniqueChar.java 589](#_Toc20324052)

[389\_Find.java 589](#_Toc20324053)

[39\_CombinationSum.java 591](#_Toc20324054)

[390\_Elimination.java 592](#_Toc20324055)

[391\_NumberSky.java 593](#_Toc20324056)

[392\_IsSubsequence.java 594](#_Toc20324057)

[393\_utf8.java 596](#_Toc20324058)

[394\_CoinsLine.java 598](#_Toc20324059)

[394\_Decode.java 599](#_Toc20324060)

[395\_CoinsII.java 602](#_Toc20324061)

[396\_RotateFunction.java 603](#_Toc20324062)

[397\_Integer.java 604](#_Toc20324063)

[398\_RandomPick.java 605](#_Toc20324064)

[399\_Evaluate.java 606](#_Toc20324065)

[4\_MedianofTwoSortedArrays.java 609](#_Toc20324066)

[40\_CombinationSumII.java 611](#_Toc20324067)

[40\_ImplementQueuebyStacks.java 613](#_Toc20324068)

[400\_Nth.java 614](#_Toc20324069)

[401\_Binary.java 615](#_Toc20324070)

[401\_KthSmallestNumberinSortedMatrix.java 617](#_Toc20324071)

[402\_Continuous.java 620](#_Toc20324072)

[402\_Remove.java 622](#_Toc20324073)

[403\_Frog.java 625](#_Toc20324074)

[405\_Convert.java 626](#_Toc20324075)

[406\_MinimumSizeSubarraySum.java 627](#_Toc20324076)

[406\_Queue.java 628](#_Toc20324077)

[408\_AddBinary.java 629](#_Toc20324078)

[408\_Valid.java 630](#_Toc20324079)

[41\_FirstMissingPositive.java 632](#_Toc20324080)

[410\_Split.java 633](#_Toc20324081)

[412\_FizzBuzz.java 634](#_Toc20324082)

[413\_ArithmeticSlices.java 635](#_Toc20324083)

[414\_ThirdMaximumNumber.java 637](#_Toc20324084)

[415\_Add.java 638](#_Toc20324085)

[416\_Partition.java 640](#_Toc20324086)

[417\_PacificAtlanticOcean.java 643](#_Toc20324087)

[418\_IntegertoRoman.java 646](#_Toc20324088)

[418\_Sentence.java 647](#_Toc20324089)

[419\_RomantoInteger.java 649](#_Toc20324090)

[421\_Maximum.java 651](#_Toc20324091)

[422\_Valid.java 652](#_Toc20324092)

[424\_ Longest Repeating Character Replacement.java 653](#_Toc20324093)

[425\_LetterCombinationsofPhoneNumber.java 654](#_Toc20324094)

[427\_Construct.java 656](#_Toc20324095)

[428\_Pow.java 657](#_Toc20324096)

[428\_Serialize and Deserialize N-ary Tree.java 659](#_Toc20324097)

[429\_N-ary Tree Level Order Traversal.java 661](#_Toc20324098)

[43\_MultiplyStrings.java 662](#_Toc20324099)

[430\_ Flatten a Multilevel Doubly Linked List.java 664](#_Toc20324100)

[433\_ Minimum Genetic Mutation.java 666](#_Toc20324101)

[200\_NumberofIslands.java 669](#_Toc20324102)

[434\_NumberofIslandsII.java 675](#_Toc20324103)

[434\_ Number of Segments in a String.java 676](#_Toc20324104)

[437\_CopyBooks.java 677](#_Toc20324105)

[437\_Path.java 679](#_Toc20324106)

[439\_Ternary.java 680](#_Toc20324107)

[44\_MinimumSubarray.java 682](#_Toc20324108)

[44\_WildcardMatching.java 683](#_Toc20324109)

[440\_K-th.java 684](#_Toc20324110)

[441\_Arranging.java 685](#_Toc20324111)

[442\_Find.java 687](#_Toc20324112)

[443\_StringCompression.java 688](#_Toc20324113)

[444\_Sequence.java 692](#_Toc20324114)

[445\_Add.java 694](#_Toc20324115)

[446\_Arithmetic.java 697](#_Toc20324116)

[447\_Number.java 698](#_Toc20324117)

[448\_Find.java 699](#_Toc20324118)

[450\_Delete.java 701](#_Toc20324119)

[450\_ReverseNodesk-Group.java 704](#_Toc20324120)

[451\_SortCharactersByFrequency.java 707](#_Toc20324121)

[452\_Minimum.java 708](#_Toc20324122)

[453\_Minimum.java 710](#_Toc20324123)

[454\_4SumII.java 711](#_Toc20324124)

[455\_Assign.java 713](#_Toc20324125)

[456\_132Pattern.java 714](#_Toc20324126)

[458\_Poor.java 715](#_Toc20324127)

[46\_NumbersPermutations.java 716](#_Toc20324128)

[461\_Hamming.java 718](#_Toc20324129)

[462\_Minimum.java 718](#_Toc20324130)

[463\_Island.java 719](#_Toc20324131)

[464\_Can.java 720](#_Toc20324132)

[465\_Optimal.java 722](#_Toc20324133)

[467\_Unique.java 724](#_Toc20324134)

[468\_Validate.java 725](#_Toc20324135)

[47\_PermutationsII.java 728](#_Toc20324136)

[470\_Implement.java 729](#_Toc20324137)

[5\_LongestPalindromicSubstring.java 787](#_Toc20324138)

[517\_Super.java 812](#_Toc20324139)

[518\_CoinChange2.java 813](#_Toc20324140)

[52\_N-QueensII.java 815](#_Toc20324141)

[520\_Detect.java 816](#_Toc20324142)

[523\_Continuous.java 818](#_Toc20324143)

[524\_Longest.java 819](#_Toc20324144)

[525\_Contiguous.java 821](#_Toc20324145)

[526\_Beautiful.java 823](#_Toc20324146)

[528\_FlattenNestedListIterator.java 825](#_Toc20324147)

[543\_Diameter.java 852](#_Toc20324148)

[544\_Output.java 855](#_Toc20324149)

[547\_FriendCircles.java 857](#_Toc20324150)

[548\_Split.java 860](#_Toc20324151)

[551\_Student.java 862](#_Toc20324152)

[552\_Student.java 863](#_Toc20324153)

[553\_OptimalDivision.java 865](#_Toc20324154)

[554\_Brick.java 866](#_Toc20324155)

[555\_Split.java 867](#_Toc20324156)

[556\_Next.java 869](#_Toc20324157)

[559\_Maximum.java 870](#_Toc20324158)

[560\_SubarraySumEqualsK.java 871](#_Toc20324159)

[561\_ArrayPartitionI.java 872](#_Toc20324160)

[562\_Longest.java 873](#_Toc20324161)

[563\_Binary.java 876](#_Toc20324162)

[565\_Array.java 878](#_Toc20324163)

[566\_Reshape.java 880](#_Toc20324164)

[567\_Permutation.java 881](#_Toc20324165)

[568\_Maximum.java 883](#_Toc20324166)

[57\_ThreeSum.java 885](#_Toc20324167)

[572\_SubtreeofAnotherTree.java 887](#_Toc20324168)

[575\_Distribute.java 888](#_Toc20324169)

[575\_ExpressionExpand.java 890](#_Toc20324170)

[576\_Out.java 891](#_Toc20324171)

[581\_Shortest.java 893](#_Toc20324172)

[582\_Kill.java 896](#_Toc20324173)

[583\_Delete.java 898](#_Toc20324174)

[588\_Design.java 899](#_Toc20324175)

[589\_N-ary.java 903](#_Toc20324176)

[59\_SpiralMatrixII.java 904](#_Toc20324177)

[590\_N-ary.java 905](#_Toc20324178)

[594\_Longest.java 906](#_Toc20324179)

[599\_Minimum.java 907](#_Toc20324180)

[6\_MergeTwoSortedArrays.java 909](#_Toc20324181)

[6\_ZigZagConversion.java 909](#_Toc20324182)

[60\_Permutation.java 910](#_Toc20324183)

[600\_Non-negative.java 912](#_Toc20324184)

[604\_Design.java 914](#_Toc20324185)

[604\_WindowSum.java 916](#_Toc20324186)

[605\_Can.java 916](#_Toc20324187)

[606\_ConstructStringfromBinaryTree.java 918](#_Toc20324188)

[609\_Find.java 919](#_Toc20324189)

[61\_SearchforaRange.java 920](#_Toc20324190)

[611\_ValidTriangleNumber.java 921](#_Toc20324191)

[613\_HighFive.java 922](#_Toc20324192)

[616\_Add.java 923](#_Toc20324193)

[621\_Task.java 925](#_Toc20324194)

[622\_GuessNumberGame.java 927](#_Toc20324195)

[624\_Maximum.java 928](#_Toc20324196)

[625\_Minimum.java 929](#_Toc20324197)

[626\_RectangleOverlap.java 930](#_Toc20324198)

[627\_LongestPalindrome.java 931](#_Toc20324199)

[628\_Maximum.java 932](#_Toc20324200)

[628\_MaximumSubtree.java 934](#_Toc20324201)

[629\_MinimumSpanningTree.java 935](#_Toc20324202)

[63\_UniquePathsII.java 936](#_Toc20324203)

[630\_Course.java 938](#_Toc20324204)

[632\_Smallest.java 939](#_Toc20324205)

[633\_Sum.java 941](#_Toc20324206)

[634\_Find.java 942](#_Toc20324207)

[634\_WordSquares.java 943](#_Toc20324208)

[635\_Design.java 946](#_Toc20324209)

[636\_Exclusive.java 947](#_Toc20324210)

[637\_Average.java 949](#_Toc20324211)

[637\_CheckWordAbbreviation.java 950](#_Toc20324212)

[638\_Shopping.java 951](#_Toc20324213)

[638\_StringsHomomorphism.java 955](#_Toc20324214)

[639\_Decode.java 957](#_Toc20324215)

[639\_WordsAbbreviation.java 959](#_Toc20324216)

[64\_MinimumPathSum.java 960](#_Toc20324217)

[640\_EditDistanceII.java 961](#_Toc20324218)

[641\_DesignDeque.java 962](#_Toc20324219)

[641\_MissingInterval.java 964](#_Toc20324220)

[642\_Design.java 965](#_Toc20324221)

[642\_SlidingWindowAveragefromDataStream.java 968](#_Toc20324222)

[643\_MaximumI.java 969](#_Toc20324223)

[643\_SystemLongestFilePath.java 971](#_Toc20324224)

[644\_MaximumII.java 972](#_Toc20324225)

[644\_MirrorNumbers.java 976](#_Toc20324226)

[645\_IdentifyCelebrity.java 977](#_Toc20324227)

[645\_SetMismatch.java 978](#_Toc20324228)

[646\_FirstPositionUniqueCharacter.java 979](#_Toc20324229)

[646\_MaximumLengthofPairChain.java 980](#_Toc20324230)

[647\_Palindromic.java 981](#_Toc20324231)

[647\_SubstringAnagrams.java 982](#_Toc20324232)

[648\_Replace.java 984](#_Toc20324233)

[648\_WordAbbreviationSet.java 986](#_Toc20324234)

[649\_BinaryTreeFlipping.java 988](#_Toc20324235)

[65\_Valid.java 989](#_Toc20324236)

[650\_BinaryTreeLeavesOrderTraversal.java 990](#_Toc20324237)

[650\_Keys.java 992](#_Toc20324238)

[651\_BinaryTreeVerticalOrderTraversal.java 993](#_Toc20324239)

[651\_Keys.java 995](#_Toc20324240)

[652\_Factorization.java 996](#_Toc20324241)

[652\_Find.java 998](#_Toc20324242)

[653\_AddOperators.java 1000](#_Toc20324243)

[653\_Two.java 1001](#_Toc20324244)

[654\_Maximum.java 1004](#_Toc20324245)

[654\_SparseMatrixMultiplication.java 1006](#_Toc20324246)

[655\_BigIntegerAddition.java 1008](#_Toc20324247)

[655\_Print.java 1009](#_Toc20324248)

[656\_BigIntegerMultiplication.java 1012](#_Toc20324249)

[656\_Coin.java 1013](#_Toc20324250)

[657\_Judge.java 1015](#_Toc20324251)

[658\_Find.java 1016](#_Toc20324252)

[658\_FindKClosestElements.java 1017](#_Toc20324253)

[659\_Split.java 1018](#_Toc20324254)

[659\_StringsSerialization.java 1020](#_Toc20324255)

[66\_PlusOne.java 1023](#_Toc20324256)

[660\_ReadCharactersFromFile.java 1023](#_Toc20324257)

[660\_Remove.java 1024](#_Toc20324258)

[661\_ConverBSTtoGreaterTree.java 1025](#_Toc20324259)

[661\_ImageSmoother.java 1026](#_Toc20324260)

[662\_MaximumWidthofBinaryTree.java 1027](#_Toc20324261)

[663\_EqualTreePartition.java 1030](#_Toc20324262)

[663\_NearestExit.java 1032](#_Toc20324263)

[665\_Nondecreasing.java 1034](#_Toc20324264)

[666\_Path.java 1036](#_Toc20324265)

[668\_Kth.java 1037](#_Toc20324266)

[669\_Trim.java 1040](#_Toc20324267)

[670\_Maximum.java 1042](#_Toc20324268)

[675\_CutOffTreesforGolfEvent.java 1045](#_Toc20324269)

[676\_Implement.java 1048](#_Toc20324270)

[677\_Map.java 1050](#_Toc20324271)

[678\_ValidParenthesisString.java 1054](#_Toc20324272)

[679\_Game.java 1056](#_Toc20324273)

[68\_Text.java 1057](#_Toc20324274)

[680\_Valid.java 1061](#_Toc20324275)

[681\_Next.java 1062](#_Toc20324276)

[682\_BaseballGame.java 1063](#_Toc20324277)

[683\_Empty.java 1065](#_Toc20324278)

[684\_Redundant.java 1067](#_Toc20324279)

[685\_Redundant.java 1069](#_Toc20324280)

[686\_RepeatedStringMatch.java 1071](#_Toc20324281)

[687\_Longest.java 1072](#_Toc20324282)

[688\_KnightProbabilityinChessboard.java 1075](#_Toc20324283)

[689\_Maximum.java 1077](#_Toc20324284)

[690\_Employee.java 1080](#_Toc20324285)

[691\_Stickers.java 1082](#_Toc20324286)

[692\_TopKFrequentWords.java 1085](#_Toc20324287)

[693\_Binary.java 1086](#_Toc20324288)

[694\_NumberIslands.java 1087](#_Toc20324289)

[695\_Max.java 1089](#_Toc20324290)

[696\_Count.java 1092](#_Toc20324291)

[697\_DegreeofArray.java 1094](#_Toc20324292)

[698\_Partition.java 1095](#_Toc20324293)

[7\_ReverseInteger.java 1098](#_Toc20324294)

[70\_ClimbingStairs.java 1099](#_Toc20324295)

[700\_Search.java 1100](#_Toc20324296)

[701\_Insert.java 1101](#_Toc20324297)

[702\_Search.java 1104](#_Toc20324298)

[703\_Kth.java 1105](#_Toc20324299)

[704\_Binary.java 1107](#_Toc20324300)

[705\_DesignHashSet.java 1108](#_Toc20324301)

[706\_DesignHashMap.java 1110](#_Toc20324302)

[707\_DesignLinkedList.java 1112](#_Toc20324303)

[708\_insert.java 1114](#_Toc20324304)

[71\_Simplify.java 1116](#_Toc20324305)

[712\_Minimum.java 1117](#_Toc20324306)

[713\_SubarrayProductLessThanK.java 1118](#_Toc20324307)

[714\_BestTimewithTransactionFee.java 1120](#_Toc20324308)

[715\_Range.java 1121](#_Toc20324309)

[716\_MaxStack.java 1122](#_Toc20324310)

[717\_bit.java 1123](#_Toc20324311)

[718\_Maximum.java 1124](#_Toc20324312)

[719\_FindK-th.java 1125](#_Toc20324313)

[72\_EditDistance.java 1127](#_Toc20324314)

[720\_Longest.java 1128](#_Toc20324315)

[721\_Accounts.java 1132](#_Toc20324316)

[722\_Remove.java 1136](#_Toc20324317)

[723\_Candy.java 1138](#_Toc20324318)

[724\_Find.java 1140](#_Toc20324319)

[725\_SplitLinkedListinParts.java 1141](#_Toc20324320)

[726\_Number.java 1142](#_Toc20324321)

[727\_Minimum.java 1145](#_Toc20324322)

[728\_Self.java 1146](#_Toc20324323)

[728\_ThreeDistinctFactors.java 1147](#_Toc20324324)

[729\_My.java 1148](#_Toc20324325)

[73\_SetMatrixZeroes.java 1151](#_Toc20324326)

[730\_Count.java 1152](#_Toc20324327)

[731\_My.java 1155](#_Toc20324328)

[732\_my.java 1157](#_Toc20324329)

[733\_FloodFill.java 1158](#_Toc20324330)

[734\_Sentence.java 1160](#_Toc20324331)

[735\_Asteroid.java 1163](#_Toc20324332)

[735\_maximumFromRight.java 1165](#_Toc20324333)

[737\_Sentence.java 1166](#_Toc20324334)

[738\_Monotone.java 1168](#_Toc20324335)

[739\_Daily.java 1170](#_Toc20324336)

[74\_FirstBadVersion.java 1171](#_Toc20324337)

[74\_Search.java 1172](#_Toc20324338)

[74\_Search2DMatrix.java 1173](#_Toc20324339)

[740\_Delete.java 1176](#_Toc20324340)

[742\_ClosestLeaf.java 1178](#_Toc20324341)

[743\_Network.java 1180](#_Toc20324342)

[744\_Find.java 1183](#_Toc20324343)

[745\_Prefix.java 1185](#_Toc20324344)

[746\_MinCostClimbingStairs.java 1189](#_Toc20324345)

[748\_Shortest.java 1191](#_Toc20324346)

[75\_SortColors.java 1192](#_Toc20324347)

[750\_Number.java 1193](#_Toc20324348)

[752\_Open.java 1195](#_Toc20324349)

[753\_Cracking.java 1199](#_Toc20324350)

[754\_Reach.java 1200](#_Toc20324351)

[755\_Pour.java 1202](#_Toc20324352)

[756\_Pyramid.java 1205](#_Toc20324353)

[758\_Bold.java 1207](#_Toc20324354)

[759\_Employee.java 1209](#_Toc20324355)

[760\_Find.java 1210](#_Toc20324356)

[787\_Cheapest.java 1252](#_Toc20324357)

[788\_Rotated.java 1254](#_Toc20324358)

[791\_CustomSortString.java 1256](#_Toc20324359)

[792\_Number.java 1257](#_Toc20324360)

[793\_Preimage.java 1259](#_Toc20324361)

[794\_Valid.java 1260](#_Toc20324362)

[795\_Number.java 1262](#_Toc20324363)

[796\_Rotate.java 1263](#_Toc20324364)

[797\_All.java 1264](#_Toc20324365)

[798\_Smallest.java 1266](#_Toc20324366)

[799\_Champagne.java 1268](#_Toc20324367)

[8\_StringtoInteger.java 1269](#_Toc20324368)

[80\_RemoveDuplicatesII.java 1269](#_Toc20324369)

[800\_Similar.java 1270](#_Toc20324370)

[801\_Minimum.java 1271](#_Toc20324371)

[802\_Find.java 1274](#_Toc20324372)

[804\_Unique.java 1276](#_Toc20324373)

[805\_MaximumAssociationSet.java 1277](#_Toc20324374)

[805\_Split.java 1278](#_Toc20324375)

[806\_Number.java 1279](#_Toc20324376)

[807\_Max.java 1281](#_Toc20324377)

[809\_Expressive.java 1282](#_Toc20324378)

[81\_DataStream.java 1284](#_Toc20324379)

[81\_searchRotated.java 1285](#_Toc20324380)

[810\_Chalkboard.java 1289](#_Toc20324381)

[811\_Subdomain.java 1290](#_Toc20324382)

[812\_Largest.java 1292](#_Toc20324383)

[813\_Largest.java 1293](#_Toc20324384)

[814\_Binary.java 1295](#_Toc20324385)

[815\_Bus.java 1297](#_Toc20324386)

[816\_Ambiguous.java 1299](#_Toc20324387)

[819\_Most.java 1300](#_Toc20324388)

[82\_Remove.java 1303](#_Toc20324389)

[820\_Short.java 1304](#_Toc20324390)

[821\_Shortest.java 1305](#_Toc20324391)

[822\_Card.java 1307](#_Toc20324392)

[823\_Binary.java 1308](#_Toc20324393)

[824\_Goat.java 1311](#_Toc20324394)

[825\_Friends.java 1312](#_Toc20324395)

[826\_Most.java 1314](#_Toc20324396)

[827\_Making.java 1315](#_Toc20324397)

[828\_Unique.java 1319](#_Toc20324398)

[829\_Consecutive.java 1322](#_Toc20324399)

[830\_Positions.java 1323](#_Toc20324400)

[831\_Masking.java 1324](#_Toc20324401)

[832\_Flipping.java 1327](#_Toc20324402)

[833\_Find.java 1328](#_Toc20324403)

[834\_Sum.java 1329](#_Toc20324404)

[835\_Image.java 1332](#_Toc20324405)

[837\_New.java 1334](#_Toc20324406)

[838\_Push.java 1335](#_Toc20324407)

[839\_Similar.java 1338](#_Toc20324408)

[84\_LargestRectangle.java 1339](#_Toc20324409)

[840\_Magic.java 1340](#_Toc20324410)

[841\_Keys.java 1342](#_Toc20324411)

[842\_Split.java 1343](#_Toc20324412)

[843\_Guess.java 1345](#_Toc20324413)

[844\_Backspace.java 1347](#_Toc20324414)

[845\_Longest.java 1350](#_Toc20324415)

[846\_Hand.java 1353](#_Toc20324416)

[847\_Shortest.java 1354](#_Toc20324417)

[848\_Shifting.java 1356](#_Toc20324418)

[849\_Maximize.java 1357](#_Toc20324419)

[85\_MaximalRectangle.java 1360](#_Toc20324420)

[851\_Loud.java 1361](#_Toc20324421)

[852\_Peak.java 1364](#_Toc20324422)

[853\_Car.java 1366](#_Toc20324423)

[854\_Similar.java 1367](#_Toc20324424)

[855\_Exam.java 1369](#_Toc20324425)

[856\_Score.java 1371](#_Toc20324426)

[857\_Minimum.java 1373](#_Toc20324427)

[858\_Mirror.java 1375](#_Toc20324428)

[859\_Buddy.java 1376](#_Toc20324429)

[86\_BSTIterator.java 1378](#_Toc20324430)

[860\_Lemonade.java 1380](#_Toc20324431)

[861\_Score.java 1382](#_Toc20324432)

[862\_Shortest.java 1384](#_Toc20324433)

[863\_All.java 1386](#_Toc20324434)

[864\_Shortest.java 1389](#_Toc20324435)

[865\_Smallest.java 1392](#_Toc20324436)

[867\_Transpose.java 1395](#_Toc20324437)

[869\_Reordered.java 1398](#_Toc20324438)

[87\_ScrambleString.java 1400](#_Toc20324439)

[870\_Advantage.java 1401](#_Toc20324440)

[871\_Minimum.java 1403](#_Toc20324441)

[872\_Leaf-Similar.java 1405](#_Toc20324442)

[873\_Length.java 1406](#_Toc20324443)

[874\_WalkingRobot.java 1408](#_Toc20324444)

[875\_Koko.java 1411](#_Toc20324445)

[876\_Middle.java 1413](#_Toc20324446)

[877\_Stone.java 1414](#_Toc20324447)

[878\_Nth.java 1415](#_Toc20324448)

[879\_Profitable.java 1417](#_Toc20324449)

[88\_MergeSortedArray.java 1419](#_Toc20324450)

[880\_Decoded.java 1419](#_Toc20324451)

[881\_Boats.java 1420](#_Toc20324452)

[883\_Projection.java 1422](#_Toc20324453)

[884\_Uncommon.java 1423](#_Toc20324454)

[885\_Spiral.java 1424](#_Toc20324455)

[886\_Possible.java 1426](#_Toc20324456)

[887\_Super.java 1429](#_Toc20324457)

[888\_Fair.java 1430](#_Toc20324458)

[889\_Construct.java 1433](#_Toc20324459)

[89\_GrayCode.java 1435](#_Toc20324460)

[89\_kSum.java 1436](#_Toc20324461)

[890\_Find.java 1438](#_Toc20324462)

[891\_Sum.java 1440](#_Toc20324463)

[892\_Surface.java 1441](#_Toc20324464)

## 11\_ContainerWater.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai).

n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0).

Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and n is at least 2.

Initially we consider the area constituting the exterior most lines. Now, to maximize the area,

we need to consider the area between the lines of larger lengths. If we try to move the pointer at the longer

line inwards, we won't gain any increase in area, since it is limited by the shorter line. But moving the shorter line's

pointer could turn out to be beneficial, as per the same argument, despite the reduction in the width.

This is done since a relatively longer line obtained by moving the shorter line's pointer might overcome the reduction

in area caused by the width reduction.

class Solution {

public int maxArea(int[] height) {

int left = 0;

int right = height.length - 1;

int max = 0;

while (left < right){

int area = (right - left) \* Math.min(height[left], height[right]);

max = Math.max(max, area);

if (height[left] < height[right]){

left++;

}else{

right--;

}

}

return max;

}

}

class Solution {

public int maxArea(int[] height) {

int left = 0;

int right = height.length - 1;

int max = 0;

while (left < right){

if (height[left] < height[right]){

max = Math.max(max, (right - left) \* height[left]);

left++;

}else{

max = Math.max(max, (right - left) \* height[right]);

right--;

}

}

return max;

}

}

## 42\_TrappingRainWater.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n non-negative integers representing an elevation map where the width of each bar is 1,

compute how much water it is able to trap after raining.

For example,

Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

Best Solution:

class Solution {

public int trap(int[] height) {

int left = 0;

int right = height.length - 1;

int i = left;

int j = right;

int res = 0;

while (i <= j){

if (height[left] <= height[right]){

if (height[i] >= height[left]){

left = i;

}else{

res += height[left] - height[i];

}

i++;

}else{

if (height[j] >= height[right]){

right = j;

}else{

res += height[right] - height[j];

}

j--;

}

}

return res;

}

}

class Solution {

public int trap(int[] height) {

if (height == null || height.length < 3){

return 0;

}

int ans = 0;

int l = 0;

int r = height.length - 1;

while (l < r && height[l] < height[l+1]){

l++;

}

while (l < r && height[r] < height[r-1]){

r--;

}

while (l < r){

int left = height[l];

int right = height[r];

if (left < right){

while (left > height[++l]){

ans += left - height[l];

}

}else{

while (right > height[--r]){

ans += right - height[r];

}

}

}

return ans;

}

}

Lintcode: 363

public class Solution {

/\*\*

\* @param heights: a list of integers

\* @return: a integer

\*/

public int trapRainWater(int[] heights) {

if (heights == null || heights.length <= 1){

return 0;

}

int left = 0;

int right = heights.length - 1;

int ans = 0;

int leftHeight = heights[left];

int rightHeight = heights[right];

while (left < right){

if (leftHeight < rightHeight){

left++;

if (leftHeight > heights[left]){

ans += leftHeight - heights[left];

}else{

leftHeight = heights[left];

}

}else{

right--;

if (rightHeight > heights[right]){

ans += rightHeight - heights[right];

}else{

rightHeight = heights[right];

}

}

}

return ans;

}

}

class Solution {

public int trap(int[] height) {

if (height == null || height.length < 3){

return 0;

}

int left = 0;

int right = height.length - 1;

int leftHeight = height[left];

int rightHeight = height[right];

int ans = 0;

while (left < right){

if (leftHeight < rightHeight){

left++;

ans += Math.max(0, leftHeight - height[left]);

leftHeight = Math.max(leftHeight, height[left]);

}else{

right--;

ans += Math.max(0, rightHeight - height[right]);

rightHeight = Math.max(rightHeight, height[right]);

}

}

return ans;

}

}

class Solution {

public int trap(int[] height) {

int n = height.length;

int left = 0;

int right = n - 1;

int i = 0;

int j = n - 1;

int res = 0;

while (i <= j){

if (height[left] < height[right]){

if (height[i] <= height[left]){

res += height[left] - height[i];

i++;

}else{

left = i;

}

}else{

if (height[j] <= height[right]){

res += height[right] - height[j];

j--;

}else{

right = j;

}

}

}

return res;

}

}

## 62\_UniquePaths.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach

the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Above is a 3 x 7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

class Solution {

public int uniquePaths(int m, int n) {

if (m == 0 || n == 0){

return 1;

}

int[][] f = new int[m][n];

f[0][0] = 1;

for (int i = 1; i < m; i++){

f[i][0] = 1;

}

for (int j = 1; j < n; j ++){

f[0][j] = 1;

}

for (int i = 1; i < m; i++){

for (int j = 1; j < n; j++){

f[i][j] = f[i-1][j] + f[i][j-1];

}

}

return f[m-1][n-1];

}

}

Better version:

class Solution {

public int uniquePaths(int m, int n) {

int[][] dp = new int[m][n];

for (int i = 0; i < m; i++){

dp[i][0] = 1;

}

for (int j = 0; j < n; j++){

dp[0][j] = 1;

}

for (int i = 1; i < m; i++){

for (int j = 1; j < n; j++){

dp[i][j] = dp[i-1][j] + dp[i][j-1];

}

}

return dp[m-1][n-1];

}

}

# Sum Target

## 1\_TwoSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Two Sum

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example:

Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,

return [0, 1].

--------------------------------------------

Method 1: HashMap

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int[] twoSum(int[] nums, int target) {

int[] result = new int[2];

if (nums == null || nums.length == 0){

return result;

}

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++){

if (map.containsKey(target - nums[i])){

result[0] = map.get(target - nums[i]);

result[1] = i;

return result;

}

map.put(nums[i], i);

}

return result;

}

}

Method 2: Sort + two points

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public int[] twoSum(int[] nums, int target) {

int[] result = new int[2];

int[] number = new int[2];

if (nums == null || nums.length == 0){

return result;

}

int[] origin = new int[nums.length];

for (int i = 0; i < nums.length; i++){

origin[i] = nums[i];

}

Arrays.sort(origin);

int left = 0;

int right = nums.length -1;

while (left < right){

int sum = origin[left] + origin[right];

if (sum == target){

number[0] = origin[left];

number[1] = origin[right];

break;

}else if (sum < target){

left++;

}else{

right--;

}

}

for (int i = 0; i < nums.length; i++){

if (nums[i] == number[0]){

result[0] = i;

break;

}

}

for (int i = nums.length - 1; i >= 0; i--){

if (nums[i] == number[1]){

result[1] = i;

break;

}

}

return result;

}

}

## 167\_TwoSumII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers that is already sorted in ascending order, find two numbers such that they add up

to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1

must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution and you may not use the same element twice.

Input: numbers={2, 7, 11, 15}, target=9

Output: index1=1, index2=2

class Solution {

public int[] twoSum(int[] numbers, int target) {

int[] result = new int[2];

if (numbers == null || numbers.length == 0){

return result;

}

int left = 0;

int right = numbers.length - 1;

while (left < right){

int sum = numbers[left] + numbers[right];

if (sum == target){

result[0] = left + 1;

result[1] = right + 1;

return result;

}else if (sum < target){

left++;

}else{

right--;

}

}

return result;

}

}

## 15\_3Sum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

Note: The solution set must not contain duplicate triplets.

For example, given array S = [-1, 0, 1, 2, -1, -4],

A solution set is:

[

[-1, 0, 1],

[-1, -1, 2]

]

Time complexity: O(N^2)

class Solution {

public List<List<Integer>> threeSum(int[] nums) {

List<List<Integer>> result = new ArrayList<>();

if (nums == null){

return result;

}

Arrays.sort(nums);

for (int i = 0; i < nums.length - 2; i++){

if (i != 0 && nums[i-1] == nums[i]){

continue;

}

int left = i + 1;

int right = nums.length - 1;

int target = - nums[i];

while (left < right){

int sum = nums[left] + nums[right];

if (sum == target){

List<Integer> item = new ArrayList<>();

item.add(nums[i]);

item.add(nums[left]);

item.add(nums[right]);

result.add(item);

left++;

right--;

while (left < right && nums[left-1] == nums[left]){

left++;

}

while (left < right && nums[right+1] == nums[right]){

right--;

}

}else if (sum < target){

left++;

}else{

right--;

}

}

}

return result;

}

}

## 16\_3SumClosest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array S of n integers, find three integers in S such that the sum is closest to a given number,

target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

For example, given array S = {-1 2 1 -4}, and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

class Solution {

public int threeSumClosest(int[] nums, int target) {

Arrays.sort(nums);

int best = Integer.MAX\_VALUE;

int result = Integer.MAX\_VALUE;

for (int i = 0; i < nums.length - 2; i++){

if (i > 0 && nums[i-1] == nums[i]){

continue;

}

int left = i + 1;

int right = nums.length - 1;

while (left < right){

int sum = nums[i] + nums[left] + nums[right];

if (sum == target){

return sum;

}else if (sum < target){

left++;

}else{

right--;

}

if (Math.abs(sum - target) < best){

best = Math.abs(sum- target);

result = sum;

}

}

}

return result;

}

}

class Solution {

public int threeSumClosest(int[] nums, int target) {

Arrays.sort(nums);

int min = Integer.MAX\_VALUE;

int res = 0;

for (int i = 2; i < nums.length; i++){

int left = 0;

int right = i - 1;

while (left < right){

int sum = nums[i] + nums[left] + nums[right];

if (sum == target){

return sum;

}else if (sum > target){

right--;

}else{

left++;

}

if (Math.abs(sum - target) < min){

res = sum;

min = Math.abs(sum - target);

}

}

}

return res;

}

}

## 18\_4Sum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target?

Find all unique quadruplets in the array which gives the sum of target.

Note: The solution set must not contain duplicate quadruplets.

For example, given array S = [1, 0, -1, 0, -2, 2], and target = 0.

A solution set is:

[

[-1, 0, 0, 1],

[-2, -1, 1, 2],

[-2, 0, 0, 2]

]

Method 1:

Time complexity: O(n^3)

Space complexity: O(1)

class Solution {

public List<List<Integer>> fourSum(int[] nums, int target) {

List<List<Integer>> result = new ArrayList<>();

Arrays.sort(nums);

for (int i = 0; i < nums.length -3 ; i++){

if (i > 0 && nums[i-1] == nums[i]){

continue;

}

threeSum(result, nums, target - nums[i], i);

}

return result;

}

private void threeSum(List<List<Integer>> result, int[] nums, int target, int start){

for (int i = start + 1; i < nums.length - 2; i++){

if (i > start + 1 && nums[i-1] == nums[i]){

continue;

}

int left = i + 1;

int right = nums.length - 1;

while (left < right){

int sum = nums[i] + nums[left] + nums[right];

if (sum == target){

List<Integer> item = new ArrayList<>();

item.add(nums[start]);

item.add(nums[i]);

item.add(nums[left]);

item.add(nums[right]);

result.add(item);

left++;

right--;

while (left < right && nums[left-1] == nums[left]){

left++;

}

while (left < right && nums[right+1] == nums[right]){

right--;

}

}else if (sum < target){

left++;

}else{

right--;

}

}

}

}

}

# Strings

## 3\_LongestSubstringWithoutRepeatingCharacters.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, find the length of the longest substring without repeating characters.

Examples:

Given "abcabcbb", the answer is "abc", which the length is 3.

Given "bbbbb", the answer is "b", with the length of 1.

Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a substring, "pwke" is a subsequence and

not a substring.

Method: the basic idea is, keep a hashmap which stores the characters in string as keys and their positions as values, and

keep two pointers which define the max substring. move the right pointer to scan through the string , and meanwhile

update the hashmap. If the character is already in the hashmap, then move the left pointer to the right of the

same character last found. Note that the two pointers can only move forward.

class Solution {

public int lengthOfLongestSubstring(String s) {

if (s == null | s.length() == 0){

return 0;

}

int count = 0;

int max = Integer.MIN\_VALUE;

int start = 0;

Map<Character, Integer> map = new HashMap<Character, Integer>();

for (int i = 0; i < s.length(); i++){

if (!map.containsKey(s.charAt(i))){

map.put(s.charAt(i), i);

count++;

}else{

start = Math.max(start, map.get(s.charAt(i)) + 1) ;;

count = i - start + 1;

map.put(s.charAt(i), i);

}

max = Math.max(max, count);

}

return max;

}

}

Method 2: Sliding window

class Solution {

public int lengthOfLongestSubstring(String s) {

if (s == null || s.length() == 0){

return 0;

}

Map<Character, Integer> map = new HashMap<>();

int start = 0;

int end = 0;

int ans = 0;

int count = 0;

while (end < s.length()){

char c = s.charAt(end);

map.put(c, map.getOrDefault(c, 0) + 1);

if (map.get(c) > 1){

count++;

}

end++;

while (count > 0){

char cstart = s.charAt(start);

if (map.get(cstart) > 1){

count--;

}

map.put(cstart, map.get(cstart) - 1);

start++;

}

ans = Math.max(ans, end - start);

}

return ans;

}

}

Best solution:

class Solution {

public int lengthOfLongestSubstring(String s) {

int res = 0;

int start = 0;

int end = 0;

Map<Character, Integer> map = new HashMap<>();

int count = 0;

while (end < s.length()){

char cEnd = s.charAt(end);

map.put(cEnd, map.getOrDefault(cEnd, 0) + 1);

if (map.get(cEnd) == 2){

count++;

}

end++;

while (count > 0){

char cStart = s.charAt(start);

if (map.get(cStart) == 2){

count--;

}

map.put(cStart, map.get(cStart) - 1);

start++;

}

res = Math.max(res, end - start);

}

return res;

}

}

Sliding window

class Solution {

public int lengthOfLongestSubstring(String s) {

int res = 0;

int start = 0;

int end = 0;

Map<Character, Integer> map = new HashMap<>();

while (end < s.length()){

char cEnd= s.charAt(end);

map.put(cEnd, map.getOrDefault(cEnd, 0) + 1);

while (start < end && map.get(cEnd) > 1){

res = Math.max(res, end - start);

char cStart = s.charAt(start);

map.put(cStart, map.get(cStart) - 1);

start++;

}

end++;

}

res = Math.max(res, end - start);

return res;

}

}

## 10\_RegularExpressionMatching.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement regular expression matching with support for '.' and '\*'.

'.' Matches any single character.

'\*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

isMatch("aaa","aa") → false

isMatch("aa", "a\*") → true

isMatch("aa", ".\*") → true

isMatch("ab", ".\*") → true

isMatch("aab", "c\*a\*b") → true

https://leetcode.com/problems/regular-expression-matching/discuss/5651/Easy-DP-Java-Solution-with-detailed-Explanation

Best version to match the rationale

dp[i][j] denotes match or not when we check the two substrings: p.substring(0, i) and s.substring(0, j)

1, If p.charAt(i-1) == s.charAt(j-1) : dp[i][j] = dp[i-1][j-1];

2, If p.charAt(i-1) == '.' : dp[i][j] = dp[i-1][j-1];

3, If p.charAt(i-1) == '\*':

1 if p.charAt(i-2) != s.charAt(j-1) : dp[i][j] = dp[i-2][j] //in this case, \* only counts as empty

2 if p.charAt(i-2) == s.charAt(j-1) or p.charAt(i-2) == '.':

dp[i][j] = dp[i-2][j] //in this case, \* counts as 0

or dp[i][j] = dp[i-2][j-1] // in this case, \* counts as single a

or dp[i][j] = dp[i][j-1] // in this case, \* counts as multiple a

class Solution {

public boolean isMatch(String s, String p) {

int m = p.length();

int n = s.length();

boolean[][] dp = new boolean[m+1][n+1];

dp[0][0] = true;

for (int i = 1; i <= m; i++){

if (p.charAt(i-1) == '\*'){

dp[i][0] = dp[i-1][0] || dp[i-2][0];

}

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (p.charAt(i-1) == s.charAt(j-1) || p.charAt(i-1) == '.'){

dp[i][j] = dp[i-1][j-1];

}else if (p.charAt(i-1) == '\*'){

if (i > 1 && p.charAt(i-2) != s.charAt(j-1) && p.charAt(i-2) != '.'){

dp[i][j] = dp[i-2][j];

}else if (i > 1 && (p.charAt(i-2) == s.charAt(j-1) || p.charAt(i-2) == '.')){

dp[i][j] = dp[i-2][j] || dp[i-2][j-1] || dp[i][j-1];

}else{//not needed

dp[i][j] = dp[i][j-1];

}

}

}

}

return dp[m][n];

}

}

1, If p.charAt(j) == s.charAt(i) : dp[i][j] = dp[i-1][j-1];

2, If p.charAt(j) == '.' : dp[i][j] = dp[i-1][j-1];

3, If p.charAt(j) == '\*':

here are two sub conditions:

1 if p.charAt(j-1) != s.charAt(i) : dp[i][j] = dp[i][j-2] //in this case, a\* only counts as empty

2 if p.charAt(i-1) == s.charAt(i) or p.charAt(i-1) == '.':

dp[i][j] = dp[i-1][j] //in this case, a\* counts as multiple a

or dp[i][j] = dp[i][j-1] // in this case, a\* counts as single a

or dp[i][j] = dp[i][j-2] // in this case, a\* counts as empty

class Solution {

public boolean isMatch(String s, String p) {

if (s == null || p == null){

return false;

}

int m = s.length();

int n = p.length();

boolean[][] dp = new boolean[m+1][n+1];

dp[0][0] = true;

char[] sc = s.toCharArray();

char[] pc = p.toCharArray();

for (int i = 1; i <= n; i++){

if (pc[i-1] == '\*'){

if (dp[0][i-1] || (i >= 2 && dp[0][i-2])){

dp[0][i] = true;

}

}

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (sc[i-1] == pc[j-1] || pc[j-1] == '.'){

dp[i][j] = dp[i-1][j-1];

}else if (pc[j-1] == '\*'){

if (j >= 2 && sc[i-1] != pc[j-2] && pc[j-2] != '.'){

dp[i][j] = dp[i][j-2];

}else{

dp[i][j] = dp[i][j-1] || dp[i-1][j] || dp[i][j-2];

}

}

}

}

return dp[m][n];

}

}

class Solution {

public boolean isMatch(String s, String p) {

int m = s.length();

int n = p.length();

boolean[][] dp = new boolean[m+1][n+1];

dp[0][0] = true;

char[] sc = s.toCharArray();

char[] pc = p.toCharArray();

for (int i = 1; i <= n; i++){

if (pc[i-1] == '\*'){

dp[0][i] = dp[0][i-1] || dp[0][i-2];

}

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (sc[i-1] == pc[j-1] || pc[j-1] == '.'){

dp[i][j] = dp[i-1][j-1];

}else if (pc[j-1] == '\*'){

if (j >= 2 && sc[i-1] != pc[j-2] && pc[j-2] != '.'){

dp[i][j] = dp[i][j-2];

}else{

dp[i][j] = dp[i][j-1] || dp[i][j-2] || dp[i-1][j];

}

}

}

}

return dp[m][n];

}

}

Better version:

class Solution {

public boolean isMatch(String s, String p) {

int m = s.length();

int n = p.length();

boolean[][] dp = new boolean[m+1][n+1];

dp[0][0] = true;

char[] sc = s.toCharArray();

char[] pc = p.toCharArray();

for (int i = 1; i <= n; i++){

if (pc[i-1] == '\*'){

dp[0][i] = dp[0][i-1] || dp[0][i-2];

}

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (sc[i-1] == pc[j-1] || pc[j-1] == '.'){

dp[i][j] = dp[i-1][j-1];

}else if (pc[j-1] == '\*'){

if (j >= 2 && (pc[j-2] == sc[i-1] || pc[j-2] == '.')){

dp[i][j] = dp[i][j-2] || dp[i][j-1] || dp[i-1][j];

}else if (j >= 2){

dp[i][j] = dp[i][j-2];

}else{

dp[i][j] = dp[i][j-1] || dp[i-1][j];

}

}

}

}

return dp[m][n];

}

}

## 79\_LCS\_LongestCommonSubstring.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings, find the longest common substring.

Return the length of it.

Notice

The characters in substring should occur continuously in original string. This is different with subsequence.

Have you met this question in a real interview?

Example

Given A = "ABCD", B = "CBCE", return 2.

Challenge

O(n x m) time and memory.

https://www.jiuzhang.com/solution/longest-common-substring/

dp[i][j] is the length of longest common subarray ending with nums[i-1] and nums[j-1], which is different from previous

longest common subsequence

public class Solution {

/\*\*

\* @param A: A string

\* @param B: A string

\* @return: the length of the longest common substring.

\*/

public int longestCommonSubstring(String A, String B) {

int m = A.length();

int n = B.length();

int[][] dp = new int[m+1][n+1];

int max = 0;

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (A.charAt(i-1) == B.charAt(j-1)){

dp[i][j] = dp[i-1][j-1] + 1;

max = Math.max(max, dp[i][j]);

}

}

}

return max;

}

}

## 79\_WordSearch.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell,

where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example,

Given board =

[

['A','B','C','E'],

['S','F','C','S'],

['A','D','E','E']

]

word = "ABCCED", -> returns true,

word = "SEE", -> returns true,

word = "ABCB", -> returns false.

class Solution {

public boolean exist(char[][] board, String word) {

if (board == null || board.length == 0){

return false;

}

boolean[][] visited = new boolean[board.length][board[0].length];

for (int i = 0; i < board.length; i++){

for (int j = 0; j < board[0].length; j++){

if (dfs(board, word, i, j, 0, visited)){

return true;

}

}

}

return false;

}

private boolean dfs(char[][] board, String word, int i, int j, int start, boolean[][] visited){

if (start == word.length()){

return true;

}

if (i >= board.length || i < 0 || j >= board[0].length || j < 0){

return false;

}

if (visited[i][j]){

return false;

}

if (word.charAt(start) != board[i][j]){

return false;

}

visited[i][j] = true;

boolean left = dfs(board, word, i - 1, j, start + 1, visited);

boolean right = dfs(board, word, i + 1, j, start + 1, visited);

boolean top = dfs(board, word, i, j + 1, start + 1, visited);

boolean bottom = dfs(board, word, i, j - 1, start + 1, visited);

if (left || right || top || bottom){

return true;

}

visited[i][j] = false; //backtracking

return false;

}

}

Better:

class Solution {

public boolean exist(char[][] board, String word) {

if (board == null || board.length == 0){

return false;

}

int m = board.length;

int n = board[0].length;

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (board[i][j] == word.charAt(0)){

if (dfs(board, word, 0, i, j, visited)){

return true;

}

}

}

}

return false;

}

private boolean dfs(char[][] board, String word, int pos, int x, int y, boolean[][] visited){

if (board[x][y] != word.charAt(pos)){

return false;

}

if (pos == word.length() - 1){

return true;

}

int m = board.length;

int n = board[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

visited[x][y] = true;

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

if (dfs(board, word, pos + 1, nx, ny, visited)){

return true;

}

}

}

visited[x][y] = false;

return false;

}

}

Best solution:

class Solution {

public boolean exist(char[][] board, String word) {

if (board == null || board.length == 0){

return false;

}

int m = board.length;

int n = board[0].length;

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (board[i][j] == word.charAt(0)){

visited[i][j] = true;

if (dfs(board, word, i, j, visited, 0)){

return true;

}

visited[i][j] = false;

}

}

}

return false;

}

private boolean dfs(char[][] board, String word, int x, int y, boolean[][] visited, int start){

if (board[x][y] != word.charAt(start)){

return false;

}

if (start == word.length() - 1){

return true;

}

int m = board.length;

int n = board[0].length;

int[][] dirs= {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

visited[nx][ny] = true;

if (dfs(board, word, nx, ny, visited, start+1)){

return true;

}

visited[nx][ny] = false;

}

}

return false;

}

}

## 212\_WordSearchII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically

neighboring. The same letter cell may not be used more than once in a word.

For example,

Given words = ["oath","pea","eat","rain"] and board =

[

['o','a','a','n'],

['e','t','a','e'],

['i','h','k','r'],

['i','f','l','v']

]

Return ["eat","oath"].

Note:

You may assume that all inputs are consist of lowercase letters a-z.

click to show hint.

You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?

If the current candidate does not exist in all words' prefix, you could stop backtracking immediately.

What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie?

If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.

class Solution {

public List<String> findWords(char[][] board, String[] words) {

List<String> result = new ArrayList<>();

if (board == null || board.length == 0 || board[0].length == 0){

return result;

}

Trie trie = new Trie();

for (String word : words){

trie.insert(word);

}

int m = board.length;

int n = board[0].length;

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

dfs(board, result, trie, "", i, j, visited);

}

}

return result;

}

private void dfs(char[][] board, List<String> result, Trie trie, String str, int i, int j, boolean[][] visited){

int m = board.length;

int n = board[0].length;

if (i >= m || i < 0 || j >= n || j < 0){

return;

}

if (visited[i][j]){

return;

}

str += board[i][j];

if (!trie.startsWith(str)){

return;

}

if (trie.search(str)){

if (!result.contains(str)){

result.add(str);

}

}

visited[i][j] = true;

int[] dx = {1, 0 , -1 , 0};

int[] dy = {0, 1, 0, - 1};

for (int k = 0; k < 4; k++){

dfs(board, result, trie, str, i + dx[k], j + dy[k], visited);

}

visited[i][j] = false;

}

class Trie {

class TrieNode{

private TrieNode[] links;

private final int R = 26;

private boolean isEnd;

TrieNode(){

links = new TrieNode[26];

isEnd = false;

}

public boolean containsKey(char ch){

return links[ch - 'a'] != null;

}

public TrieNode get(char ch){

return links[ch - 'a'];

}

public void put(char ch, TrieNode node){

links[ch - 'a'] = node;

}

public void setEnd(){

isEnd = true;

}

public boolean getEnd(){

return isEnd;

}

}

private TrieNode root;

/\*\* Initialize your data structure here. \*/

public Trie() {

root = new TrieNode();

}

/\*\* Inserts a word into the trie. \*/

public void insert(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

if (!node.containsKey(word.charAt(i))){

node.put(word.charAt(i), new TrieNode());

}

node = node.get(word.charAt(i));

}

node.setEnd();

}

/\*\* Returns if the word is in the trie. \*/

public boolean search(String word) {

TrieNode node = searchPrefix(word);

return node != null && node.getEnd();

}

/\*\* Returns if there is any word in the trie that starts with the given prefix. \*/

public boolean startsWith(String prefix) {

TrieNode node = searchPrefix(prefix);

return node != null;

}

private TrieNode searchPrefix(String prefix){

TrieNode node = root;

for (int i = 0; i < prefix.length(); i++){

if (!node.containsKey(prefix.charAt(i))){

return null;

}

node = node.get(prefix.charAt(i));

}

return node;

}

}

}

Clean version:

class Solution {

class Trie {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode () {

children = new TrieNode[26];

isEnd = false;

}

}

TrieNode root;

public Trie () {

root = new TrieNode();

}

public void insert(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

public boolean startWith(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return true;

}

public boolean search(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return node.isEnd;

}

}

public List<String> findWords(char[][] board, String[] words) {

if (board == null || board.length == 0){

return new ArrayList<String>();

}

int m = board.length;

int n = board[0].length;

Set<String> res = new HashSet<>();

Trie trie = new Trie();

for (String word : words){

trie.insert(word);

}

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

dfs(board, trie, res, "", visited, i, j);

}

}

return new ArrayList<>(res);

}

private void dfs(char[][] board, Trie trie, Set<String> res, String word, boolean[][] visited, int x, int y){

word += board[x][y];

if (!trie.startWith(word)){

return;

}

if (trie.search(word)){

res.add(word);

}

int m = board.length;

int n = board[0].length;

int[][] dirs = {{1,0}, {0, 1}, {-1, 0}, {0, -1}};

visited[x][y] = true;

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

dfs(board, trie, res, word, visited, nx, ny);

}

}

visited[x][y] = false;

}

}

Better version:

class Solution {

class Trie {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode(){

children = new TrieNode[26];

isEnd = false;

}

}

TrieNode root;

public Trie (){

root = new TrieNode();

}

public void insert (String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

public boolean search (String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c- 'a'];

}

return node.isEnd;

}

public boolean startWith(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return true;

}

}

public List<String> findWords(char[][] board, String[] words) {

Set<String> res = new HashSet<>();

Trie trie = new Trie();

for (String str : words){

trie.insert(str);

}

int m = board.length;

int n = board[0].length;

boolean[][] visited = new boolean[m][n];

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

visited[i][j] = true;

backtrack(res, board, trie, visited, i, j, "" + board[i][j], dirs);

visited[i][j] = false;

}

}

return new ArrayList<String>(res);

}

private void backtrack(Set<String> res, char[][] board, Trie trie, boolean[][] visited, int x, int y, String item, int[][] dirs){

if (!trie.startWith(item)){

return;

}

if (trie.search(item)){

res.add(item);

}

int m = board.length;

int n = board[0].length;

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

visited[nx][ny] = true;

backtrack(res, board, trie, visited, nx, ny, item + board[nx][ny], dirs);

visited[nx][ny] = false;

}

}

}

}

class Solution {

class Trie{

class TrieNode{

TrieNode[] children;

boolean isEnd;

public TrieNode(){

children = new TrieNode[26];

isEnd = false;

}

}

TrieNode root;

public Trie(){

root = new TrieNode();

}

public void insert(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

public boolean startWith(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return true;

}

public boolean search(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return node.isEnd;

}

}

public List<String> findWords(char[][] board, String[] words) {

if (board == null || board.length == 0){

return new ArrayList<>();

}

Trie trie = new Trie();

for (String word : words){

trie.insert(word);

}

Set<String> res = new HashSet<>();

int m = board.length;

int n = board[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

visited[i][j] = true;

backtrack(board, trie, res, i, j, "", dirs, visited);

visited[i][j] = false;

}

}

return new ArrayList<>(res);

}

private void backtrack(char[][] board, Trie trie, Set<String> res, int x, int y, String item, int[][] dirs, boolean[][] visited){

item += board[x][y];

if (!trie.startWith(item)){

return;

}

if (trie.search(item)){

res.add(item);

}

int m = board.length;

int n = board[0].length;

for (int[] dir : dirs){

int nx = dir[0] + x;

int ny = dir[1] + y;

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

visited[nx][ny] = true;

backtrack(board, trie, res, nx, ny, item, dirs, visited);

visited[nx][ny] = false;

}

}

}

}

## 14\_LongestCommonPrefix.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

Example 1:

Input: ["flower","flow","flight"]

Output: "fl"

Example 2:

Input: ["dog","racecar","car"]

Output: ""

Explanation: There is no common prefix among the input strings.

class Solution {

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0){

return "";

}

String pre = strs[0];

for (int i = 1; i < strs.length; i++){

while (strs[i].indexOf(pre) != 0){

pre = pre.substring(0, pre.length() - 1);

}

}

return pre;

}

}

class Solution {

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0){

return "";

}

StringBuilder sb = new StringBuilder();

Boolean found = false;

for (int i = 0; i < strs[0].length(); i++){

char c = strs[0].charAt(i);

for (int j = 1; j < strs.length; j++){

if (i == strs[j].length() || c != strs[j].charAt(i)){

found = true;

break;

}

}

if (!found){

sb.append(c);

}else{

break;

}

}

return sb.toString();

}

}

Better: after sorting, only compare first and last

class Solution {

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0){

return "";

}

StringBuilder sb = new StringBuilder();

Arrays.sort(strs);

String first = strs[0];

String last = strs[strs.length - 1];

for (int i = 0; i < first.length(); i++){

if (first.charAt(i) != last.charAt(i)){

break;

}else{

sb.append(first.charAt(i));

}

}

return sb.toString();

}

}

## 151\_ReverseWordsinString.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an input string, reverse the string word by word.

For example,

Given s = "the sky is blue",

return "blue is sky the".

Update (2015-02-12):

For C programmers: Try to solve it in-place in O(1) space.

click to show clarification.

Clarification:

What constitutes a word?

A sequence of non-space characters constitutes a word.

Could the input string contain leading or trailing spaces?

Yes. However, your reversed string should not contain leading or trailing spaces.

How about multiple spaces between two words?

Reduce them to a single space in the reversed string.

Method:

public class Solution {

public String reverseWords(String s) {

String[] words = s.trim().split(" +"); // + is regular expression, means at least one space

Collections.reverse(Arrays.asList(words));

return String.join(" ", words);

}

}

public class Solution {

public String reverseWords(String s) {

if (s == null || s.trim().length() == 0){

return s.trim();

}

String[] arr = s.trim().split("\\s+");

int n = arr.length;

StringBuilder sb = new StringBuilder();

for (int i = 0; i < n; i++){

StringBuilder temp = new StringBuilder(arr[i]);

sb.append(temp.reverse());

sb.append(" ");

}

return sb.reverse().toString().trim();

}

}

## 186\_ReverseWordsinStringII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an input string, reverse the string word by word. A word is defined as a sequence of non-space characters.

The input string does not contain leading or trailing spaces and the words are always separated by a single space.

For example,

Given s = "the sky is blue",

return "blue is sky the".

Could you do it in-place without allocating extra space?

Related problem: Rotate Array

class Solution {

public void reverseWords(char[] str) {

if (str == null || str.length == 0){

return;

}

int start = 0;

int i = 0;

for (i = 0; i < str.length; i++){

if (str[i] == ' '){

reverse(str, start, i - 1);

start = i + 1;

}

}

reverse(str, start, i - 1);

reverse(str, 0, str.length - 1);

}

private void reverse(char[] str, int start, int end){

while (start < end){

char temp = str[start];

str[start] = str[end];

str[end] = temp;

start++;

end--;

}

}

}

## 271\_EncodeAndDecodeStrings.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

string encode(vector<string> strs) {

// ... your code

return encoded\_string;

}

Machine 2 (receiver) has the function:

vector<string> decode(string s) {

//... your code

return strs;

}

So Machine 1 does:

string encoded\_string = encode(strs);

and Machine 2 does:

vector<string> strs2 = decode(encoded\_string);

strs2 in Machine 2 should be the same as strs in Machine 1.

Implement the encode and decode methods.

Note:

The string may contain any possible characters out of 256 valid ascii characters. Your algorithm should be generalized enough to work on any possible characters.

Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.

Do not rely on any library method such as eval or serialize methods. You should implement your own encode/decode algorithm.

public class Codec {

// Encodes a list of strings to a single string.

public String encode(List<String> strs) {

StringBuilder sb = new StringBuilder();

for (String str : strs){

sb.append(String.valueOf(str.length()));

sb.append(":");

sb.append(str);

}

return sb.toString();

}

// Decodes a single string to a list of strings.

public List<String> decode(String s) {

List<String> result = new ArrayList<>();

int i = 0;

int numPt = 0;

while (i < s.length()){

if (s.charAt(i) == ':'){

int len = Integer.parseInt(s.substring(numPt, i));

String str = s.substring(i+1, i+1+len);

result.add(str);

numPt = i+1+len;

i = numPt;

}else{

i++;

}

}

return result;

}

}

// Your Codec object will be instantiated and called as such:

// Codec codec = new Codec();

// codec.decode(codec.encode(strs));

## 459\_RepeatedSubstringPattern.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty string check if it can be constructed by taking a substring of it and

appending multiple copies of the substring together. You may assume the given string consists

of lowercase English letters only and its length will not exceed 10000.

Example 1:

Input: "abab"

Output: True

Explanation: It's the substring "ab" twice.

Example 2:

Input: "aba"

Output: False

Example 3:

Input: "abcabcabcabc"

Output: True

Explanation: It's the substring "abc" four times. (And the substring "abcabc" twice.)

The length of the repeating substring must be a divisor of the length of the input string

Search for all possible divisor of str.length, starting for length/2

If i is a divisor of length, repeat the substring from 0 to i the number of times i is contained in s.length

If the repeated substring is equals to the input str return true

class Solution {

public boolean repeatedSubstringPattern(String str) {

int len = str.length();

for(int i=len/2 ; i>=1 ; i--) {

if(len%i == 0) {

int m = len/i;

String subS = str.substring(0,i);

int j;

for(j=1;j<m;j++) {

if(!subS.equals(str.substring(j\*i,i+j\*i)))

break;

}

if(j==m)

return true;

}

}

return false;

}

}

Time complexity: O(N^2)

class Solution {

public boolean repeatedSubstringPattern(String str) {

int len = str.length();

for(int i=len/2 ; i >= 1 ; i--) {

if(len%i == 0) {

int m = len/i;

StringBuilder sb = new StringBuilder();

String subS = str.substring(0,i);

for (int j = 0; j < m; j++){

sb.append(subS);

}

if (sb.toString().equals(str)){

return true;

}

}

}

return false;

}

}

## 557\_ReverseWordsInAStringIII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, you need to reverse the order of characters in each word within a sentence while

still preserving whitespace and initial word order.

Example 1:

Input: "Let's take LeetCode contest"

Output: "s'teL ekat edoCteeL tsetnoc"

Note: In the string, each word is separated by single space and there will not be any extra space in the string.

Method 1: general solution for multiple spaces, and/or leading trailing sapce between words

class Solution {

public String reverseWords(String s) {

if (s == null || s.length() == 0){

return s;

}

int start = 0;

int i = 0;

char[] charArray = s.toCharArray();

int n = charArra.length;

while (i < n){

while (i < n && charArray[i] == ' '){

i++;

}

start = i;

while (i <n && charArray[i] != ' '){

i++;

}

reverse(charArray, start, i-1);

}

return String.valueOf(charArray);

}

private void reverse(char[] charArray, int i, int j){

while (i < j){

char temp = charArray[i];

charArray[i] = charArray[j];

charArray[j] = temp;

i++;

j--;

}

}

}

Method 2: similar as reverse string II for one space between words

class Solution {

public String reverseWords(String s) {

if (s == null || s.length() == 0){

return s;

}

int start = 0;

int i = 0;

char[] charArray = s.toCharArray();

int n = charArra.length;

while (i <= n){

if (i == s.length() || charArray[i] == ' '){

reverse(charArray, start, i-1);

start = i + 1;

}

i++;

}

return String.valueOf(charArray);

}

private void reverse(char[] charArray, int i, int j){

while (i < j){

char temp = charArray[i];

charArray[i] = charArray[j];

charArray[j] = temp;

i++;

j--;

}

}

}

Method 3:

class Solution {

public String reverseWords(String s) {

if (s == null || s.length() == 0){

return s;

}

String[] arr = s.trim().split("\\s+");

StringBuilder sb = new StringBuilder();

for (int i = 0; i < arr.length; i++){

StringBuilder temp = new StringBuilder(arr[i]);

sb.append(temp.reverse().toString());

sb.append(" ");

}

return sb.toString().trim();

}

}

https://leetcode.com/articles/reverse-words-in-a-string/

# Stocks

## 121\_BestTimetoBuyandSellStock.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock),

design an algorithm to find the maximum profit.

Example 1:

Input: [7, 1, 5, 3, 6, 4]

Output: 5

max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to be larger than buying price)

Example 2:

Input: [7, 6, 4, 3, 1]

Output: 0

In this case, no transaction is done, i.e. max profit = 0.

Method 1: brute force

Time complexity: O(n^2)

class Solution {

public int maxProfit(int[] prices) {

if (prices == null || prices.length == 0){

return 0;

}

int max = Integer.MIN\_VALUE;

int[] f = new int[prices.length];

for (int i = 0; i < prices.length; i++){

for (int j = 0; j < i; j++){

f[i] = Math.max(f[i], prices[i] - prices[j]);

}

}

for (int i = 0; i < prices.length; i++){

max = Math.max(max, f[i]);

}

return max;

}

}

Method 2: dynamic programming

class Solution {

public int maxProfit(int[] prices) {

if (prices == null || prices.length == 0){

return 0;

}

int max = 0;

int min = Integer.MAX\_VALUE;

for (int i = 0; i < prices.length; i++){

min = Math.min(min, prices[i]);

max = Math.max(max, prices[i] - min);

}

return max;

}

}

Best solution: per generalization form

class Solution {

public int maxProfit(int[] prices) {

int minCost = Integer.MAX\_VALUE;

int maxProfits = 0;

for (int price : prices){

minCost = Math.min(minCost, price);//not price - maxProfits because only one transaction is allowed

maxProfits = Math.max(maxProfits, price - minCost);

}

return maxProfits;

}

}

Why DP?

https://leetcode.com/problems/best-time-to-buy-and-sell-stock/discuss/39112

## 122\_BestTimetoBuyandSellStockII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time

(ie, you must sell the stock before you buy again).

class Solution {

public int maxProfit(int[] prices) {

int max = 0;

for (int i = 1; i < prices.length; i++){

if (prices[i] > prices[i-1]){

max += prices[i] - prices[i-1];

}

}

return max;

}

}

Generalized form

class Solution {

public int maxProfit(int[] prices) {

int minCost = Integer.MAX\_VALUE;

int maxProfits = 0;

for (int price: prices){

minCost = Math.min(minCost, price - maxProfits);//not price because any number of transaction is allowed

maxProfits = Math.max(maxProfits, price - minCost);

}

return maxProfits;

}

}

## 123\_BestTimetoBuyandSellStockIII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

class Solution {

public int maxProfit(int[] prices) {

int minCost1 = Integer.MAX\_VALUE;

int maxProfit1 = 0;

int minCost2 = Integer.MAX\_VALUE;

int maxProfit2 = 0;

for (int i = 0; i < prices.length; i++){

minCost1 = Math.min(minCost1, prices[i]);

maxProfit1 = Math.max(maxProfit1, prices[i] - minCost1);

minCost2 = Math.min(minCost2, prices[i] - maxProfit1);

maxProfit2 = Math.max(maxProfit2, prices[i] - minCost2);

}

return maxProfit2;

}

}

General format:

class Solution {

public int maxProfit(int[] prices) {

int[] minCost = new int[3];

int[] maxprofits = new int[3];

for (int i = 0; i < 3; i++){

minCost[i] = Integer.MAX\_VALUE;

}

for (int i = 0; i < prices.length; i++){

for (int j = 1; j <= 2; j++){

minCost[j] = Math.min(minCost[j], prices[i] - maxprofits[j-1]);

maxprofits[j] = Math.max(maxprofits[j], prices[i] - minCost[j]);

}

}

return maxprofits[2];

}

}

## 188\_BestTimetoBuyandSellStockIV.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

class Solution {

public int maxProfit(int k, int[] prices) {

if (k >= prices.length / 2) { // if k >= n/2, then you can make maximum number of transactions

int profit = 0;

for (int i = 1; i < prices.length; i++)

if (prices[i] > prices[i - 1]) profit += prices[i] - prices[i - 1];

return profit;

}

int[] maxProfit = new int[k+1];

int[] minCost = new int[k+1];

for (int i = 0; i <= k ; i++){

minCost[i] = Integer.MAX\_VALUE;

}

for (int i = 0; i < prices.length; i++){

for (int j = 1; j <= k; j++){

minCost[j] = Math.min(minCost[j], prices[i] - maxProfit[j-1]);

maxProfit[j] = Math.max(maxProfit[j], prices[i] - minCost[j]);

}

}

return maxProfit[k];

}

}

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/discuss/54125/

## 309\_Best Time to Buy and Sell Stock with Cooldown.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like

(ie, buy one and sell one share of the stock multiple times) with the following restrictions:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

prices = [1, 2, 3, 0, 2]

maxProfit = 3

transactions = [buy, sell, cooldown, buy, sell]

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/discuss/75931/Easiest-JAVA-solution-with-explanations

buy[i]: Max profit till index i. The series of transaction is ending with a buy.

sell[i]: Max profit till index i. The series of transaction is ending with a sell.

Till index i, the buy / sell action must happen and must be the last action. It may not happen at index i.

It may happen at i - 1, i - 2, ... 0.

In the end n - 1, return sell[n - 1]. Apparently we cannot finally end up with a buy.

In that case, we would rather take a rest at n - 1.

For special case no transaction at all, classify it as sell[i], so that in the end, we can still return sell[n - 1].

Method 1: DP

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int maxProfit(int[] prices) {

if (prices.length < 2){

return 0;

}

int n = prices.length;

int[] buy = new int[n]; //maxProfit which transaction ends with buy

int[] sell = new int[n]; //maxProfit which transaction ends with sell

buy[0] = -prices[0];

sell[0] = 0;

buy[1] = Math.max(-prices[0], -prices[1]);

sell[1] = Math.max(sell[0], prices[1] - prices[0]);

for (int i = 2; i < n; i++){

buy[i] = Math.max(buy[i-1], sell[i-2] - prices[i]);

sell[i] = Math.max(sell[i-1], buy[i-1] + prices[i]);

}

return sell[n-1];

}

}

Method 2: DP

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public int maxProfit(int[] prices) {

if (prices.length < 2){

return 0;

}

int n = prices.length;

int b2 = -prices[0];

int s2 = 0;

int b1 = Math.max(-prices[0], -prices[1]);

int s1 = Math.max(0, prices[1] - prices[0]);

int buy = b1;

int sell = s1;

for (int i = 2; i < n; i++){

buy = Math.max(b1, s2 - prices[i]);

sell = Math.max(s1, b1 + prices[i]);

b2 = b1;

b1 = buy;

s2 = s1;

s1 = sell;

}

return sell;

}

}

# Palindrome

## 125\_ValidPalindrome.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,

"A man, a plan, a canal: Panama" is a palindrome.

"race a car" is not a palindrome.

Note:

Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

class Solution {

public boolean isPalindrome(String s) {

if (s == null || s.length() == 0){

return true;

}

int l = 0;

int r = s.length() - 1;

while (l < s.length() && r >= 0 && l < r){

while (l < s.length() && !Character.isLetterOrDigit(s.charAt(l))){

l++;

}

while (r >= 0 && !Character.isLetterOrDigit(s.charAt(r))){

r--;

}

if (l < r){

if(Character.toLowerCase(s.charAt(l)) != Character.toLowerCase(s.charAt(r))){

return false;

}

l++;

r--;

}

}

return true;

}

}

Better version:

class Solution {

public boolean isPalindrome(String s) {

if (s == null || s.length() == 0){

return true;

}

char[] chars = s.toCharArray();

int left = 0;

int right = s.length() - 1;

while (left < right){

while (left < right && !Character.isLetterOrDigit(chars[left])){//isLetterOrDigit: alphanumeric

left++;

}

while (left < right && !Character.isLetterOrDigit(chars[right])){

right--;

}

if (Character.toLowerCase(chars[left]) != Character.toLowerCase(chars[right])){//ignore cases

return false;

}

left++;

right--;

}

return true;

}

}

## 131\_PalindromePartitioning.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example, given s = "aab",

Return

[

["aa","b"],

["a","a","b"]

]

Method 1: DFS

class Solution {

public List<List<String>> partition(String s) {

List<List<String>> result = new ArrayList<>();

if (s == null || s.length() == 0){

return result;

}

List<String> item = new ArrayList<>();

dfs(result, item, s, 0);

return result;

}

private void dfs(List<List<String>> result, List<String> item, String s, int start){

if (start == s.length()){

result.add(new ArrayList<String>(item));

return;

}

for (int i = start; i < s.length(); i++){

String str = s.substring(start, i+1);

if (isValid(str)){

item.add(str);

dfs(result, item, s, i + 1);

item.remove(item.size() - 1);

}

}

}

private boolean isValid(String s){

int left = 0;

int right = s.length() - 1;

while (left < right){

if (s.charAt(left) != s.charAt(right)){

return false;

}

left++;

right--;

}

return true;

}

}

Method 2: Recursion + memo

Best solution:

note the difference with Leetcode 140 word break II

https://github.com/optimisea/Leetcode/blob/master/Java/140\_WordBreakII.java

In this case, there is always be able to partition

class Solution {

Map<String, List<List<String>>> map = new HashMap<>();

public List<List<String>> partition(String s) {

List<List<String>> res = new ArrayList<>();

if (s == null || s.length() == 0){

return res;

}

if (map.containsKey(s)){

return map.get(s);

}

for (int i = 1; i <= s.length(); i++){

String sub = s.substring(0, i);

if (isValid(sub)){

List<List<String>> temp = partition(s.substring(i));

if (temp.size() == 0){

List<String> list = new ArrayList<>();

list.add(sub);

res.add(list);

}else{ // if size == 0, it won't go into loop

for (List<String> item: temp){

List<String> list = new ArrayList<>();

list.add(sub);

list.addAll(item);

res.add(list);

}

}

}

}

map.put(s, res);

return res;

}

private boolean isValid(String s){

int left = 0;

int right = s.length() - 1;

while (left < right){

if (s.charAt(left) != s.charAt(right)){

return false;

}

left++;

right--;

}

return true;

}

}

Best solution:

class Solution {

public List<List<String>> partition(String s) {

Map<String, List<List<String>>> map = new HashMap<>();

return dfs(s, map);

}

private List<List<String>> dfs(String s, Map<String, List<List<String>>> map){

List<List<String>> res = new ArrayList<>();

if (s.length() == 0){

List<String> item = new ArrayList<>();

res.add(item);

return res;

}

if (map.containsKey(s)){

return map.get(s);

}

for (int i = 1; i <= s.length(); i++){

String sub = s.substring(0, i);

if (isValid(sub)){

List<List<String>> temp = dfs(s.substring(i), map);

for (List<String> list : temp){

List<String> item = new ArrayList<>();

item.add(sub);

item.addAll(list);

res.add(item);

}

}

}

map.put(s, res);

return res;

}

private boolean isValid(String s){

int left = 0;

int right = s.length() - 1;

while (left < right){

if (s.charAt(left) != s.charAt(right)){

return false;

}

left++;

right--;

}

return true;

}

}

## 132\_Palindrome.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

Example:

Input: "aab"

Output: 1

Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.

https://leetcode.com/problems/palindrome-partitioning-ii/discuss/42213/Easiest-Java-DP-Solution-(97.36)

Best solution:

class Solution {

public int minCut(String s) {

int n = s.length();

int[] dp = new int[n];

boolean[][] cache = new boolean[n][n];

for (int i = 0; i < n; i++){

dp[i] = i;

for (int j = 0; j <= i; j++){

if ((j+1 > i - 1 || cache[j+1][i-1]) && s.charAt(j) == s.charAt(i)){

cache[j][i] = true;

dp[i] = j == 0 ? 0 : Math.min(dp[i], dp[j-1] + 1);

}

}

}

return dp[n-1];

}

}

Similar as Longest Increasing Sequence

Time complexity: O(n^2)

Space complexity: O(n^2)

class Solution {

public int minCut(String s) {

int n = s.length();

int[] cut = new int[n];

boolean[][] dp = new boolean[n][n];

for (int i = 0; i < n; i++){

cut[i] = i;

for (int j = 0; j <= i; j++){

if ((j + 1 > i - 1 || dp[j+1][i-1]) && s.charAt(j) == s.charAt(i)){

dp[j][i] = true;

cut[i] = j == 0 ? 0 : Math.min(cut[i], cut[j-1] + 1);

}

}

}

return cut[n-1];

}

}

https://leetcode.com/problems/palindrome-partitioning-ii/discuss/42213/Easiest-Java-DP-Solution-(97.36)

This can be solved by two points:

cut[i] is the minimum of cut[j - 1] + 1 (j <= i), if [j, i] is palindrome.

If [j, i] is palindrome, [j + 1, i - 1] is palindrome, and c[j] == c[i].

The 2nd point reminds us of using dp (caching).

a b a | c c

j i

j-1 | [j, i] is palindrome

cut(j-1) + 1

Best solution:

class Solution {

public int minCut(String s) {

int n = s.length();

int[] dp = new int[n+1];

boolean[][] cache = new boolean[n][n];

for (int i = 1; i <= n; i++){

dp[i] = i-1;

for (int j = 0; j < i; j++){

if ((j + 1 >= i - 2 || cache[j+1][i-2]) && s.charAt(i-1) == s.charAt(j)){

cache[j][i-1] = true;

dp[i] = j == 0 ? 0 : Math.min(dp[i], dp[j] + 1);

}

}

}

return dp[n];

}

}

class Solution {

public int minCut(String s) {

int n = s.length();

int[] dp = new int[n];

boolean[][] cache = new boolean[n][n];

for (int i = 0; i < n; i++){

dp[i] = i;

for (int j = 0; j <= i; j++){

if ((j+1 > i - 1 || cache[j+1][i-1]) &&s.charAt(i) == s.charAt(j)){

cache[j][i] = true;

dp[i] = j == 0 ? 0 : Math.min(dp[i], dp[j-1] + 1);

}

}

}

return dp[n-1];

}

}

Method 2:

Time complexity: O(n^3)

Space complexity: O(n)

class Solution {

public int minCut(String s) {

int n = s.length();

int[] dp = new int[n];

for (int i = 0; i < n; i++){

dp[i] = i;

for (int j = 0; j <= i; j++){

if (isValid(s, j, i)){

dp[i] = j == 0 ? 0 : Math.min(dp[i], dp[j-1] + 1);

}

}

}

return dp[n-1];

}

private boolean isValid(String s, int j, int i){

while (j < i){

if (s.charAt(j) != s.charAt(i)){

return false;

}

j++;

i--;

}

return true;

}

}

class Solution {

public int minCut(String s) {

int n = s.length();

int[] dp = new int[n+1];

for (int i = 1; i <= n; i++){

dp[i] = i-1;

for (int j = 0; j < i; j++){

if (isValid(s, j, i-1)){

dp[i] = j == 0 ? 0 : Math.min(dp[i], dp[j] + 1);

}

}

}

return dp[n];

}

private boolean isValid(String s, int j, int i){

while (j < i){

if (s.charAt(j) != s.charAt(i)){

return false;

}

j++;

i--;

}

return true;

}

}

class Solution {

public int minCut(String s) {

int n = s.length();

int[] dp = new int[n];

for (int i = 0; i < n; i++){

dp[i] = i;

for (int j = 0; j <= i; j++){

if (isValid(s, j, i)){

dp[i] = j == 0 ? 0 : Math.min(dp[i], dp[j-1] + 1);

}

}

}

return dp[n-1];

}

private boolean isValid(String s, int j, int i){

while (j < i){

if (s.charAt(i) != s.charAt(j)){

return false;

}

j++;

i--;

}

return true;

}

}

## 266\_PalindromePermutation.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, determine if a permutation of the string could form a palindrome.

Example 1:

Input: "code"

Output: false

Example 2:

Input: "aab"

Output: true

Example 3:

Input: "carerac"

Output: true

Method 1:

class Solution {

public boolean canPermutePalindrome(String s) {

Map<Character, Integer> map = new HashMap<>();

for(int i = 0; i < s.length(); i++){

map.put(s.charAt(i), map.getOrDefault(s.charAt(i), 0) + 1);

}

int singleCount = 0;

for (char key : map.keySet()){

if (map.get(key) % 2 == 1){

singleCount++;

}

if (singleCount > 1){

return false;

}

}

return true;

}

}

Method 2:

class Solution {

public boolean canPermutePalindrome(String s) {

Set<Character> set = new HashSet<>();

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (!set.contains(c)){

set.add(c);

}else{

set.remove(c);

}

}

return set.size() <= 1;

}

}

## 267\_ PalindromePermutationII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list

if no palindromic permutation could be form.

Example 1:

Input: "aabb"

Output: ["abba", "baab"]

Example 2:

Input: "abc"

Output: []

Similar to permutation II to avoid duplication.

Backtracking for StringBuilder class

Basically, the idea is to perform permutation on half of the palindromic string and then form the full palindromic result.

class Solution {

public List<String> generatePalindromes(String s) {

List<String> result = new ArrayList<>();

Map<Character, Integer> map = new HashMap<>();

// step 1. build character count map and count odds

for(int i = 0; i < s.length(); i++){

map.put(s.charAt(i), map.getOrDefault(s.charAt(i), 0) + 1);

}

int singelCount = 0;

for (char ch : map.keySet()){

if (map.get(ch) % 2 == 1){

singelCount++;

}

}

if (singelCount > 1){

return result;

}

// step 2. add half count of each character to list

List<Character> list = new ArrayList<>();

String mid = "";

for (char key : map.keySet()){

int val = map.get(key);

if (val % 2 == 1){

mid += key;

}

for (int i = 0; i < val / 2; i++){

list.add(key);

}

}

// step 3. generate all the permutations

dfs(result, list, mid, new boolean[list.size()], new StringBuilder());

return result;

}

private void dfs(List<String> result, List<Character> list, String mid, boolean[] used, StringBuilder sb){

if (sb.length() == list.size()){

result.add(sb.toString() + mid + sb.reverse().toString());

sb.reverse();

return;

}

for (int i = 0; i < list.size(); i++){

if (i > 0 && list.get(i) == list.get(i-1) && !used[i-1]){

continue;

}

if (!used[i]){

used[i] = true;

sb.append(list.get(i));

dfs(result, list, mid, used, sb);

used[i] = false;

sb.deleteCharAt(sb.length() - 1);

}

}

}

}

## 336\_PalindromePairs.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of unique words, find all pairs of distinct indices (i, j) in the given list,

so that the concatenation of the two words, i.e. words[i] + words[j] is a palindrome.

Example 1:

Given words = ["bat", "tab", "cat"]

Return [[0, 1], [1, 0]]

The palindromes are ["battab", "tabbat"]

Example 2:

Given words = ["abcd", "dcba", "lls", "s", "sssll"]

Return [[0, 1], [1, 0], [3, 2], [2, 4]]

The palindromes are ["dcbaabcd", "abcddcba", "slls", "llssssll"]

https://leetcode.com/problems/palindrome-pairs/discuss/79210/The-Easy-to-unserstand-JAVA-Solution

There are several cases to be considered that isPalindrome(s1 + s2):

Case1: If s1 is a blank string, then for any string that is palindrome s2, s1+s2 and s2+s1 are palindrome.

Case 2: If s2 is the reversing string of s1, then s1+s2 and s2+s1 are palindrome.

Case 3: If s1[0:cut] is palindrome and there exists s2 is the reversing string of s1[cut+1:] , then s2+s1 is palindrome.

Case 4: Similiar to case3. If s1[cut+1: ] is palindrome and there exists s2 is the reversing string of s1[0:cut] ,

then s1+s2 is palindrome.

To make the search faster, build a HashMap to store the String-idx pairs.

class Solution {

public List<List<Integer>> palindromePairs(String[] words) {

List<List<Integer>> result = new ArrayList<>();

if (words == null || words.length == 0){

return result;

}

Map<String, Integer> map = new HashMap<>();

for (int i = 0; i < words.length; i++){

map.put(words[i], i);

}

if (map.containsKey("")){

int blankInd = map.get("");

for (int i = 0; i < words.length; i++){

if (i == blankInd){

continue;

}

if (isPalindrome(words[i])){

result.add(Arrays.asList(blankInd, i));

result.add(Arrays.asList(i, blankInd));

}

}

}

for(int i = 0; i < words.length; i++){

if (map.containsKey(reverse(words[i]))){

int found = map.get(reverse(words[i]));

if (found == i){

continue;

}

result.add(Arrays.asList(i, found));

}

}

for (int i = 0; i < words.length; i++){

for (int j = 1; j < words[i].length(); j++){

String sub1 = words[i].substring(0, j);

String sub2 = words[i].substring(j);

if (isPalindrome(sub1)){

String reverseSub2 = reverse(sub2);

if (map.containsKey(reverseSub2)){

result.add(Arrays.asList(map.get(reverseSub2), i));

}

}

if (isPalindrome(sub2)){

String reverseSub1 = reverse(sub1);

if (map.containsKey(reverseSub1)){

result.add(Arrays.asList(i, map.get(reverseSub1)));

}

}

}

}

return result;

}

private String reverse(String str){

StringBuilder sb = new StringBuilder(str);

return sb.reverse().toString();

}

private boolean isPalindrome(String str){

int start = 0;

int end = str.length() - 1;

while (start < end){

if (str.charAt(start) != str.charAt(end)){

return false;

}

start++;

end--;

}

return true;

}

}

# Parentheses

## 20\_ValidParentheses.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

Method 1: Use ASCII table

class Solution {

public boolean isValid(String s) {

if (s == null || s.length() == 0){

return false;

}

Stack<Character> stack = new Stack<>();

for (int i = 0; i < s.length(); i++){

if (stack.isEmpty()){

stack.push(s.charAt(i));

}else if (s.charAt(i) - stack.peek() == 1 || s.charAt(i) - stack.peek() == 2){

stack.pop();

}else{

stack.push(s.charAt(i));

}

}

return stack.isEmpty();

}

}

Method 2:

public boolean isValid(String s) {

if (s == null || s.length() == 0){

return false;

}

Stack<Character> stack = new Stack<Character>();

for (char c : s.toCharArray()) {

if (c == '(')

stack.push(')');

else if (c == '{')

stack.push('}');

else if (c == '[')

stack.push(']');

else if (stack.isEmpty() || stack.pop() != c)

return false;

}

return stack.isEmpty();

}

## 32\_LongestValidParenthesesSubstring.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed)

parentheses substring.

Example 1:

Input: "(()"

Output: 2

Explanation: The longest valid parentheses substring is "()"

Example 2:

Input: ")()())"

Output: 4

Explanation: The longest valid parentheses substring is "()()"

Method 1: TLE

Time complexity: O(n^3)

Space complexity: O(n)

class Solution {

public int longestValidParentheses(String s) {

int ans = 0;

for (int i = 0; i < s.length(); i++){

for (int j = i+2; j <= s.length(); j += 2){

if (isValid(s.substring(i, j))){

ans = Math.max(ans, j - i);

}

}

}

return ans;

}

public boolean isValid(String s) {

Stack<Character> stack = new Stack<Character>();

for (int i = 0; i < s.length(); i++) {

if (s.charAt(i) == '(') {

stack.push(')');

} else if (stack.empty() || stack.pop() != s.charAt(i)) {

return false;

}

}

return stack.empty();

}

}

Method 2:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int longestValidParentheses(String s) {

if (s == null || s.length() == 0){

return 0;

}

int n = s.length();

int max = 0;

int[] dp = new int[n]; // denotes the longest valid parentheses ending at index i (but include index i)

for (int i = 1; i < n; i++){

if (s.charAt(i) == ')'){

if (s.charAt(i-1) == '('){

dp[i] = (i - 2 >= 0 ? dp[i-2] : 0) + 2;

}else if (i - dp[i-1] - 1 >= 0 && s.charAt(i-dp[i-1]-1) == '('){

dp[i] = (i-dp[i-1]-2 >= 0 ? dp[i-dp[i-1]-2]: 0) + dp[i-1] + 2;

}

}

max = Math.max(max, dp[i]);

}

return max;

}

}

Method 3:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int longestValidParentheses(String s) {

Stack<Integer> stack = new Stack<>();

stack.push(-1);

int ans = 0;

for (int i = 0; i < s.length(); i++){

if (s.charAt(i) == '('){

stack.push(i);

}else{

stack.pop();

if (stack.isEmpty()){

stack.push(i);

}

ans = Math.max(ans, i - stack.peek());

}

}

return ans;

}

}

Method 4: Best solution

Time complexity: O(n)

Space complexity: O(1)

must scan from left to right and from right to left.

e.g., (()

class Solution {

public int longestValidParentheses(String s) {

int left = 0, right = 0, ans = 0;

for (int i = 0; i < s.length(); i++){

if (s.charAt(i) == '('){

left++;

}else {

right++;

}

if (left == right){

ans = Math.max(ans, left + right);

}else if (left < right){

left = 0;

right = 0;

}

}

left = right = 0;

for (int i = s.length() - 1; i >= 0; i--){

if (s.charAt(i) == '('){

left++;

}else{

right++;

}

if (left == right){

ans = Math.max(ans, left + right);

}else if (left > right){

left = 0;

right = 0;

}

}

return ans;

}

}

## 301\_RemoveInvalidParentheses.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses ( and ).

Examples:

"()())()" -> ["()()()", "(())()"]

"(a)())()" -> ["(a)()()", "(a())()"]

")(" -> [""]

https://leetcode.com/problems/remove-invalid-parentheses/discuss/75032/Share-my-Java-BFS-solution

Time complexity: O(n\* 2^n)

class Solution {

public List<String> removeInvalidParentheses(String s) {

List<String> res = new ArrayList<>();

if (s == null){

return res;

}

Queue<String> queue = new LinkedList<>();

Set<String> visited = new HashSet<>();

queue.offer(s);

visited.add(s);

boolean found = false;

while(!queue.isEmpty() && !found){

int size = queue.size();

for (int j = 0; j < size; j++){

String curr = queue.poll();

if (isValid(curr)){

res.add(curr);

found = true;

}

if (!found){

for (int i = 0; i < curr.length(); i++){

char c = curr.charAt(i);

if (Character.isLetter(c)){

continue;

}

String str = curr.substring(0, i) + curr.substring(i+1);

if (!visited.contains(str)){

queue.offer(str);

visited.add(str);

}

}

}

}

}

return res;

}

private boolean isValid(String str){

int count = 0;

for (char c : str.toCharArray()){

if (c == '('){

count++;

}else if (c == ')'){

count--;

if (count < 0){

return false;

}

}

}

return count == 0;

}

}

Another way to check is valid parenthesis without stack

private boolean isValid(String str){

int count = 0;

for (char c : str.toCharArray()){

if (c == '('){

count++;

}else if (c == ')'){

count--;

if (count < 0){

return false;

}

}

}

count = 0;

for (int i = str.length() - 1; i >= 0; i--){

char c = str.charAt(i);

if (c == ')'){

count++;

}else if (c == '('){

count--;

if (count <0){

return false;

}

}

}

return true;

}

Check Leetcode 921: minimum add to make parenthesis valid

https://github.com/optimisea/Leetcode/blob/master/Java/921\_Minimum.java

## 423\_ValidParentheses.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

Have you met this question in a real interview? Yes

Example

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

public class Solution {

/\*

\* @param s: A string

\* @return: whether the string is a valid parentheses

\*/

public boolean isValidParentheses(String s) {

Stack<Character> stack = new Stack<>();

HashMap<Character, Character> map = new HashMap<>();

map.put(']', '[');

map.put('}', '{');

map.put(')', '(');

char[] sc = s.toCharArray();

for (int i = 0; i < sc.length; i++){

if (sc[i] == '(' || sc[i] == '[' || sc[i] == '{'){

stack.push(sc[i]);

}else{

if (stack.isEmpty() || map.get(sc[i]) != stack.pop()){

return false;

}

}

}

return stack.isEmpty();

}

}

class Solution {

public boolean isValid(String s) {

Stack<Character> stack = new Stack<>();

for (char c : s.toCharArray()){

if (c == '('){

stack.push(')');

}else if (c == '['){

stack.push(']');

}else if (c == '{'){

stack.push('}');

}else if (stack.isEmpty() || c != stack.pop()){

return false;

}

}

return stack.isEmpty();

}

}

## 1021\_RemoveOutermostParentheses.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A valid parentheses string is either empty (""), "(" + A + ")", or A + B, where A and B are valid parentheses strings, and + represents string concatenation. For example, "", "()", "(())()", and "(()(()))" are all valid parentheses strings.

A valid parentheses string S is primitive if it is nonempty, and there does not exist a way to split it into S = A+B, with A and B nonempty valid parentheses strings.

Given a valid parentheses string S, consider its primitive decomposition: S = P\_1 + P\_2 + ... + P\_k, where P\_i are primitive valid parentheses strings.

Return S after removing the outermost parentheses of every primitive string in the primitive decomposition of S.

Example 1:

Input: "(()())(())"

Output: "()()()"

Explanation:

The input string is "(()())(())", with primitive decomposition "(()())" + "(())".

After removing outer parentheses of each part, this is "()()" + "()" = "()()()".

Example 2:

Input: "(()())(())(()(()))"

Output: "()()()()(())"

Explanation:

The input string is "(()())(())(()(()))", with primitive decomposition "(()())" + "(())" + "(()(()))".

After removing outer parentheses of each part, this is "()()" + "()" + "()(())" = "()()()()(())".

Example 3:

Input: "()()"

Output: ""

Explanation:

The input string is "()()", with primitive decomposition "()" + "()".

After removing outer parentheses of each part, this is "" + "" = "".

Note:

S.length <= 10000

S[i] is "(" or ")"

S is a valid parentheses string

class Solution {

public String removeOuterParentheses(String S) {

StringBuilder sb = new StringBuilder();

int count = 0;

int start = 0;

for (int i = 0; i < S.length(); i++){

char c = S.charAt(i);

if (c == '('){

count++;

}else{

count--;

}

if (count == 0){

sb.append(S.substring(start+1, i));

start = i+1;

}

}

return sb.toString();

}

}

# Trie

## [208\_Trie.java](#_208_Trie.java)

# Longest

## 76\_LIS\_LongestIncreasingSubsequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sequence of integers, find the longest increasing subsequence (LIS).

You code should return the length of the LIS.

Have you met this question in a real interview? Yes

Clarification

What's the definition of longest increasing subsequence?

The longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous, or unique.

https://en.wikipedia.org/wiki/Longest\_increasing\_subsequence

Example

For [5, 4, 1, 2, 3], the LIS is [1, 2, 3], return 3

For [4, 2, 4, 5, 3, 7], the LIS is [2, 4, 5, 7], return 4

public class Solution {

/\*

\* @param nums: An integer array

\* @return: The length of LIS (longest increasing subsequence)

\*/

public int longestIncreasingSubsequence(int[] nums) {

int[] f = new int[nums.length];

for (int i = 0; i < nums.length; i++){

f[i] = 1;

}

for (int i = 0; i < nums.length; i++){

for (int j = 0; j < i; j++){

if (nums[j] < nums[i]){

f[i] = Math.max(f[i], f[j] + 1);

}

}

}

int result = 0;

for (int i = 0; i < nums.length; i++){

result = Math.max(result, f[i]);

}

return result;

}

}

Best solution:

class Solution {

public int lengthOfLIS(int[] nums) {

int[] dp = new int[nums.length];

int max = 1;

for (int i = 0; i < nums.length; i++){

dp[i] = 1;

for (int j = 0; j < i; j++){

if (nums[j] < nums[i]){

dp[i] = Math.max(dp[i], dp[j] + 1);

}

}

max = Math.max(max, dp[i]);

}

return max;

}

}

## 76\_MinimumWindowSubstring.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example,

S = "ADOBECODEBANC"

T = "ABC"

Minimum window is "BANC".

Note:

If there is no such window in S that covers all characters in T, return the empty string "".

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

class Solution {

public String minWindow(String s, String t) {

if (s == null || s.length() == 0){

return "";

}

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i < t.length(); i++){

map.put(t.charAt(i), map.getOrDefault(t.charAt(i), 0) + 1);

}

int count = map.size();

int start = 0;

int end = 0;

int len = Integer.MAX\_VALUE;

int head = 0;

while (end < s.length()){

char c = s.charAt(end);

if (map.containsKey(c)){

map.put(c, map.get(c) - 1);

if (map.get(c) == 0){

count--;

}

}

end++;

while (count == 0){

if (end - start < len){

len = end - start;

head = start;

}

char cstart = s.charAt(start);

if (map.containsKey(cstart)){

if (map.get(cstart) == 0){

count++;

}

map.put(cstart, map.get(cstart) + 1);

}

start++;

}

}

if (len == Integer.MAX\_VALUE){

return "";

}

return s.substring(head, head + len);

}

}

Better solution to match the sliding window template

class Solution {

public String minWindow(String s, String t) {

Map<Character, Integer> map = new HashMap<>();

for (char c : t.toCharArray()){

map.put(c, map.getOrDefault(c, 0) + 1);

}

int count = map.size();

int start = 0;

int end = 0;

int head = 0;

int minLen = Integer.MAX\_VALUE;

while (end < s.length()){

char cEnd = s.charAt(end);

if (map.containsKey(cEnd)){

map.put(cEnd, map.get(cEnd) - 1);

if (map.get(cEnd) == 0){

count--;

}

}

end++;

while (count == 0){

if (end - start < minLen){

minLen = end - start;

head = start;

}

char cStart = s.charAt(start);

if (map.containsKey(cStart)){

if (map.get(cStart) == 0){

count++;

}

map.put(cStart, map.get(cStart) + 1);

}

start++;

}

}

return minLen == Integer.MAX\_VALUE ? "" : s.substring(head, head + minLen);

}

}

## 124\_ LongestSequenceOfRepetativeChars.java

public class LSS {

public static void main(String args[]) {

String myString = "aaaabbbbcccchhhhiiiiibbbbbbbbbccccccc";

LongestSequenceOfRepetativeChars(myString);

}

private static void LongestSequenceOfRepetativeChars(String s) {

int counter = 1;

char cur = '\0';

char next = '\0';

int max = 0;

char res = '\0';

for (int i = 0; i < s.length() - 1; i++) {

cur = s.charAt(i);

next = s.charAt(i + 1);

// If character's are in sequence , increase the counter

if (cur == next) {

counter++;

} else {

// When sequence is completed, check if it is longest

if (counter > max) {

max = counter;

res = cur;

}

counter = 1; // re-initialize counter

}

}

// Check if last string sequence is longest

if (counter > max) {

max = counter;

res = cur;

}

var repeated = new String(new char[max]).replace('\0', res);

System.out.println(repeated);

System.out.println("Longest character sequence is of character "

+ res + " and is " + max + " long");

}

}

## 124\_LongestConsecutiveSequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

Have you met this question in a real interview? Yes

Clarification

Your algorithm should run in O(n) complexity.

Example

Given [100, 4, 200, 1, 3, 2],

The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

public class Solution {

/\*

\* @param num: A list of integers

\* @return: An integer

\*/

public int longestConsecutive(int[] num) {

if (num.length == 0){

return 0;

}

Set<Integer> set = new HashSet<>();

for (Integer item : num){

set.add(item);

}

int ans = 0;

for (Integer item : num){

if (set.contains(item)){

set.remove(item);

}

int up = item + 1;

int down = item - 1;

while (set.contains(up)){

set.remove(up);

up++;

}

while (set.contains(down)){

set.remove(down);

down--;

}

ans = Math.max(ans, up - down - 1);

}

return ans;

}

}

Best solution:

class Solution {

public int longestConsecutive(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

Set<Integer> set = new HashSet<>();

for (int num : nums){

set.add(num);

}

int res = 1;

for (int num : nums){

int up = 1;

while (set.contains(num+up)){

set.remove(num+up);

up++;

}

int down = 1;

while (set.contains(num-down)){

set.remove(num-down);

down++;

}

set.remove(num);

res = Math.max(res, up + down - 1);

}

return res;

}

}

## 128\_LongestConsecutiveSequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,

Given [100, 4, 200, 1, 3, 2],

The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in O(n) complexity.

class Solution {

public int longestConsecutive(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

Set<Integer> set = new HashSet<>();

int ans = 0;

for (int i = 0; i < nums.length; i++){

if (!set.contains(nums[i])){

set.add(nums[i]);

}

}

for (int i = 0; i < nums.length; i++){

int up = nums[i] + 1;

while (set.contains(up)){

set.remove(up);

up++;

}

int down = nums[i] - 1;

while (set.contains(down)){

set.remove(down);

down--;

}

ans = Math.max(ans, up - down - 1);

}

return ans;

}

}

## 159\_LongestSubstringWithAtMostTwoDistinctCharacters.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s , find the length of the longest substring t that contains at most 2 distinct characters.

Example 1:

Input: "eceba"

Output: 3

Explanation: t is "ece" which its length is 3.

Example 2:

Input: "ccaabbb"

Output: 5

Explanation: t is "aabbb" which its length is 5.

Sliding window

class Solution {

public int lengthOfLongestSubstringTwoDistinct(String s) {

if (s == null){

return 0;

}

Map<Character, Integer> map = new HashMap<>();

int start = 0;

int end = 0;

int max = 0;

int count = 0;

while (end < s.length()){

char charEnd = s.charAt(end);

map.put(charEnd, map.getOrDefault(charEnd, 0) + 1);

if (map.get(charEnd) == 1){

count++;

}

while (count > 2){

if (max < end - start){

max = end - start;

}

char charStart = s.charAt(start);

if (map.get(charStart) == 1){

count--;

}

map.put(charStart, map.get(charStart) - 1);

start++;

}

end++;

}

return Math.max(max, end - start);

}

}

Better solution to match longest sliding window template

class Solution {

public int lengthOfLongestSubstringTwoDistinct(String s) {

if (s == null){

return 0;

}

Map<Character, Integer> map = new HashMap<>();

int start = 0;

int end = 0;

int max = 0;

int count = 0;

while (end < s.length()){

char charEnd = s.charAt(end);

map.put(charEnd, map.getOrDefault(charEnd, 0) + 1);

if (map.get(charEnd) == 1){

count++;

}

end++;

while (count > 2){

char charStart = s.charAt(start);

if (map.get(charStart) == 1){

count--;

}

map.put(charStart, map.get(charStart) - 1);

start++;

}

max = Math.max(max, end - start);

}

return max;

}

}

## 300\_LIS\_LengthOfLongestIncreasingSubsequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted array of integers, find the length of longest increasing subsequence.

For example,

Given [10, 9, 2, 5, 3, 7, 101, 18],

The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4.

Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in O(n2) complexity.

Follow up: Could you improve it to O(n log n) time complexity?

Method 1:

Time complexity: O(n2)

class Solution {

public int lengthOfLIS(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int[] f = new int[nums.length];

for (int i = 0; i < nums.length; i++){

f[i] = 1;

}

for (int i = 1; i < nums.length; i++){

for (int j = 0; j < i; j++){

if (nums[j] < nums[i]){

f[i] = Math.max(f[i], f[j] + 1);

}

}

}

int ans = 1;

for (int i = 0; i < nums.length; i++){

ans = Math.max(ans, f[i]);

}

return ans;

}

}

class Solution {

public int lengthOfLIS(int[] nums) {

int[] dp = new int[nums.length];

for (int i = 0; i < nums.length; i++){

dp[i] = 1;

for (int j = 0; j < i; j++){

if (nums[j] < nums[i]){

dp[i] = Math.max(dp[i], dp[j] + 1);

}

}

}

int max = 0;

for (int i : dp){

max = Math.max(max, i);

}

return max;

}

}

Method 2:

https://discuss.leetcode.com/topic/39681/fast-java-binary-search-solution-with-detailed-explanation/4

https://leetcode.com/problems/longest-increasing-subsequence/discuss/74824/JavaPython-Binary-search-O(nlogn)-time-with-explanation

Note that the length is correct but the sequence may be wrong,

because even it did the substitution, it didn't change the LIS' length. so even some of the number is added incorrectly,

it won't change the fact that there existed such a sequence with this length.

so whenever a new digit added, there are two cases. one is that it been added to somewhere in the middle,

which won't change the last digit. the last digit matters if we will add a new one or not.

Second case is it been added to the last, which indicate it will also works for the true LIS.

So the list DP maybe wrong, the length is right

Here's another example:

[1, 6, 8, 9 ,2, 3]

The sequence will be like:

1;

1, 6;

1, 6, 8;

1, 6, 8, 9;

1, 2, 8, 9;

1, 2, 3, 9;

Obviously, 1,2 ,3, 9 is not the correct sequence, but the length value is correct.

Time complexity: O(nlogn)

class Solution {

public int lengthOfLIS(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int[] dp = new int[nums.length];

int end = 0;

dp[0] = nums[0];

for (int i = 1; i < nums.length; i++){

int index = binarySearch(dp, 0, end, nums[i]);

if (nums[i] < dp[index]){

dp[index] = nums[i];

}

if (index > end){

dp[index] = nums[i];

end++;

}

}

return end+1;

}

private int binarySearch(int[] A, int start, int end, int target){

int length = end - start + 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (A[mid] == target){

return mid;

}else if (A[mid] < target){

start = mid;

}else{

end = mid;

}

}

if (A[end] < target){

return length;

}else if (A[start] > target){

return 0;

}else if (A[start] == target){

return start;

}

return end;

}

}

https://leetcode.com/problems/longest-increasing-subsequence/discuss/74825/Short-Java-solution-using-DP-O(n-log-n)

class Solution {

public int lengthOfLIS(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int[] dp = new int[nums.length];

int len = 0;

for (int i = 0; i < nums.length; i++){

int index = Arrays.binarySearch(dp, 0, len, nums[i]);

if (index < 0){

index = - (index + 1);

}

dp[index] = nums[i];

if (index == len){

len++;

}

}

return len;

}

}

Best solution:

Time complexity: O(nlogn)

https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/

Key idea: The end element of smaller list is smaller than end elements of larger lists

Three cases:

1. If A[i] is smallest among all end

candidates of active lists, we will start

new active list of length 1.

2. If A[i] is largest among all end candidates of

active lists, we will clone the largest active

list, and extend it by A[i].

3. If A[i] is in between, we will find a list with

largest end element that is smaller than A[i].

Clone and extend this list by A[i]. We will discard all

other lists of same length as that of this modified list.

Algorithm:

Querying length of longest is fairly easy. Note that we are dealing with end elements only. We need not to maintain all the lists.

We can store the end elements in an array. Discarding operation can be simulated with replacement, and extending a list is analogous

to adding more elements to array.

We will use an auxiliary array to keep end elements. The maximum length of this array is that of input. In the worst case the array

divided into N lists of size one (note that it does’t lead to worst case complexity). To discard an element, we will trace ceil value

of A[i] in auxiliary array (again observe the end elements in your rough work), and replace ceil value with A[i]. We extend a list by

adding element to auxiliary array. We also maintain a counter to keep track of auxiliary array length.

class Solution {

public int lengthOfLIS(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int n = nums.length;

int[] tail = new int[n];

tail[0] = nums[0];

int len = 1;

for (int i = 1; i < n; i++){

if (nums[i] < tail[0]){

tail[0] = nums[i];

}else if (nums[i] > tail[len-1]){

tail[len++] = nums[i];

}else{

int index = binarySearch(tail, 0, len - 1, nums[i]); // find the index of the first that is greater than target

tail[index] = nums[i];

}

}

return len;

}

private int binarySearch(int[] nums, int start, int end, int target){

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] >= target){

end = mid;

}else{

start = mid;

}

}

if (nums[start] >= target){

return start;

}

return end;

}

//the following binary search works too

private int binarySearch(int[] nums, int start, int end, int target){

while (start <= end){

int mid = start + (end - start) / 2;

if (nums[mid] >= target){

end = mid-1;

}else{

start = mid+1;

}

}

return start;

}

}

## 340\_LongestSubstringWithAtMostKDistinctCharacters.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, find the length of the longest substring T that contains at most k distinct characters.

For example, Given s = “eceba” and k = 2,

T is "ece" which its length is 3.

class Solution {

public int lengthOfLongestSubstringKDistinct(String s, int k) {

if (s == null || s.length() == 0 || k == 0){

return 0;

}

Map<Character, Integer> map = new HashMap<>();

int start = 0;

int end = 0;

int count = 0; // track how many different characters now

int max = Integer.MIN\_VALUE;

while (end < s.length()){

char c = s.charAt(end);

map.put(c, map.getOrDefault(c, 0) + 1);

if (map.get(c) == 1){

count++;

}

end++;

while (count > k){

char cs = s.charAt(start);

map.put(cs, map.get(cs) - 1);

if (map.get(cs) == 0){

count--;

}

start++;

}

max = Math.max(max, end - start);

}

return max;

}

}

## 395\_LongestSubstringWithAtLeastKRepeatingCharacters.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the length of the longest substring T of a given string (consists of lowercase letters only)

such that every character in T appears no less than k times.

Example 1:

Input:

s = "aaabb", k = 3

Output:

3

The longest substring is "aaa", as 'a' is repeated 3 times.

Example 2:

Input:

s = "ababbc", k = 2

Output:

5

The longest substring is "ababb", as 'a' is repeated 2 times and 'b' is repeated 3 times.

Method 1: O(N^2)

class Solution {

public int longestSubstring(String s, int k) {

if (s == null || s.length() == 0){

return 0;

}

if (k <= 0){

return s.length();

}

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i < s.length(); i++){

map.put(s.charAt(i), map.getOrDefault(s.charAt(i), 0) + 1);

}

char badchar = '\0';

for (Character c : map.keySet()){

if (map.get(c) < k){

badchar = c;

break;

}

}

if (badchar == '\0'){

return s.length();

}

String[] strArray = s.split(String.valueOf(badchar));

int max = 0;

for (String sub : strArray){

max = Math.max(max, longestSubstring(sub, k));

}

return max;

}

}

Method 2: sliding window, based on Leetcode 340 template

Time complexity: O(26\*N)

class Solution {

public int longestSubstring(String s, int k) {

if (s == null || s.length() == 0){

return 0;

}

int max = 0;

for (int numTargetDistinct = 1; numTargetDistinct <= 26; numTargetDistinct++){

int len = longestDistinct(s, k, numTargetDistinct);

max = Math.max(max, len);

}

return max;

}

private int longestDistinct(String s, int k, int numTargetDistinct){

Map<Character, Integer> map = new HashMap<>();

int start = 0;

int end = 0;

int uniqueNum = 0;

int noLessThanKNum = 0;

int max = 0;

while (end < s.length()){

char cEnd = s.charAt(end);

map.put(cEnd, map.getOrDefault(cEnd, 0) + 1);

if (map.get(cEnd) == 1){

uniqueNum++;

}

if (map.get(cEnd) == k){

noLessThanKNum++;

}

end++;

while (uniqueNum > numTargetDistinct){

char cStart = s.charAt(start);

if (map.get(cStart) == k){

noLessThanKNum--;

}

if (map.get(cStart) == 1){

uniqueNum--;

}

map.put(cStart, map.get(cStart) - 1);

start++;

}

if (uniqueNum == noLessThanKNum && uniqueNum == numTargetDistinct){

max = Math.max(max, end - start);

}

}

return max;

}

}

Best solution:

class Solution {

public int longestSubstring(String s, int k) {

if (s == null || s.length() == 0){

return 0;

}

int max = 0;

for (int numTargetDistinct = 1; numTargetDistinct <= 26; numTargetDistinct++){

int len = longestDistinct(s, k, numTargetDistinct);

max = Math.max(max, len);

}

return max;

}

private int longestDistinct(String s, int k, int numTargetDistinct){

Map<Character, Integer> map = new HashMap<>();

int start = 0;

int end = 0;

int uniqueNum = 0;

int noLessThanKNum = 0;

int max = 0;

while (end < s.length()){

char cEnd = s.charAt(end);

map.put(cEnd, map.getOrDefault(cEnd, 0) + 1);

if (map.get(cEnd) == 1){

uniqueNum++;

}

if (map.get(cEnd) == k){

noLessThanKNum++;

}

end++;

while (uniqueNum > numTargetDistinct){

char cStart = s.charAt(start);

if (map.get(cStart) == k){

noLessThanKNum--;

}

if (map.get(cStart) == 1){

uniqueNum--;

}

map.put(cStart, map.get(cStart) - 1);

start++;

}

if (noLessThanKNum == numTargetDistinct){//make sure every char is no less than k

max = Math.max(max, end - start);

}

}

return max;

}

}

## 397\_LICS\_LongestContinuousIncreasingSubsequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Give an integer array，find the longest increasing continuous subsequence in this array.

An increasing continuous subsequence:

Can be from right to left or from left to right.

Indices of the integers in the subsequence should be continuous.

Notice

O(n) time and O(1) extra space.

Have you met this question in a real interview?

Example

For [5, 4, 2, 1, 3], the LICS is [5, 4, 2, 1], return 4.

For [5, 1, 2, 3, 4], the LICS is [1, 2, 3, 4], return 4.

public class Solution {

/\*

\* @param A: An array of Integer

\* @return: an integer

\*/

public int longestIncreasingContinuousSubsequence(int[] A) {

if (A == null || A.length == 0){

return 0;

}

int ans = 1;

int length = 1;

for (int i = 1; i < A.length; i++){

if (A[i] > A[i-1]){

length++;

}else{

length = 1;

}

ans = Math.max(ans, length);

}

length = 1;

for (int i = A.length - 2; i >= 0; i--){

if (A[i+1] < A[i]){

length++;

}else{

length = 1;

}

ans = Math.max(ans, length);

}

return ans;

}

}

## 516\_LongestPalindromicSubsequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

Example 1:

Input:

"bbbab"

Output:

4

One possible longest palindromic subsequence is "bbbb".

Example 2:

Input:

"cbbd"

Output:

2

One possible longest palindromic subsequence is "bb".

Different from longest palindromic substring which characters must be continuous, but the solution is similar

dp[i][j]: the longest palindromic subsequence's length of substring(i, j), here i, j represent left, right indexes in the string

State transition:

dp[i][j] = dp[i+1][j-1] + 2 if s.charAt(i) == s.charAt(j)

otherwise, dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1])

Initialization: dp[i][i] = 1

Method 1: DP

class Solution {

public int longestPalindromeSubseq(String s) {

int n = s.length();

int[][] dp = new int[n][n];

for (int i = n - 1; i >= 0; i--){

for (int j = i; j < n; j++){

if (j == i){

dp[i][j] = 1;

}else{

dp[i][j] = s.charAt(i) == s.charAt(j) ? dp[i+1][j-1] + 2 : Math.max(dp[i][j-1], dp[i+1][j]);

}

}

}

return dp[0][n-1];

}

}

Method 2:

Top bottom recursive method with memoization

public class Solution {

public int longestPalindromeSubseq(String s) {

return helper(s, 0, s.length() - 1, new Integer[s.length()][s.length()]);

}

private int helper(String s, int i, int j, Integer[][] memo) {

if (memo[i][j] != null) {

return memo[i][j];

}

if (i > j) return 0;

if (i == j) return 1;

if (s.charAt(i) == s.charAt(j)) {

memo[i][j] = helper(s, i + 1, j - 1, memo) + 2;

} else {

memo[i][j] = Math.max(helper(s, i + 1, j, memo), helper(s, i, j - 1, memo));

}

return memo[i][j];

}

}

## 673\_LCIS\_Number\_LongestContinuousIncreasingSubsequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted array of integers, find the number of longest increasing subsequence.

Example 1:

Input: [1,3,5,4,7]

Output: 2

Explanation: The two longest increasing subsequence are [1, 3, 4, 7] and [1, 3, 5, 7].

Example 2:

Input: [2,2,2,2,2]

Output: 5

Explanation: The length of longest continuous increasing subsequence is 1, and there are 5 subsequences' length is 1, so output 5.

Note: Length of the given array will be not exceed 2000 and the answer is guaranteed to be fit in 32-bit signed int.

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public int findNumberOfLIS(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int[] lengths = new int[nums.length]; //lengths[i] = length of longest ending in nums[i]

int[] counts = new int[nums.length]; //count[i] = number of longest ending in nums[i]

for (int i = 0; i < nums.length; i++){

lengths[i] = 1;

}

Arrays.fill(counts, 1);

for (int i = 0; i < nums.length; i++){

for (int j = 0; j < i; j++){

if (nums[j] < nums[i]){

if (lengths[i] < lengths[j] + 1){

lengths[i] = lengths[j] + 1;

counts[i] = counts[j];

}else if (lengths[i] == lengths[j] + 1){

counts[i] += counts[j];

}

}

}

}

int longest = 0;

int count = 0;

for (int length : lengths){

longest = Math.max(longest, length);

}

for (int i = 0; i < nums.length; i++){

if (lengths[i] == longest){

count += counts[i];

}

}

return count;

}

}

Best solution:

class Solution {

public int findNumberOfLIS(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int[] lengths = new int[nums.length];

int[] counts = new int[nums.length];

int longest = 0;

int count = 0;

for (int i = 0; i < nums.length; i++){

lengths[i] = 1;

counts[i] = 1;

for (int j = 0; j < i; j++){

if (nums[j] < nums[i]){

int cand = lengths[j] + 1;

if (lengths[i] < cand){

lengths[i] = cand;

counts[i] = counts[j];

}else if (lengths[i] == cand){

counts[i] += counts[j];

}

}

}

if (lengths[i] == longest){

count += counts[i];

}else if (lengths[i] > longest){

longest = lengths[i];

count = counts[i];

}

}

return count;

}

}

class Solution {

public int findNumberOfLIS(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int[] lengths = new int[nums.length];

int[] counts = new int[nums.length];

for (int i = 0; i < nums.length; i++){

lengths[i] = 1;

}

Arrays.fill(counts, 1);

int longest = 0;

int count = 0;

for (int i = 0; i < nums.length; i++){

for (int j = 0; j < i; j++){

if (nums[j] < nums[i]){

if (lengths[i] < lengths[j] + 1){

lengths[i] = lengths[j] + 1;

counts[i] = counts[j];

}else if (lengths[i] == lengths[j] + 1){

counts[i] += counts[j];

}

}

}

}

for (int i = 0; i < nums.length; i++){

if (lengths[i] == longest){

count += counts[i];

}else if (lengths[i] > longest){

longest = lengths[i];

count = counts[i];

}

}

return count;

}

}

## 674\_LCIS\_LongestContinuousIncreasingSubsequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted array of integers, find the length of longest continuous increasing subsequence (subarray).

Example 1:

Input: [1,3,5,4,7]

Output: 3

Explanation: The longest continuous increasing subsequence is [1,3,5], its length is 3.

Even though [1,3,5,7] is also an increasing subsequence, it's not a continuous one where 5 and 7 are separated by 4.

Example 2:

Input: [2,2,2,2,2]

Output: 1

Explanation: The longest continuous increasing subsequence is [2], its length is 1.

class Solution {

public int findLengthOfLCIS(int[] nums) {

int max = 0;

int n = nums.length;

int i = 0;

int start = 0;

while (i < n){

while (i < n && ((i > 0 && nums[i] > nums[i-1]) || i == 0)){

i++;

}

max = Math.max(max, i - start);

start = i;

i++;

}

return max;

}

}

class Solution {

public int findLengthOfLCIS(int[] nums) {

int max = 0;

int n = nums.length;

int count = 0;

for (int i = 0; i < n; i++){

if (i == 0 || nums[i] > nums[i-1]){

max = Math.max(max, ++count);

}else{

count = 1;

}

}

return max;

}

}

## 1027\_LongestArithmeticSubsequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of integers, return the length of the longest arithmetic subsequence in A.

Recall that a subsequence of A is a list A[i\_1], A[i\_2], ..., A[i\_k] with 0 <= i\_1 < i\_2 < ... < i\_k <= A.length - 1, and that a sequence B is arithmetic if B[i+1] - B[i] are all the same value (for 0 <= i < B.length - 1).

Example 1:

Input: [3,6,9,12]

Output: 4

Explanation:

The whole array is an arithmetic sequence with steps of length = 3.

Example 2:

Input: [9,4,7,2,10]

Output: 3

Explanation:

The longest arithmetic subsequence is [4,7,10].

Example 3:

Input: [20,1,15,3,10,5,8]

Output: 4

Explanation:

The longest arithmetic subsequence is [20,15,10,5].

Note:

2 <= A.length <= 2000

0 <= A[i] <= 10000

Similar as Leetcode 446

https://github.com/optimisea/Leetcode/blob/master/Java/446\_Arithmetic.java

class Solution {

public int longestArithSeqLength(int[] A) {

int res = 0;

int n = A.length;

Map<Integer, Integer>[] maps = new Map[A.length];

for (int i = 0; i < n; i++){

maps[i] = new HashMap<>();

for (int j = 0; j < i; j++){

long diff = (long)A[i] - A[j];

if (diff >= Integer.MAX\_VALUE || diff <= Integer.MIN\_VALUE){

continue;

}

int d = (int)diff;

int c1 = maps[j].getOrDefault(d, 1);//the count at ending with index j and difference of d

// int c2 = maps[i].getOrDefault(d, 0);// the count at ending with index i and difference between A[i] and A[j] is d

res = Math.max(res, c1 + 1);

maps[i].put(d, c1 + 1);

}

}

return res;

}

}

# LinkedList

## 19\_RemoveNth.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a linked list, remove the nth node from the end of list and return its head.

For example,

Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.

Note:

Given n will always be valid.

Try to do this in one pass.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode removeNthFromEnd(ListNode head, int n) {

ListNode dummy = new ListNode(-1);

dummy.next = head;

ListNode slow = dummy;

ListNode fast = dummy;

int count = 1;

while (count <= n + 1 && fast != null){

fast = fast.next;

count++;

}

while (fast != null){

fast = fast.next;

slow = slow.next;

}

slow.next = slow.next.next;

return dummy.next;

}

}

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode removeNthFromEnd(ListNode head, int n) {

ListNode dummy = new ListNode(-1);

dummy.next = head;

ListNode slow = dummy;

ListNode fast = dummy;

int count = -1;

while (count < n){

fast = fast.next;

count++;

}

while (fast != null){

slow = slow.next;

fast = fast.next;

}

slow.next = slow.next.next;

return dummy.next;

}

}

## 21\_MergeTwoSortedLists.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Merge two sorted linked lists and return it as a new list. The new list should be made by

splicing together the nodes of the first two lists.

Example:

Input: 1->2->4, 1->3->4

Output: 1->1->2->3->4->4

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(0);

ListNode lastNode = dummy;

while (l1 != null && l2 != null){

if (l1.val > l2.val){

lastNode.next = l2;

l2 = l2.next;

}else{

lastNode.next = l1;

l1 = l1.next;

}

lastNode = lastNode.next;

}

if (l1 != null){

lastNode.next = l1;

}

if (l2 != null){

lastNode.next = l2;

}

return dummy.next;

}

}

## 23\_MergekSortedLists.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Method 1: divide & conquer

Time complexity: Nlogk

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode mergeKLists(ListNode[] lists) {

if (lists == null || lists.length == 0){

return null;

}

return mergeSort(lists, 0, lists.length - 1);

}

private ListNode mergeSort(ListNode[] lists, int start, int end){

if (start > end){

return null;

}

if (start == end){

return lists[start];

}

int mid = start + (end - start) / 2;

ListNode left = mergeSort(lists, start, mid);

ListNode right = mergeSort(lists, mid + 1, end);

return merge(left, right);

}

private ListNode merge(ListNode l1, ListNode l2){

if (l1 == null){

return l2;

}

if (l2 == null){

return l1;

}

ListNode dummy = new ListNode(-1);

ListNode node = dummy;

while (l1 != null && l2 != null){

if (l1.val < l2.val){

node.next = l1;

l1 = l1.next;

}else{

node.next = l2;

l2 = l2.next;

}

node = node.next;

}

if (l1 == null){

node.next = l2;

}

if (l2 == null){

node.next = l1;

}

return dummy.next;

}

}

Method 2: heap priority queue

class Solution {

public ListNode mergeKLists(ListNode[] lists) {

ListNode dummy = new ListNode(-1);

ListNode node = dummy;

Queue<ListNode> pq = new PriorityQueue<>(new Comparator<ListNode>(){

public int compare (ListNode n1, ListNode n2){

return n1.val - n2.val;

}

});

for (ListNode n : lists){

if (n != null){

pq.offer(n);

}

}

while (!pq.isEmpty()){

ListNode curr = pq.poll();

node.next = curr;

node = node.next;

if (curr.next != null){

pq.offer(curr.next);

}

}

return dummy.next;

}

}

## 24\_SwapNodes.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a linked list, swap every two adjacent nodes and return its head.

For example,

Given 1->2->3->4, you should return the list as 2->1->4->3.

Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

Method: use two nodes as a pair, so define three pointers: prev, cur, next

class Solution {

public ListNode swapPairs(ListNode head) {

if (head == null || head.next == null){

return head;

}

ListNode dummy = new ListNode(-1);

dummy.next = head;

ListNode prev = dummy;

ListNode cur = head;

ListNode next = head.next;

while (cur != null && next != null){

ListNode temp = next.next;

next.next = cur;

cur.next = temp;

prev.next = next;

prev = cur;

cur = temp;

if (temp != null){

next= temp.next;

}else{

next = null;

}

}

return dummy.next;

}

}

## 83\_RemoveDuplicates.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,

Given 1->1->2, return 1->2.

Given 1->1->2->3->3, return 1->2->3.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

Method 1:

class Solution {

public ListNode deleteDuplicates(ListNode head) {

if (head == null){

return head;

}

ListNode slow = head;

ListNode fast = head;

while (fast != null){

if (slow.val != fast.val){

slow.next = fast;

slow = fast;

}

fast = fast.next;

}

slow.next = null;

return head;

}

}

Method 2:

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode deleteDuplicates(ListNode head) {

ListNode cur = head;

while (cur != null && cur.next != null){

if (cur.val == cur.next.val){

cur.next = cur.next.next;

}else{

cur = cur.next;

}

}

return head;

}

}

## 105\_CopyListwithRandomPointer.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

Have you met this question in a real interview? Yes

Example

Challenge

Could you solve it with O(1) space?

/\*\*

\* Definition for singly-linked list with a random pointer.

\* class RandomListNode {

\* int label;

\* RandomListNode next, random;

\* RandomListNode(int x) { this.label = x; }

\* };

\*/

public class Solution {

/\*\*

\* @param head: The head of linked list with a random pointer.

\* @return: A new head of a deep copy of the list.

\*/

public class Solution {

public RandomListNode copyRandomList(RandomListNode head) {

if (head == null){

return head;

}

Map<RandomListNode, RandomListNode> map = new HashMap<>();

RandomListNode dummy = new RandomListNode(0);

RandomListNode newHead = dummy;

RandomListNode t = head;

while (head != null){

RandomListNode node = new RandomListNode(head.label);

map.put(head, node);

newHead.next = node;

newHead = node;

head = head.next;

}

head = t;

while (head != null){

RandomListNode temp = map.get(head);

temp.random = map.get(head.random);

head = head.next;

}

return dummy.next;

}

}

/\*\*

\* Definition for singly-linked list with a random pointer.

\* class RandomListNode {

\* int label;

\* RandomListNode next, random;

\* RandomListNode(int x) { this.label = x; }

\* };

\*/

public class Solution {

public RandomListNode copyRandomList(RandomListNode head) {

Map<RandomListNode, RandomListNode> map = new HashMap<>();

if (head == null){

return null;

}

RandomListNode node = head;

RandomListNode dummy = new RandomListNode(-1);

RandomListNode newnode = dummy;

while (node != null){

RandomListNode temp = new RandomListNode(node.label);

map.put(node, temp);

node = node.next;

newnode.next = temp;

newnode = newnode.next;

}

node = head;

newnode = dummy.next;

while (node != null){

newnode.random = map.get(node.random);

newnode = newnode.next;

node = node.next;

}

return dummy.next;

}

}

Better version:

/\*\*

\* Definition for singly-linked list with a random pointer.

\* class RandomListNode {

\* int label;

\* RandomListNode next, random;

\* RandomListNode(int x) { this.label = x; }

\* };

\*/

public class Solution {

public RandomListNode copyRandomList(RandomListNode head) {

Map<RandomListNode, RandomListNode> map = new HashMap<>();

RandomListNode dummy = new RandomListNode(-1);

RandomListNode copyNode = dummy;

RandomListNode node = head;

while (node != null){

copyNode.next = new RandomListNode(node.label);

copyNode = copyNode.next;

map.put(node, copyNode);

node = node.next;

}

copyNode = dummy.next;

node = head;

while (node != null){

copyNode.random = map.get(node.random);

copyNode = copyNode.next;

node = node.next;

}

return dummy.next;

}

}

## 129\_RehashingKeysForHashTable.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The size of the hash table is not determinate at the very beginning. If the total size of keys is too large

(e.g. size >= capacity / 10), we should double the size of the hash table and rehash every keys. Say you have

a hash table looks like below:

size=3, capacity=4

[null, 21, 14, null]

↓ ↓

9 null

↓

null

The hash function is:

int hashcode(int key, int capacity) {

return key % capacity;

}

here we have three numbers, 9, 14 and 21, where 21 and 9 share the same position as they all have the same

hashcode 1 (21 % 4 = 9 % 4 = 1). We store them in the hash table by linked list.

rehashing this hash table, double the capacity, you will get:

size=3, capacity=8

index: 0 1 2 3 4 5 6 7

hash : [null, 9, null, null, null, 21, 14, null]

Given the original hash table, return the new hash table after rehashing .

Notice

For negative integer in hash table, the position can be calculated as follow:

C++/Java: if you directly calculate -4 % 3 you will get -1. You can use function: a % b = (a % b + b) % b to

make it is a non negative integer.

Python: you can directly use -1 % 3, you will get 2 automatically.

Have you met this question in a real interview? Yes

Example

Given [null, 21->9->null, 14->null, null],

return [null, 9->null, null, null, null, 21->null, 14->null, null]

/\*\*

\* Definition for ListNode

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

/\*\*

\* @param hashTable: A list of The first node of linked list

\* @return: A list of The first node of linked list which have twice size

\*/

public ListNode[] rehashing(ListNode[] hashTable) {

if (hashTable == null || hashTable.length == 0){

return hashTable;

}

int newCapacity = 2 \* hashTable.length;

ListNode[] newTable = new ListNode[newCapacity];

for (int i = 0; i < hashTable.length; i++){

while (hashTable[i] != null){

int newIndex = (hashTable[i].val % newCapacity + newCapacity) % newCapacity;

if (newTable[newIndex] == null){

newTable[newIndex] = new ListNode(hashTable[i].val);

}else{

ListNode dummy = newTable[newIndex];

while (dummy.next != null){

dummy = dummy.next;

}

dummy.next = new ListNode(hashTable[i].val);

}

hashTable[i] = hashTable[i].next;

}

}

return newTable;

}

};

## 141\_DetectLinkedListCycle.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a linked list, determine if it has a cycle in it.

Follow up:

Can you solve it without using extra space?

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public boolean hasCycle(ListNode head) {

if (head == null){

return false;

}

ListNode slow = head;

ListNode fast = head;

while (fast != null && fast.next != null){

slow = slow.next;

fast = fast.next.next;

if (slow == fast){

return true;

}

}

return false;

}

}

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public boolean hasCycle(ListNode head) {

if (head == null){

return false;

}

ListNode fast = head.next;

ListNode slow = head;

while (fast != null && fast.next != null && fast != slow){

fast = fast.next.next;

slow = slow.next;

}

if (fast == null || fast.next == null){

return false;

}

return true;

}

}

## 142\_LinkedListCycleII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Note: Do not modify the linked list.

Follow up:

Can you solve it without using extra space?

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode detectCycle(ListNode head) {

if (head == null){

return head;

}

ListNode slow = head;

ListNode fast = head;

ListNode entry = head;

while (fast != null && fast.next != null){

slow = slow.next;

fast = fast.next.next;

if (slow == fast){

while (slow != entry){

slow = slow.next;

entry = entry.next;

}

return entry;

}

}

return null;

}

}

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode detectCycle(ListNode head) {

if (head == null){

return null;

}

ListNode slow = head;

ListNode fast = head;

boolean hasCylce = false;

while (fast != null && fast.next != null){

fast = fast.next.next;

slow = slow.next;

if (fast == slow){

hasCylce = true;

break;

}

}

if (!hasCylce){

return null;

}

ListNode node = head;

while (node != slow){

node = node.next;

slow = slow.next;

}

return node;

}

}

## 147\_InsertionSortLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sort a linked list using insertion sort.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode insertionSortList(ListNode head) {

if (head == null){

return head;

}

ListNode dummy = new ListNode(Integer.MIN\_VALUE);

dummy.next = head;

ListNode prev = dummy;

ListNode cur = head;

while (cur != null){

if (cur.val >= prev.val){

prev = cur;

cur = cur.next;

continue;

}

ListNode node = dummy;

while (cur.val > node.next.val){

node = node.next;

}

ListNode temp = cur.next;

cur.next = node.next;

node.next = cur;

cur = temp;

prev.next = temp;

}

return dummy.next;

}

}

## 148\_SortLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sort a linked list in O(n log n) time using constant space complexity.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode sortList(ListNode head) {

if (head == null || head.next == null){

return head;

}

ListNode middle = findMiddle(head);

ListNode right = sortList(middle.next);

middle.next = null;

ListNode left = sortList(head);

return merge(left, right);

}

private ListNode findMiddle(ListNode head){

if (head == null || head.next == null){

return head;

}

ListNode fast = head.next;

ListNode slow = head;

while (fast != null && fast.next != null){

fast = fast.next.next;

slow = slow.next;

}

return slow;

}

private ListNode merge(ListNode l1, ListNode l2){

ListNode dummy = new ListNode(0);

ListNode lastNode = dummy;

while (l1 != null && l2 != null){

if (l1.val > l2.val){

lastNode.next = l2;

l2 = l2.next;

}else{

lastNode.next = l1;

l1 = l1.next;

}

lastNode = lastNode.next;

}

if (l1 != null){

lastNode.next = l1;

}

if (l2 != null){

lastNode.next = l2;

}

return dummy.next;

}

}

## 160\_IntersectionTwoLinkedLists.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

A: a1 → a2

↘

c1 → c2 → c3

↗

B: b1 → b2 → b3

begin to intersect at node c1.

Notes:

If the two linked lists have no intersection at all, return null.

The linked lists must retain their original structure after the function returns.

You may assume there are no cycles anywhere in the entire linked structure.

Your code should preferably run in O(n) time and use only O(1) memory.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if (headA == null || headB == null){

return null;

}

ListNode currA = headA;

int countA = 0;

while (currA != null){

countA++;

currA = currA.next;

}

ListNode currB = headB;

int countB = 0;

while (currB != null){

countB++;

currB = currB.next;

}

currA = headA;

currB = headB;

if (countA < countB){

for (int i = 0; i < countB - countA && currB != null; i++){

currB = currB.next;

}

}else{

for (int i = 0; i < countA - countB && currA != null; i++){

currA = currA.next;

}

}

while (currA != null && currB != null){

if (currA == currB){

return currA;

}

currA = currA.next;

currB = currB.next;

}

return null;

}

}

Better version:

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

int lenA = getLen(headA);

int lenB = getLen(headB);

ListNode nodeA = headA;

ListNode nodeB = headB;

if (lenA > lenB){

int diff = lenA - lenB;

while (nodeA != null && diff >0){

nodeA = nodeA.next;

diff--;

}

}else{

int diff = lenB - lenA;

while (nodeB != null && diff >0){

nodeB = nodeB.next;

diff--;

}

}

while (nodeA != null && nodeB != null){

if (nodeA == nodeB){

return nodeA;

}

nodeA = nodeA.next;

nodeB = nodeB.next;

}

return null;

}

private int getLen(ListNode head){

ListNode node = head;

int len = 0;

while (node != null){

len++;

node = node.next;

}

return len;

}

}

## 167\_AddTwoNumbersLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in reverse order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

Have you met this question in a real interview? Yes

Example

Given 7->1->6 + 5->9->2. That is, 617 + 295.

Return 2->1->9. That is 912.

Given 3->1->5 and 5->9->2, return 8->0->8.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

The same as array addition

public class Solution {

/\*

\* @param l1: the first list

\* @param l2: the second list

\* @return: the sum list of l1 and l2

\*/

public ListNode addLists(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(-1);

ListNode tail = dummy;

int carry = 0;

for (ListNode i = l1, j = l2; i != null || j != null;){

int sum = carry;

sum += (i != null) ? i.val : 0;

sum += (j != null) ? j.val : 0;

ListNode newNode = new ListNode(sum % 10);

tail.next = newNode;

tail = newNode;

carry = sum / 10;

i = (i != null) ? i.next : i;

j = (j != null) ? j.next : j;

}

if (carry != 0){

tail.next = new ListNode(carry);

}

return dummy.next;

}

}

Best solution

class Solution {

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(-1);

ListNode head = dummy;

ListNode n1= l1;

ListNode n2 = l2;

int carry = 0;

int sum = 0;

while (n1 != null || n2 != null){

sum = carry;

sum += (n1 != null) ? n1.val : 0;

sum += (n2 != null) ? n2.val : 0;

carry = sum / 10;

ListNode node = new ListNode(sum % 10);

head.next = node;

head = node;

n1 = (n1 != null) ? n1.next : n1;

n2 = (n2 != null) ? n2.next : n2;

}

if (carry != 0){

head.next = new ListNode(carry);

}

return dummy.next;

}

}

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(0);

ListNode node = dummy;

ListNode node1 = l1;

ListNode node2 = l2;

int digit = 0;

int val = 0;

while (node1 != null && node2 != null){

val = node1.val + node2.val + digit;

node.next = new ListNode(val%10);

digit = val / 10;

node = node.next;

node1 = node1.next;

node2 = node2.next;

}

if (node1 == null){

while (node2 != null){

val = digit + node2.val;

node.next = new ListNode(val%10);

digit = val / 10;

node = node.next;

node2 = node2.next;

}

}else{

while (node1 != null){

val = digit + node1.val;

node.next = new ListNode(val%10);

digit = val / 10;

node = node.next;

node1 = node1.next;

}

}

if (digit > 0){

node.next = new ListNode(digit);

}

return dummy.next;

}

}

## 170\_RotateLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list, rotate the list to the right by k places, where k is non-negative.

Have you met this question in a real interview? Yes

Example

Given 1->2->3->4->5 and k = 2, return 4->5->1->2->3.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

/\*

\* @param head: the List

\* @param k: rotate to the right k places

\* @return: the list after rotation

\*/

public ListNode rotateRight(ListNode head, int k) {

if (head == null){

return head;

}

int len = getLength(head);

k = k % len;

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode slow = head;

ListNode fast = head;

for (int i = 0; i < k; i++){

fast = fast.next;

}

while (fast.next != null){

slow = slow.next;

fast = fast.next;

}

fast.next = dummy.next;

dummy.next = slow.next;

slow.next = null;

return dummy.next;

}

private int getLength(ListNode head){

int len = 0;

while (head != null){

len++;

head = head.next;

}

return len;

}

}

## 203\_RemoveLinkedListElements.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Remove all elements from a linked list of integers that have value val.

Example

Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, val = 6

Return: 1 --> 2 --> 3 --> 4 --> 5

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode removeElements(ListNode head, int val) {

if (head == null){

return head;

}

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode cur = dummy;

while (cur.next != null){

if (cur.next.val == val){

cur.next = cur.next.next;

}else{

cur = cur.next;

}

}

return dummy.next;

}

}

## 206\_ReverseLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Reverse a singly linked list.

public class ListNode{

int val;

ListNode next;

ListNode(int x){

val = x;

}

}

click to show more hints.

Hint:

A linked list can be reversed either iteratively or recursively. Could you implement both?

Method 1: iteration

class Solution {

public ListNode reverseList(ListNode head) {

ListNode prev = null;

while (head != null){

ListNode temp = head.next;

head.next = prev;

prev = head;

head = temp;

}

return prev;

}

}

Method 2: recurstion

class Solution {

public ListNode reverseList(ListNode head) {

return reverse(head, null);

}

ListNode reverse(ListNode head, ListNode newHead){

if (head == null){

return newHead;

}

ListNode temp = head.next;

head.next = newHead;

return reverse(temp, head);

}

}

## 234\_PalindromeLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a singly linked list, determine if it is a palindrome.

Follow up:

Could you do it in O(n) time and O(1) space?

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isPalindrome(ListNode head) {

if (head == null || head.next == null){

return true;

}

ListNode cur = head;

int len = 0;

while (cur != null){

len++;

cur = cur.next;

}

cur = head;

ListNode middleNode = findMiddle(cur);

ListNode newHead2 = middleNode.next;

cur = head;

middleNode.next = null;

ListNode newHead = reverse(cur);

if (len %2 == 1){

newHead = newHead.next;

}

while (newHead != null && newHead2 != null){

if (newHead.val != newHead2.val){

return false;

}

newHead = newHead.next;

newHead2 = newHead2.next;

}

return true;

}

private ListNode findMiddle(ListNode head){

ListNode slow = head;

ListNode fast = head.next;

while (fast != null && fast.next != null){

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

private ListNode reverse(ListNode head){

ListNode prev = null;

ListNode cur = head;

while (cur != null){

ListNode temp = cur.next;

cur.next = prev;

prev = cur;

cur = temp;

}

return prev;

}

}

Better version:

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isPalindrome(ListNode head) {

if (head == null || head.next == null){

return true;

}

//find middle

ListNode slow = head;

ListNode fast = head.next;

while (fast != null && fast.next != null){

fast = fast.next.next;

slow = slow.next;

}

ListNode halfHead = slow.next;

slow.next = null;

//reverse the 2nd half

ListNode prev = null;

ListNode node = halfHead;

while (node != null){

ListNode temp = node.next;

node.next = prev;

prev = node;

node = temp;

}

ListNode newHead = prev;

ListNode node2 = newHead;

ListNode node1 = head;

while (node2 != null){

if (node1.val != node2.val){

return false;

}

node1 = node1.next;

node2 = node2.next;

}

return true;

}

}

## 237\_DeleteNodeInLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Supposed the linked list is 1 -> 2 -> 3 -> 4 and you are given the third node with value 3, the linked list

should become 1 -> 2 -> 4 after calling your function.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public void deleteNode(ListNode node) {

if (node == null){

return;

}

ListNode prev = null;

while (node.next != null){

node.val = node.next.val;

prev = node;

node = node.next;

}

prev.next = null;

}

}

## 328\_OddEvenLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a singly linked list, group all odd nodes together followed by the even nodes.

Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

Example:

Given 1->2->3->4->5->NULL,

return 1->3->5->2->4->NULL.

Note:

The relative order inside both the even and odd groups should remain as it was in the input.

The first node is considered odd, the second node even and so on ...

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode oddEvenList(ListNode head) {

ListNode dummy1 = new ListNode(0);

ListNode dummy2 = new ListNode(0);

ListNode l1 = dummy1;

ListNode l2 = dummy2;

dummy1.next = head;

ListNode curr = head;

int count = 1;

while (curr != null){

if (count % 2 == 1){

l1.next = curr;

l1 = l1.next;

curr = curr.next;

l1.next = null;

}else{

l2.next = curr;

l2 = l2.next;

curr = curr.next;

l2.next = null;

}

count++;

}

l1.next = dummy2.next;

dummy2.next = null;

return dummy1.next;

}

}

Method 2: Best solution

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode oddEvenList(ListNode head) {

if (head == null){

return head;

}

ListNode odd = head;

ListNode evenHead = odd.next;

ListNode even = evenHead;

while (even != null && even.next != null){

odd.next = odd.next.next;

even.next = even.next.next;

odd = odd.next;

even = even.next;

}

odd.next = evenHead;

return head;

}

}

## 817\_Linked.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given head, the head node of a linked list containing unique integer values.

We are also given the list G, a subset of the values in the linked list.

Return the number of connected components in G, where two values are connected if they appear consecutively in the linked list.

Example 1:

Input:

head: 0->1->2->3

G = [0, 1, 3]

Output: 2

Explanation:

0 and 1 are connected, so [0, 1] and [3] are the two connected components.

Example 2:

Input:

head: 0->1->2->3->4

G = [0, 3, 1, 4]

Output: 2

Explanation:

0 and 1 are connected, 3 and 4 are connected, so [0, 1] and [3, 4] are the two connected components.

Note:

If N is the length of the linked list given by head, 1 <= N <= 10000.

The value of each node in the linked list will be in the range [0, N - 1].

1 <= G.length <= 10000.

G is a subset of all values in the linked list.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public int numComponents(ListNode head, int[] G) {

Set<Integer> set = new HashSet<>();

for (int i : G){

set.add(i);

}

int ans = 0;

while (head != null){

if (set.contains(head.val) && (head.next == null || !set.contains(head.next.val))){

ans++;

}

head = head.next;

}

return ans;

}

}

https://leetcode.com/problems/linked-list-components/discuss/123842/C++JavaPython-Easy-and-Concise-Solution-with-Explanation

818\_Race.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Your car starts at position 0 and speed +1 on an infinite number line. (Your car can go into negative positions.)

Your car drives automatically according to a sequence of instructions A (accelerate) and R (reverse).

When you get an instruction "A", your car does the following: position += speed, speed \*= 2.

When you get an instruction "R", your car does the following: if your speed is positive then speed = -1 , otherwise speed = 1.

(Your position stays the same.)

For example, after commands "AAR", your car goes to positions 0->1->3->3, and your speed goes to 1->2->4->-1.

Now for some target position, say the length of the shortest sequence of instructions to get there.

Example 1:

Input:

target = 3

Output: 2

Explanation:

The shortest instruction sequence is "AA".

Your position goes from 0->1->3.

Example 2:

Input:

target = 6

Output: 5

Explanation:

The shortest instruction sequence is "AAARA".

Your position goes from 0->1->3->7->7->6.

Note:

1 <= target <= 10000.

https://leetcode.com/problems/race-car/discuss/123834/C++JavaPython-DP-solution

## 1030\_NextLargerNode.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given a linked list with head as the first node. Let's number the nodes in the list: node\_1, node\_2, node\_3, ... etc.

Each node may have a next larger value: for node\_i, next\_larger(node\_i) is the node\_j.val such that j > i, node\_j.val > node\_i.val, and j is the smallest possible choice. If such a j does not exist, the next larger value is 0.

Return an array of integers answer, where answer[i] = next\_larger(node\_{i+1}).

Note that in the example inputs (not outputs) below, arrays such as [2,1,5] represent the serialization of a linked list with a head node value of 2, second node value of 1, and third node value of 5.

Example 1:

Input: [2,1,5]

Output: [5,5,0]

Example 2:

Input: [2,7,4,3,5]

Output: [7,0,5,5,0]

Example 3:

Input: [1,7,5,1,9,2,5,1]

Output: [7,9,9,9,0,5,0,0]

Note:

1 <= node.val <= 10^9 for each node in the linked list.

The given list has length in the range [0, 10000].

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public int[] nextLargerNodes(ListNode head) {

ListNode node = head;

List<Integer> list = new ArrayList<>();

while (node != null){

list.add(node.val);

node = node.next;

}

int n = list.size();

int[] res = new int[n];

int[] A = new int[n];

for (int i = 0; i < n; i++){

A[i] = list.get(i);

}

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < n; i++){

while (!stack.isEmpty() && A[stack.peek()] < A[i]){

res[stack.pop()] = A[i];

}

stack.push(i);

}

return res;

}

}

# BinaryTree

## 70\_BinaryTreeLevelOrderTraversalII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the bottom-up level order traversal of its nodes' values.

(ie, from left to right, level by level from leaf to root).

Have you met this question in a real interview? Yes

Example

Given binary tree {3,9,20,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its bottom-up level order traversal as:

[

[15,7],

[9,20],

[3]

]

/\*\*

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

public class Solution {

/\*

\* @param root: A tree

\* @return: buttom-up level order a list of lists of integer

\*/

public List<List<Integer>> levelOrderBottom(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null){

return result;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

List<Integer> level = new ArrayList<>();

int size = queue.size();

for (int i = 0; i < size; i++){

TreeNode cur = queue.poll();

level.add(cur.val);

if (cur.left != null){

queue.offer(cur.left);

}

if (cur.right != null){

queue.offer(cur.right);

}

}

result.add(level);

}

Collections.reverse(result);

return result;

}

}

## 236\_LowestCommonAncestorBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v

and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).”

\_\_\_\_\_\_\_3\_\_\_\_\_\_

/ \

\_\_\_5\_\_ \_\_\_1\_\_

/ \ / \

6 \_2 0 8

/ \

7 4

For example, the lowest common ancestor (LCA) of nodes 5 and 1 is 3. Another example is LCA of

nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Method: divide and conquer

class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if (root == null || root == p || root == q){

return root;

}

TreeNode left = lowestCommonAncestor(root.left, p, q);

TreeNode right = lowestCommonAncestor(root.right, p, q);

if (left != null && right != null){

return root;

}

if (left == null){

return right;

}

if (right == null){

return left;

}

return null;

}

}

class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

return lca(root, p, q);

}

private TreeNode lca(TreeNode root, TreeNode p, TreeNode q){

if (root == null){

return null;

}

if (root.val == p.val || root.val == q.val){

return root;

}

TreeNode left = lca(root.left, p, q);

TreeNode right = lca(root.right, p, q);

if (left == null && right == null){

return null;

}

if (left == null){

return right;

}

if (right == null){

return left;

}

return root;

}

}

## 298\_BinaryTreeLongestConsecutiveSequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections.

The longest consecutive path need to be from parent to child (cannot be the reverse).

For example,

1

\

3

/ \

2 4

\

5

Longest consecutive sequence path is 3-4-5, so return 3.

2

\

3

/

2

/

1

Longest consecutive sequence path is 2-3,not3-2-1, so return 2.

Method: tranverse

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

private int max = 0;

public int longestConsecutive(TreeNode root) {

helper(root, null, 0);

return max;

}

private void helper(TreeNode root, TreeNode parent, int lengthWithoutRoot){

if (root == null){

return;

}

int len = (parent != null && parent.val + 1 == root.val) ? lengthWithoutRoot + 1 : 1;

helper(root.left, root, len);

helper(root.right, root, len);

max = Math.max(max, len);

}

}

Method 2: Best solution

Refer to https://github.com/optimisea/Leetcode/blob/master/Java/549\_BinaryTree.java

Refer to https://github.com/optimisea/Leetcode/blob/master/Java/687\_Longest.java

int max = 0;

public int longestConsecutive(TreeNode root){

longestIncludeRoot(root);

return max;

}

private int longestIncludeRoot(TreeNode root){

if (root == null){

return 0;

}

int left = longestIncludeRoot(root.left);

int right = longestIncludeRoot(root.right);

int leftIncludeRoot = 1;

int rightIncludeRoot = 1;

if (root.left != null && (root.left.val > root.val){

leftIncludeRoot = left + 1;

}

if (root.right != null && root.right.val > root.val){

rightIncludeRoot = right + 1);

}

max = Math.max(max, Math.max(leftIncludeRoot, rightIncludeRoot));

return Math.max(leftIncludeRoot, rightIncludeRoot);

}

## 448\_InorderSuccessorinBinarySearchTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary search tree (See Definition) and a node in it, find the in-order successor of that node in the BST.

If the given node has no in-order successor in the tree, return null.

Notice

It's guaranteed p is one node in the given tree. (You can directly compare the memory address to find p)

Have you met this question in a real interview? Yes

Example

Given tree = [2,1] and node = 1:

2

/

1

return node 2.

Given tree = [2,1,3] and node = 2:

2

/ \

1 3

return node 3.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

• 节点的successor 后继就是比给定节点大的所有节点中最小的那个

思路:

• 考虑p和root之间的关系(p 为给定的节点,要找到p节点的后继)

• 两种情况:

1. root的值 =< p的值 答案就在右子树中

2. root的值 > p的值 答案有两个备选: a) 就是root

b) 左子树中继续找,并且如果找到就一定是它,因为左子树的中的元素都比根小

public class Solution {

/\*

\* @param root: The root of the BST.

\* @param p: You need find the successor node of p.

\* @return: Successor of p.

\*/

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

if (root == null || p == null){

return null;

}

if (p.val >= root.val){

return inorderSuccessor(root.right, p);

}else{

TreeNode left = inorderSuccessor(root.left, p);

return (left != null) ? left : root;

}

}

}

## 545\_BoundaryofBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the values of its boundary in anti-clockwise direction starting from root.

Boundary includes left boundary, leaves, and right boundary in order without duplicate nodes.

Left boundary is defined as the path from root to the left-most node. Right boundary is defined as

the path from root to the right-most node. If the root doesn't have left subtree or right subtree,

then the root itself is left boundary or right boundary. Note this definition only applies to the input

binary tree, and not applies to any subtrees.

The left-most node is defined as a leaf node you could reach when you always firstly travel to the left

subtree if exists. If not, travel to the right subtree. Repeat until you reach a leaf node.

The right-most node is also defined by the same way with left and right exchanged.

Example 1

Input:

1

\

2

/ \

3 4

Ouput:

[1, 3, 4, 2]

Explanation:

The root doesn't have left subtree, so the root itself is left boundary.

The leaves are node 3 and 4.

The right boundary are node 1,2,4. Note the anti-clockwise direction means you should output reversed right boundary.

So order them in anti-clockwise without duplicates and we have [1,3,4,2].

Example 2

Input:

\_\_\_\_1\_\_\_\_\_

/ \

2 3

/ \ /

4 5 6

/ \ / \

7 8 9 10

Ouput:

[1,2,4,7,8,9,10,6,3]

Explanation:

The left boundary are node 1,2,4. (4 is the left-most node according to definition)

The leaves are node 4,7,8,9,10.

The right boundary are node 1,3,6,10. (10 is the right-most node).

So order them in anti-clockwise without duplicate nodes we have [1,2,4,7,8,9,10,6,3].

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

List<Integer> result = new ArrayList<>();

public List<Integer> boundaryOfBinaryTree(TreeNode root) {

if (root == null){

return result;

}

result.add(root.val);

leftBound(root.left);

leaves(root.left);

leaves(root.right);

rightBound(root.right);

return result;

}

private void leftBound(TreeNode root){

if (root == null || (root.left == null && root.right == null)){

return;

}

result.add(root.val);

if (root.left == null){

leftBound(root.right);

}else{

leftBound(root.left);

}

}

private void rightBound(TreeNode root){

if (root == null || (root.left == null && root.right == null)){

return;

}

if (root.right == null){

rightBound(root.left);

}else{

rightBound(root.right);

}

result.add(root.val);

}

private void leaves(TreeNode root){

if (root == null){

return;

}

if (root.left == null && root.right == null){

result.add(root.val);

}

leaves(root.left);

leaves(root.right);

}

}

## 549\_LongestConsecutivePathInBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, you need to find the length of Longest Consecutive Path in Binary Tree.

Especially, this path can be either increasing or decreasing. For example, [1,2,3,4] and [4,3,2,1]

are both considered valid, but the path [1,2,4,3] is not valid. On the other hand, the path can be

in the child-Parent-child order, where not necessarily be parent-child order.

Example 1:

Input:

1

/ \

2 3

Output: 2

Explanation: The longest consecutive path is [1, 2] or [2, 1].

Example 2:

Input:

2

/ \

1 3

Output: 3

Explanation: The longest consecutive path is [1, 2, 3] or [3, 2, 1].

Note: All the values of tree nodes are in the range of [-1e7, 1e7].

This solution is very simple. With every node, we associate two values/variables named inrinrinr and dcrdcrdcr,

where incrincrincr represents the length of the longest incrementing branch below the current node including itself,

and dcrdcrdcr represents the length of the longest decrementing branch below the current node including itself.

We make use of a recursive function longestPath(node) which returns an array of the form [inr,dcr][inr, dcr][inr,dcr]

for the calling node. We start off by assigning both inrinrinr and dcrdcrdcr as 1 for the current node. This is

because the node itself always forms a consecutive increasing as well as decreasing path of length 1.

Then, we obtain the length of the longest path for the left child of the current node using longestPath[root.left].

Now, if the left child is just smaller than the current node, it forms a decreasing sequence with the current node.

Thus, the dcrdcrdcr value for the current node is stored as the left child's dcrdcrdcr value + 1. But, if the left

child is just larger than the current node, it forms an incrementing sequence with the current node. Thus, we update

the current node's inrinrinr value as left\_child(inr)+1left\\_child(inr) + 1left\_child(inr)+1.

Then, we do the same process with the right child as well. But, for obtaining the inrinrinr and dcrdcrdcr value for

the current node, we need to consider the maximum value out of the two values obtained from the left and the right

child for both inrinrinr and dcrdcrdcr, since we need to consider the longest sequence possible.

Further, after we've obtained the final updated values of inrinrinr and dcrdcrdcr for a node, we update the

length of the longest consecutive path found so far as maxval=max(inr+dcr−1)maxval = \text{max}(inr + dcr - 1)

maxval=max(inr+dcr−1).

Complexity Analysis

Time complexity : O(n). The whole tree is traversed only once.

Space complexity : O(n). The recursion goes upto a depth of nnn in the worst case.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

private int max = 0;

public int longestConsecutive(TreeNode root) {

traverse(root);

return max;

}

private int[] traverse(TreeNode root){

if (root == null){

return new int[]{0, 0};

}

int[] left = traverse(root.left);

int[] right = traverse(root.right);

int decr = 1;

int incr = 1;

if (root.left != null){

if (root.val == root.left.val + 1){

decr = left[0] + 1;

}else if (root.val == root.left.val - 1){

incr = left[1] + 1;

}

}

if (root.right != null){

if (root.val == root.right.val + 1){

decr = Math.max(decr, right[0] + 1);

}else if (root.val == root.right.val - 1){

incr = Math.max(incr, right[1] + 1);

}

}

max = Math.max(max, decr + incr - 1);

return new int[]{decr, incr};

}

}

Best solution

https://github.com/optimisea/Leetcode/blob/master/Java/687\_Longest.java

class Solution {

public int longestConsecutive(TreeNode root) {

int[] max = new int[1];

longestIncludeRoot(root, max);

return max[0];

}

private int[] longestIncludeRoot(TreeNode root){

if (root == null){

return new int[]{0, 0};//{incr, decr}

}

int[] left = longestIncludeRoot(root.left);

int[] right = longestIncludeRoot(root.right);

int[] incr = new int[]{1, 1};//incr[0]: left longest; incr[1]: right longest

int[] decr = new int[]{1, 1};//decr[0]: left longest: decr[1]: right longest

if (root.left != null){

if (root.val + 1 == root.left.val){

incr[0] = left[0] + 1;

}else if (root.val - 1 == root.left.val){

decr[0] = left[1] + 1;

}

}

if (root.right != null){

if (root.val + 1 == root.right.val){

incr[1] = right[0] + 1;

}else if (root.val - 1 == root.right.val){

decr[1] = right[1] + 1;

}

}

max[0] = Math.max(max[0], Math.max(incr[0] + decr[1] - 1, incr[1] + decr[0] - 1));

return new int[]{Math.max(incr[0], incr[1]), Math.max(decr[0], decr[1]};

}

}

## 617\_MergeTwoBinaryTrees.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two binary trees and imagine that when you put one of them to cover the other, some nodes of

the two trees are overlapped while the others are not.

You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then

sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.

Example 1:

Input:

Tree 1 Tree 2

1 2

/ \ / \

3 2 1 3

/ \ \

5 4 7

Output:

Merged tree:

3

/ \

4 5

/ \ \

5 4 7

Note: The merging process must start from the root nodes of both trees.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {

if (t1 == null && t2 == null){

return null;

}

int val = 0;

if (t1 != null){

val += t1.val;

}

if (t2 != null){

val += t2.val;

}

TreeNode root = new TreeNode(val);

root.left = mergeTrees(t1 == null ? null : t1.left, t2 == null ? null : t2.left);

root.right = mergeTrees(t1 == null ? null : t1.right, t2 == null ? null : t2.right);

return root;

}

}

Better version:

class Solution {

public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {

if (t1 == null){

return t2;

}

if (t2 == null){

return t1;

}

TreeNode root = new TreeNode(t1.val + t2.val);

root.left = mergeTrees(t1.left, t2.left);

root.right = mergeTrees(t1.right, t2.right);

return root;

}

}

## 623\_AddOneRowToTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the root of a binary tree, then value v and depth d, you need to add a row of nodes with value v at the given depth d. The root node is at depth 1.

The adding rule is: given a positive integer depth d, for each NOT null tree nodes N in depth d-1, create two tree nodes with value v as N's left subtree root and right subtree root. And N's original left subtree should be the left subtree of the new left subtree root, its original right subtree should be the right subtree of the new right subtree root. If depth d is 1 that means there is no depth d-1 at all, then create a tree node with value v as the new root of the whole original tree, and the original tree is the new root's left subtree.

Example 1:

Input:

A binary tree as following:

4

/ \

2 6

/ \ /

3 1 5

v = 1

d = 2

Output:

4

/ \

1 1

/ \

2 6

/ \ /

3 1 5

Example 2:

Input:

A binary tree as following:

4

/

2

/ \

3 1

v = 1

d = 3

Output:

4

/

2

/ \

1 1

/ \

3 1

Note:

The given d is in range [1, maximum depth of the given tree + 1].

The given binary tree has at least one tree node.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode addOneRow(TreeNode root, int v, int d) {

if (d == 1){

TreeNode newRoot = new TreeNode(v);

newRoot.left = root;

return newRoot;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

int level = 0;

while (!queue.isEmpty()){

level++;

int size = queue.size();

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

if (level == d - 1){

TreeNode leftNode = new TreeNode(v);

leftNode.left = node.left;

node.left = leftNode;

TreeNode rightNode = new TreeNode(v);

rightNode.right = node.right;

node.right = rightNode;

}

if (node.left != null){

queue.offer(node.left);

}

if (node.right != null){

queue.offer(node.right);

}

}

if (level == d - 1){

break;

}

}

return root;

}

}

## 671\_SecondMinimumBinary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty special binary tree consisting of nodes with the non-negative value,

where each node in this tree has exactly two or zero sub-node. If the node has two sub-nodes,

then this node's value is the smaller value among its two sub-nodes.

Given such a binary tree, you need to output the second minimum value in the set made of

all the nodes' value in the whole tree.

If no such second minimum value exists, output -1 instead.

Example 1:

Input:

2

/ \

2 5

/ \

5 7

Output: 5

Explanation: The smallest value is 2, the second smallest value is 5.

Example 2:

Input:

2

/ \

2 2

Output: -1

Explanation: The smallest value is 2, but there isn't any second smallest value.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

long firstMin = Long.MAX\_VALUE;

long secondMin = Long.MAX\_VALUE;

public int findSecondMinimumValue(TreeNode root) {

dfs(root);

if (secondMin == Long.MAX\_VALUE){

return -1;

}

return (int) secondMin;

}

private void dfs(TreeNode root){

if (root == null){

return;

}

if (root.val < firstMin){

secondMin = firstMin;

firstMin = root.val;

}else if (root.val > firstMin && root.val < secondMin){

secondMin = root.val;

}

dfs(root.left);

dfs(root.right);

}

}

Better version:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int findSecondMinimumValue(TreeNode root) {

int[] res = new int[2];

res[0] = Integer.MAX\_VALUE;

res[1] = Integer.MAX\_VALUE;

preorder(root, res);

return res[1] == Integer.MAX\_VALUE ? -1 : res[1];

}

private void preorder(TreeNode root, int[] res){

if (root == null){

return;

}

if (root.val < res[0]){

res[1] = res[0];

res[0] = root.val;

}else if (root.val > res[0] && root.val < res[1]){

res[1] = root.val;

}

preorder(root.left, res);

preorder(root.right, res);

}

}

## 1008\_ConstructBinarySearchTreeFromPreorderTraversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Return the root node of a binary search tree that matches the given preorder traversal.

(Recall that a binary search tree is a binary tree where for every node, any descendant of node.left has a value < node.val, and any descendant of node.right has a value > node.val. Also recall that a preorder traversal displays the value of the node first, then traverses node.left, then traverses node.right.)

Example 1:

Input: [8,5,1,7,10,12]

Output: [8,5,10,1,7,null,12]

Note:

1 <= preorder.length <= 100

The values of preorder are distinct.

Check Leetcode 105: Construct Binary Tree from Preorder and Inorder Traversal

https://github.com/optimisea/Leetcode/blob/master/Java/105\_Construct.java

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode bstFromPreorder(int[] preorder) {

return dfs(0, preorder.length - 1, preorder);

}

private TreeNode dfs(int start, int end, int[] preorder){

if (start > end){

return null;

}

if (start == end){

return new TreeNode(preorder[start]);

}

int pivot = preorder[start];

TreeNode root = new TreeNode(pivot);

int index = findFirstLarge(pivot, preorder, start+1, end);

if (index != -1){

root.left = dfs(start+1, index-1, preorder);

root.right = dfs(index, end, preorder);

}else{//if there is larger one, then no right tree

root.left = dfs(start+1, end, preorder);

}

return root;

}

private int findFirstLarge(int num, int[] A, int start, int end){

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (A[mid] > num){

end = mid;

}else{

start = mid;

}

}

if (A[start] > num){

return start;

}else if (A[end] > num){

return end;

}

return -1;

}

}

## 1028\_Recover.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We run a preorder depth first search on the root of a binary tree.

At each node in this traversal, we output D dashes (where D is the depth of this node), then we output the value of this node. (If the depth of a node is D, the depth of its immediate child is D+1. The depth of the root node is 0.)

If a node has only one child, that child is guaranteed to be the left child.

Given the output S of this traversal, recover the tree and return its root.

Example 1:

Input: "1-2--3--4-5--6--7"

Output: [1,2,5,3,4,6,7]

Example 2:

Input: "1-2--3---4-5--6---7"

Output: [1,2,5,3,null,6,null,4,null,7]

Example 3:

Input: "1-401--349---90--88"

Output: [1,401,null,349,88,90]

Note:

The number of nodes in the original tree is between 1 and 1000.

Each node will have a value between 1 and 10^9.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode recoverFromPreorder(String S) {

if (S == null || S.length() == 0){

return null;

}

return build(S, 1);

}

private TreeNode build (String S, int depth){

if (S == null || S.length() == 0){

return null;

}

StringBuilder sb = new StringBuilder();

int count = 0;

while (depth > count){

sb.append('-');

count++;

}

String pattern = sb.toString();

int indexRoot = findIndex(S, pattern, 1, depth);

if (indexRoot < 0){

return new TreeNode(Integer.parseInt(S));

}

TreeNode root = new TreeNode(Integer.parseInt(S.substring(0, indexRoot)));

int indexLeft = findIndex(S, pattern, indexRoot + depth, depth);

if (indexLeft < 0){//only has left child

root.left = build(S.substring(indexRoot + depth), depth + 1);

return root;

}

root.left = build(S.substring(indexRoot + depth, indexLeft), depth + 1);

root.right = build(S.substring(indexLeft + depth), depth + 1);

return root;

}

private int findIndex(String S, String pattern, int start, int depth){

while (start < S.length() - depth){

char first = S.charAt(start-1);

char last = S.charAt(start + depth);

String sub = S.substring(start, start + depth);

if (Character.isDigit(first) && Character.isDigit(last) && sub.equals(pattern)){

return start;

}

start++;

}

return -1;

}

}

## 100\_SameTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two binary trees, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical and the nodes have the same value.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

if (p == null && q == null){

return true;

}

if (p == null && q != null){

return false;

}

if (p != null && q == null){

return false;

}

if (p.val != q.val){

return false;

}

return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);

}

}

## 101\_CheckBinaryTreeSymmetricMirror.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree [1,2,2,3,4,4,3] is symmetric:

1

/ \

2 2

/ \ / \

3 4 4 3

But the following [1,2,2,null,3,null,3] is not:

1

/ \

2 2

\ \

3 3

Note:

Bonus points if you could solve it both recursively and iteratively.

https://leetcode.com/problems/symmetric-tree/solution/

Recurive

public boolean isSymmetric(TreeNode root) {

return isMirror(root, root);

}

public boolean isMirror(TreeNode t1, TreeNode t2) {

if (t1 == null && t2 == null) return true;

if (t1 == null || t2 == null) return false;

return (t1.val == t2.val)

&& isMirror(t1.right, t2.left)

&& isMirror(t1.left, t2.right);

}

Iterative

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isSymmetric(TreeNode root) {

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

queue.offer(root);

while(!queue.isEmpty()){

TreeNode t1 = queue.poll();

TreeNode t2 = queue.poll();

if (t1 == null && t2 == null){

continue;

}

if (t1 == null || t2 == null){

return false;

}

if (t1.val != t2.val){

return false;

}

queue.add(t1.left);

queue.add(t2.right);

queue.add(t1.right);

queue.add(t2.left);

}

return true;

}

}

## 103\_BinaryTreeZigzagTraversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right,

then right to left for the next level and alternate between).

For example:

Given binary tree [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

return its zigzag level order traversal as:

[

[3],

[20,9],

[15,7]

]

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null){

return result;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

boolean leftToRight = false;

while (!queue.isEmpty()){

int size = queue.size();

List<Integer> list = new ArrayList<>();

leftToRight = !leftToRight;

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

if (leftToRight){

list.add(node.val);

}else{

list.add(0, node.val);

}

if (node.left != null){

queue.offer(node.left);

}

if (node.right != null){

queue.offer(node.right);

}

}

result.add(list);

}

return result;

}

}

class Solution {

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

if (root == null){

return res;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

boolean even = true;

while (!queue.isEmpty()){

int size = queue.size();

List<Integer> list = new ArrayList<>();

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

if (node.left != null){

queue.offer(node.left);

}

if (node.right != null){

queue.offer(node.right);

}

if (even){

list.add(node.val);

}else{

stack.push(node.val);

}

}

if (even){

res.add(list);

}else{

List<Integer> temp = new ArrayList<>();

while (!stack.isEmpty()){

temp.add(stack.pop());

}

res.add(temp);

}

even = !even;

}

return res;

}

}

## 104\_MaximumDepthBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int maxDepth(TreeNode root) {

if (root == null){

return 0;

}

int left = maxDepth(root.left);

int right = maxDepth(root.right);

return Math.max(left, right) + 1;

}

}

## 105\_ConstructBinaryTreeFromPreorderAndInorderTraversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

For example, given

preorder = [3,9,20,15,7]

inorder = [9,3,15,20,7]

Return the following binary tree:

3

/ \

9 20

/ \

15 7

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

Method 1: Recursion

Time complexity: O(nlogn)

class Solution {

public TreeNode buildTree(int[] preorder, int[] inorder) {

return build(preorder, 0, preorder.length - 1, inorder, 0, inorder.length - 1);

}

private TreeNode build(int[] preorder, int startP, int endP, int[] inorder, int startI, int endI){

if (startP > endP || startI > endI){

return null;

}

TreeNode root = new TreeNode(preorder[startP]);

int index = findIndex(inorder, startI, endI, preorder[startP]);

root.left = build(preorder, startP+1, startP+index, inorder, startI, startI+index-1);

root.right = build(preorder, startP+1+index, endP, inorder, startI+index+1, endI);

return root;

}

private int findIndex(int[] A, int start, int end, int target){

for (int i = start; i <= end; i++){

if (A[i] == target){

return i - start;//must use relative distance index

}

}

return -1;

}

}

Method 2: Recursion + HashMap

Time complexity: O(n)

class Solution {

public TreeNode buildTree(int[] preorder, int[] inorder) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < inorder.length; i++){

map.put(inorder[i], i);

}

return dfs(preorder, 0, preorder.length - 1, inorder, 0, inorder.length - 1, map);

}

private TreeNode dfs(int[] preorder, int pStart, int pEnd, int[] inorder, int iStart, int iEnd, Map<Integer, Integer> map){

if (pStart > pEnd || iStart > iEnd){

return null;

}

TreeNode root = new TreeNode(preorder[pStart]);

int deltaIndex = map.get(preorder[pStart]) - iStart;

root.left = dfs(preorder, pStart + 1, pStart + deltaIndex, inorder, iStart, iStart + deltaIndex - 1, map);

root.right = dfs(preorder, pStart + deltaIndex + 1, pEnd, inorder, iStart + deltaIndex + 1, iEnd, map);

return root;

}

}

## 106\_ConstructBinaryTreeFromInorderAndPostorderTraversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

For example, given

inorder = [9,3,15,20,7]

postorder = [9,15,7,20,3]

Return the following binary tree:

3

/ \

9 20

/ \

15 7

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

Method 1: Recursion

Time complexity: O(nlogn)

class Solution {

public TreeNode buildTree(int[] inorder, int[] postorder) {

return build(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1);

}

private TreeNode build(int[] inorder, int startI, int endI, int[] postorder, int startP, int endP){

if (startI > endI || startP > endP){

return null;

}

TreeNode root = new TreeNode(postorder[endP]);

int index = findIndex(inorder, startI, endI, postorder[endP]);

root.left = build(inorder, startI, startI+index-1, postorder, startP, startP+index-1);

root.right = build(inorder, startI+index+1, endI, postorder, startP+index, endP-1);

return root;

}

private int findIndex(int[] A, int start, int end, int target){

for (int i = start; i <= end; i++){

if (A[i] == target){

return i - start;

}

}

return -1;

}

}

Method 2: Recursion + HashMap

Time complexity: O(n)

class Solution {

public TreeNode buildTree(int[] inorder, int[] postorder) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < inorder.length; i++){

map.put(inorder[i], i);

}

return dfs(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1, map);

}

private TreeNode dfs(int[] inorder, int iStart, int iEnd, int[] postorder, int pStart, int pEnd, Map<Integer, Integer> map){

if (iStart > iEnd || pStart > pEnd){

return null;

}

TreeNode root = new TreeNode(postorder[pEnd]);

int deltaIndex = map.get(postorder[pEnd]) - iStart;

root.left = dfs(inorder, iStart, iStart + deltaIndex - 1, postorder, pStart, pStart + deltaIndex -1, map);

root.right = dfs(inorder, iStart + deltaIndex + 1, iEnd, postorder, pStart + deltaIndex, pEnd - 1, map);

return root;

}

}

class Solution {

Map<Integer, Integer> map = new HashMap<>();

public TreeNode buildTree(int[] inorder, int[] postorder) {

for (int i = 0 ;i < inorder.length; i++){

map.put(inorder[i], i);

}

return dfs(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1);

}

private TreeNode dfs(int[] inorder, int startI, int endI, int[] postorder, int startP, int endP){

if (startI > endI || startP > endP){

return null;

}

if (startI == endI){

return new TreeNode(inorder[startI]);

}

if (startP == endP){

return new TreeNode(postorder[startP]);

}

TreeNode root = new TreeNode(postorder[endP]);

int delta = map.get(postorder[endP]) - startI;

root.left = dfs(inorder, startI, startI + delta - 1, postorder, startP, startP + delta - 1);

root.right = dfs(inorder, startI + delta + 1, endI, postorder, startP + delta, endP - 1);

return root;

}

}

## 106\_ConvertSortedListtoBalancedBST.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Have you met this question in a real interview? Yes

Example

2

1->2->3 => / \

1 3

/\*\*

\* Definition for ListNode.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int val) {

\* this.val = val;

\* this.next = null;

\* }

\* }

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

public class Solution {

/\*

\* @param head: The first node of linked list.

\* @return: a tree node

\*/

public TreeNode sortedListToBST(ListNode head) {

if (head == null){

return null;

}

return helper(head, null);

}

private TreeNode helper(ListNode head, ListNode tail){

if (head == tail){

return null;

}

ListNode slow = head;

ListNode fast = head.next;

while (fast != tail && fast.next != tail){

fast = fast.next.next;

slow = slow.next;

}

TreeNode root = new TreeNode(slow.val);

root.left = helper(head, slow);

root.right = helper(slow.next, tail);

return root;

}

}

/\*\*

\* Definition for ListNode.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int val) {

\* this.val = val;

\* this.next = null;

\* }

\* }

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

public class Solution {

/\*

\* @param head: The first node of linked list.

\* @return: a tree node

\*/

public TreeNode sortedListToBST(ListNode head) {

if (head == null){

return null;

}

return build(head, null);

}

private TreeNode build(ListNode head, ListNode tail){

if (head == tail){

return null;

}

ListNode fast = head;

ListNode slow = head;

while (fast != tail && fast.next != tail){

fast = fast.next.next;

slow = slow.next;

}

TreeNode root = new TreeNode(slow.val);

root.left = build(head, slow);

root.right = build(slow.next, tail);

return root;

}

}

## 107\_BinaryTreeLevelOrderTraversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:

Given binary tree [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

return its bottom-up level order traversal as:

[

[15,7],

[9,20],

[3]

]

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<Integer>> levelOrderBottom(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null){

return result;

}

Queue<TreeNode> queue =new LinkedList<>();

queue.offer(root);

while(!queue.isEmpty()){

int size = queue.size();

List<Integer> item = new ArrayList<>();

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

item.add(node.val);

if (node.left != null){

queue.offer(node.left);

}

if (node.right != null){

queue.offer(node.right);

}

}

result.add(0, item);

}

return result;

}

}

Method 2: dfs

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<Integer>> levelOrderBottom(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

dfs(res, root, 0);

return res;

}

private void dfs(List<List<Integer>> res, TreeNode root, int height){

if (root == null){

return;

}

if (res.size() == height){

List<Integer> item = new ArrayList<>();

res.add(0, item);

}

dfs(res, root.left, height + 1);

dfs(res, root.right, height + 1);

res.get(res.size() - height - 1).add(root.val);

}

}

## 110\_BalancedBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which

the depth of the two subtrees of every node never differ by more than 1.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

class ResultType{

boolean isBal;

int depth;

public ResultType(boolean isBal, int depth){

this.isBal = isBal;

this.depth = depth;

}

}

public boolean isBalanced(TreeNode root) {

ResultType result = helper(root);

return result.isBal;

}

private ResultType helper(TreeNode root){

if (root == null){

return new ResultType(true, -1);

}

ResultType left = helper(root.left);

ResultType right = helper(root.right);

boolean bal;

if (left.isBal == true && right.isBal == true && Math.abs(left.depth - right.depth) <= 1){

bal = true;

}else{

bal = false;

}

return new ResultType(bal, Math.max(left.depth, right.depth) + 1);

}

}

Method 2:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

boolean res = true;

public boolean isBalanced(TreeNode root) {

if (root == null){

return true;

}

depth(root);

return res;

}

private int depth(TreeNode root){

if (root == null){

return 0;

}

int left = depth(root.left);

int right =depth(root.right);

if (Math.abs(left - right) > 1){

res = false;;

}

return Math.max(left, right) + 1;

}

}

Best solution:

class Solution {

public boolean isBalanced(TreeNode root) {

if (root == null){

return true;

}

boolean[] res = new boolean[1];

res[0] = true;

depth(root, res);

return res[0];

}

private int depth(TreeNode root, boolean[] res){

if (root == null){

return 0;

}

int left = depth(root.left, res);

int right =depth(root.right, res);

if (Math.abs(left - right) > 1){

res[0] = false;;

}

return Math.max(left, right) + 1;

}

}

## 111\_MinimumDepthofBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int minDepth(TreeNode root) {

if (root == null){

return 0;

}

if (root.left == null){

return minDepth(root.right) + 1;

}

if (root.right == null){

return minDepth(root.left) + 1;

}

int left = minDepth(root.left);

int right = minDepth(root.right);

return Math.min(left, right) + 1;

}

}

## 112\_PathSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values

along the path equals the given sum.

For example:

Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ \

7 2 1

return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean hasPathSum(TreeNode root, int sum) {

if (root == null){

return false;

}

if (root.left == null && root.right == null && root.val == sum){

return true;

}

return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);

}

}

## 113\_PathSumII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

iven a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:

Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ / \

7 2 5 1

return

[

[5,4,11,2],

[5,8,4,5]

]

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<Integer>> pathSum(TreeNode root, int sum) {

List<List<Integer>> result = new ArrayList<>();

if (root == null){

return result;

}

List<Integer> list = new ArrayList<Integer>();

list.add(root.val);

dfs(result, list, root, sum);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, TreeNode root, int sum){

if(root.left == null && root.right == null && root.val == sum){

result.add(new ArrayList<Integer>(item));

return;

}

if (root.left != null){

item.add(root.left.val);

dfs(result, item, root.left, sum - root.val);

item.remove(item.size() - 1);

}

if (root.right != null){

item.add(root.right.val);

dfs(result, item, root.right, sum - root.val);

item.remove(item.size() - 1);

}

}

}

Better version:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<Integer>> pathSum(TreeNode root, int sum) {

List<List<Integer>> res = new ArrayList<>();

dfs(res, new ArrayList<Integer>(), root, sum);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, TreeNode root, int sum){

if (root == null){

return;

}

if (root.val == sum && root.left == null && root.right == null){

item.add(root.val);

res.add(new ArrayList<>(item));

item.remove(item.size() - 1);

return;

}

item.add(root.val);

dfs(res, item, root.left, sum - root.val);

dfs(res, item, root.right, sum - root.val);

item.remove(item.size() - 1);

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<Integer>> pathSum(TreeNode root, int sum) {

List<List<Integer>> res = new ArrayList<>();

if (root == null){

return res;

}

dfs(res, new ArrayList<Integer>(), root, sum);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, TreeNode root, int sum){

if (root.val == sum && root.left == null && root.right == null){

item.add(root.val);

res.add(new ArrayList<>(item));

item.remove(item.size() - 1);

return;

}

item.add(root.val);

if (root.left != null){

dfs(res, item, root.left, sum - root.val);

}

if (root.right != null){

dfs(res, item, root.right, sum - root.val);

}

item.remove(item.size() - 1);

}

}

## 114\_FlattenBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, flatten it to a linked list in-place.

For example,

Given

1

/ \

2 5

/ \ \

3 4 6

The flattened tree should look like:

1

\

2

\

3

\

4

\

5

\

6

click to show hints.

Hints:

If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.

Best solution:

class Solution {

public void flatten(TreeNode root) {

if (root == null){

return;

}

Stack<TreeNode> stack = new Stack<>();

stack.push(root);

while (!stack.isEmpty()){

TreeNode curr = stack.pop();

if (curr.right != null){

stack.push(curr.right);

}

if (curr.left != null){

stack.push(curr.left);

}

if (!stack.isEmpty()){

curr.right = stack.peek();//note that curr.left is not necessary to be the next, but stack.peek() will be for sure.

curr.left = null;

}

}

}

}

Best solution: for recursion

Because the required traversal is preorder, we think/work from reverse preorder traversal

class Solution {

TreeNode lastNode = null;

public void flatten(TreeNode root) {

if (root == null){

return;

}

//reverse preorder

flatten(root.right);

flatten(root.left);

root.right = lastNode;

root.left = null;

lastNode = root;

}

}

Method 1: Divide && Conquer

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public void flatten(TreeNode root) {

getLastNode(root);

}

//flatten and get the last node

private TreeNode getLastNode(TreeNode root){

if (root == null){

return root;

}

TreeNode leftLastNode = getLastNode(root.left);

TreeNode rightLastNode = getLastNode(root.right);

if (leftLastNode != null){

leftLastNode.right = root.right;

root.right = root.left;

root.left = null;

}

if (rightLastNode != null){

return rightLastNode;

}

if (leftLastNode != null){

return leftLastNode;

}

return root;

}

}

Method 2: Preorder traversal

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

private TreeNode lastNode = null;

public void flatten(TreeNode root) {

if (root == null){

return;

}

if (lastNode != null){

lastNode.right = root;

lastNode.left = null;

}

lastNode = root;

TreeNode temp = root.right;

flatten(root.left);

flatten(temp);

}

}

With extra space and just the same as standard preorder traversal

Best solution:

class Solution {

public void flatten(TreeNode root) {

if (root == null){

return;

}

Stack<TreeNode> stack = new Stack<>();

stack.push(root);

while (!stack.isEmpty()){

TreeNode curr = stack.pop();

if (curr.right != null){

stack.push(curr.right);

}

if (curr.left != null){

stack.push(curr.left);

}

if (!stack.isEmpty()){

curr.right = stack.peek();

curr.left = null;

}

}

}

}

// helper function takes in the previous head, do the flattening and returns the head of the flatten binary tree

class Solution {

public void flatten(TreeNode root) {

root = reversePreorder(root, null);

}

private TreeNode reversePreorder(TreeNode root, TreeNode prev){

if (root == null){

return prev;

}

prev = reversePreorder(root.right, prev);//assign prev to the root.right

prev = reversePreorder(root.left, prev);//assign prev to the root.left

root.right = prev;

root.left = null;

// prev = root;

return root;

}

}

## 124\_MaximumPathSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node

in the tree along the parent-child connections. The path must contain at least one node and does not

need to go through the root.

For example:

Given the below binary tree,

1

/ \

2 3

Return 6.

Similar as 687. Longest Univalue Path, Refer to the solution in 687

Method 1: use global variable

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

int max = Integer.MIN\_VALUE;

public int maxPathSum(TreeNode root) {

maxPathSumIncludeRoot(root);

return max;

}

private int maxPathSumIncludeRoot(TreeNode root){

if (root == null){

return 0;

}

int left = maxPathSumIncludeRoot(root.left);

int right = maxPathSumIncludeRoot(root.right);

int leftMax = Math.max(left, 0);

int rightMax = Math.max(right, 0);

max = Math.max(max, leftMax + rightMax + root.val);

return root.val + Math.max(leftMax, rightMax);

}

}

class Solution {

int max = Integer.MIN\_VALUE;

public int maxPathSum(TreeNode root) {

maxIncludeRoot(root);

return max;

}

private int maxIncludeRoot(TreeNode root){

if (root == null){

return 0;

}

int left = maxIncludeRoot(root.left);

int right = maxIncludeRoot(root.right);

int leftMaxIncludeRoot = Math.max(left, 0);

int rightMaxIncluderoot = Math.max(right, 0);

max = Math.max(max, leftMaxIncludeRoot + rightMaxIncluderoot + root.val);

return Math.max(leftMaxIncludeRoot, rightMaxIncluderoot) + root.val;

}

}

Method 2: without using global variable

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int maxPathSum(TreeNode root) {

int[] max = new int[1];

max[0] = Integer.MIN\_VALUE;

maxPathSumIncludeRoot(root, max);

return max[0];

}

private int maxPathSumIncludeRoot(TreeNode root, int[] max){

if (root == null){

return 0;

}

int left = maxPathSumIncludeRoot(root.left, max);

int right = maxPathSumIncludeRoot(root.right, max);

int leftMax = Math.max(left, 0);

int rightMax = Math.max(right, 0);

max[0] = Math.max(max[0], leftMax + rightMax + root.val);

return root.val + Math.max(leftMax, rightMax);

}

}

## 144\_BinaryTreePreorderTraversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the preorder traversal of its nodes' values.

For example:

Given binary tree [1,null,2,3],

1

\

2

/

3

return [1,2,3].

Note: Recursive solution is trivial, could you do it iteratively?

Method 1: iteration

class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

if (root == null){

return result;

}

Stack<TreeNode> stack = new Stack<TreeNode>();

stack.push(root);

while(!stack.isEmpty()){

TreeNode node = stack.pop();

result.add(node.val);

if (node.right != null){

stack.push(node.right);

}

if (node.left != null){

stack.push(node.left);

}

}

return result;

}

}

Method 2:

Recursion(Traverse)

class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

ArrayList<Integer> result = new ArrayList<Integer>();

preorder(root, result);

return result;

}

private void preorder (TreeNode root, ArrayList<Integer> result){

if (root == null){

return;

}

result.add(root.val);

preorder(root.left, result);

preorder(root.right,result);

}

}

Method 3:

Recursion(divide & conquer)

class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

ArrayList<Integer> result = new ArrayList<Integer>();

if (root == null){

return result;

}

List<Integer> left = preorderTraversal(root.left);

List<Integer> right = preorderTraversal(root.right);

result.add(root.val);

result.addAll(left);

result.addAll(right);

return result;

}

}

Good practice for iteration template:

refer to https://github.com/optimisea/Leetcode/blob/master/Java/114\_FlattenBinaryTree.java

class Solution {

public void flatten(TreeNode root) {

if (root == null){

return;

}

Stack<TreeNode> stack = new Stack<>();

stack.push(root);

while (!stack.isEmpty()){

TreeNode curr = stack.pop();

if (curr.right != null){

stack.push(curr.right);

}

if (curr.left != null){

stack.push(curr.left);

}

if (!stack.isEmpty()){

curr.right = stack.peek();

curr.left = null;

}

}

}

}

## 145\_BinaryTreePostorderTraversa.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the postorder traversal of its nodes' values.

For example:

Given binary tree [1,null,2,3],

1

\

2

/

3

return [3,2,1].

Note: Recursive solution is trivial, could you do it iteratively?

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

dfs(root, result);

return result;

}

private void dfs(TreeNode root, List<Integer> result){

if (root == null){

return;

}

dfs(root.left, result);

dfs(root.right, result);

result.add(root.val);

}

}

Method 2: iteration: similar logic compared to preOrder iteration

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<>();

if (root == null){

return res;

}

Stack<TreeNode> stack = new Stack<>();

stack.push(root);

while (!stack.isEmpty()){

TreeNode node = stack.pop();

res.add(0, node.val);

if (node.left != null){

stack.push(node.left);

}

if (node.right != null){

stack.push(node.right);

}

}

return res;

}

}

## 156\_Binary Tree Upside Down.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node)

or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

Example:

Input: [1,2,3,4,5]

1

/ \

2 3

/ \

4 5

Output: return the root of the binary tree [4,5,2,#,#,3,1]

4

/ \

5 2

/ \

3 1

Similar as reverse LinkedList, this is reverse tree

recursion: bottom up

iteration: top down

Method 1: recursion with global variable

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

TreeNode newRoot = null;

public TreeNode upsideDownBinaryTree(TreeNode root) {

inOrder(root);

return newRoot;

}

private void inOrder(TreeNode root){

if (root == null){

return;

}

if (root.left == null && root.right == null){

if (newRoot == null){

newRoot = root;

}

return;

}

inOrder(root.left);

root.left.left = root.right;

root.left.right = root;

root.left = null;

root.right = null;

inOrder(root.right);

}

}

Method 2: recursion without global variable

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode upsideDownBinaryTree(TreeNode root) {

return inOrder(root);

}

private TreeNode inOrder(TreeNode root){

if (root == null || (root.left == null && root.right == null)){

return root;

}

TreeNode newRoot = inOrder(root.left);

root.left.left = root.right;

root.left.right = root;

root.left = null;

root.right = null;

return newRoot;

}

}

Best solution:

public TreeNode upsideDownBinaryTree(TreeNode root) {

if(root == null || root.left == null) {

return root;

}

TreeNode newRoot = upsideDownBinaryTree(root.left);

root.left.left = root.right; // node 2 left children

root.left.right = root; // node 2 right children

root.left = null;

root.right = null;

return newRoot;

}

Method 3: Iteration (best solution)

class Solution {

public TreeNode upsideDownBinaryTree(TreeNode root) {

if (root == null){

return root;

}

TreeNode prev = null;

TreeNode curr = root;

TreeNode next = null;

TreeNode temp = null;

while (curr != null){

next = curr.left;

curr.left = temp;

temp = curr.right;

curr.right = prev;

prev = curr;

curr = next;

}

return prev;

}

}

## 199\_BinaryTreeRightSideView.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, imagine yourself standing on the right side of it, return the values of

the nodes you can see ordered from top to bottom.

For example:

Given the following binary tree,

1 <---

/ \

2 3 <---

\ \

5 4 <---

You should return [1, 3, 4].

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> rightSideView(TreeNode root) {

List<Integer> result = new ArrayList<>();

if (root == null){

return result;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

if (i == 0){

result.add(node.val);

}

if (node.right != null){

queue.offer(node.right);

}

if (node.left != null){

queue.offer(node.left);

}

}

}

return result;

}

}

## 226\_InvertBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Invert a binary tree.

4

/ \

2 7

/ \ / \

1 3 6 9

to

4

/ \

7 2

/ \ / \

9 6 3 1

Method 1: recursive

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode invertTree(TreeNode root) {

if (root == null){

return root;

}

TreeNode temp = invertTree(root.left);

root.left = invertTree(root.right);

root.right = temp;

return root;

}

}

Method 2: iterative

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode invertTree(TreeNode root) {

if (root == null){

return root;

}

Queue<TreeNode> q = new LinkedList<>();

q.offer(root);

while (!q.isEmpty()){

TreeNode node = q.poll();

TreeNode temp = node.left;

node.left = node.right;

node.right = temp;

if (node.left != null){

q.offer(node.left);

}

if (node.right != null){

q.offer(node.right);

}

}

return root;

}

}

## 235\_LowestCommonAncestorBinarySearchTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v

and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).”

\_\_\_\_\_\_\_6\_\_\_\_\_\_

/ \

\_\_\_2\_\_ \_\_\_8\_\_

/ \ / \

0 \_4 7 9

/ \

3 5

For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes

2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

Method 1: recursion

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if (root == null){

return null;

}

if ((p.val <= root.val && root.val <= q.val) || (q.val <= root.val && root.val <= p.val )){

return root;

}

if (p.val < root.val && q.val < root.val){

return lowestCommonAncestor(root.left, p, q);

}

if (p.val > root.val && q.val > root.val){

return lowestCommonAncestor(root.right, p, q);

}

return null;

}

}

class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if (root == null || root == p || root == q){

return root;

}

if (root.val > p.val && root.val > q.val){

return lowestCommonAncestor(root.left, p, q);

}

if (root.val < p.val && root.val < q.val){

return lowestCommonAncestor(root.right, p, q);

}

return root;

}

}

Method 2: iteration

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if (root == null){

return null;

}

while(true){

if ((p.val <= root.val && root.val <= q.val) || (q.val <= root.val && root.val <= p.val )){

return root;

}

if (p.val < root.val && q.val < root.val){

root = root.left;

}

if (p.val > root.val && q.val > root.val){

root = root.right;

}

}

}

}

## 250\_CountNodesWithSameValue.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

For example:

Given binary tree,

5

/ \

1 5

/ \ \

5 5 5

return 4.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int countUnivalSubtrees(TreeNode root) {

if (root == null){

return 0;

}

int[] count = new int[1]; //serve similar as global variable

helper(root, count);

return count[0];

}

private boolean helper(TreeNode root, int[] count){

if (root == null){

return true;

}

boolean left = helper(root.left, count);

boolean right = helper(root.right, count);

if (left && right){

if (root.left != null && root.left.val != root.val || root.right != null && root.right.val != root.val){

return false;

}

count[0]++;

return true;

}

return false;

}

}

## 257\_BinaryTreePaths.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return all root-to-leaf paths.

For example, given the following binary tree:

1

/ \

2 3

\

5

All root-to-leaf paths are:

["1->2->5", "1->3"]

Best solution:

class Solution {

public List<String> binaryTreePaths(TreeNode root) {

List<String> result = new ArrayList<>();

if (root == null){

return result;

}

if (root.left == null && root.right == null){

result.add(Integer.toString(root.val));

return result;

}

List<String> left = binaryTreePaths(root.left);

List<String> right = binaryTreePaths(root.right);

for (String str : left){

result.add(root.val + "->" + str);

}

for (String str : right){

result.add(root.val + "->" + str);

}

return result;

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<String> binaryTreePaths(TreeNode root) {

List<String> res = new ArrayList<>();

if (root == null){

return res;

}

List<String> left = binaryTreePaths(root.left);

List<String> right = binaryTreePaths(root.right);

if (!left.isEmpty()){

for (String s : left){

res.add(root.val + "->" + s);

}

}

if (!right.isEmpty()){

for (String s : right){

res.add(root.val + "->" + s);

}

}

if (left.isEmpty() && right.isEmpty()){

res.add(String.valueOf(root.val));

}

return res;

}

}

## 285\_InorderSuccessorInBST.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

Note: If the given node has no in-order successor in the tree, return null.

Example 1:

Input: root = [2,1,3], p = 1

2

/ \

1 3

Output: 2

Example 2:

Input: root = [5,3,6,2,4,null,null,1], p = 6

5

/ \

3 6

/ \

2 4

/

1

Output: null

Method 1: tree traversal

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

List<TreeNode> list = new ArrayList<TreeNode>();

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

inOrder(root, list);

for (int i = 0; i < list.size(); i++){

TreeNode node = list.get(i);

if (node.val == p.val){

if (i == list.size() - 1){

return null;

}else{

return list.get(i+1);

}

}

}

return null;

}

private void inOrder(TreeNode root, List<TreeNode> list){

if (root == null){

return;

}

inOrder(root.left, list);

list.add(root);

inOrder(root.right, list);

}

}

Method 2: divide conquer

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

if (root == null){

return null;

}

if (p.val >= root.val){

return inorderSuccessor(root.right, p);

}

TreeNode left = inorderSuccessor(root.left, p);

return left == null ? root : left;

}

}

For Predecessor

class Solution {

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

if (root == null){

return null;

}

if (p.val <= root.val){

return inorderSuccessor(root.left, p);

}

TreeNode right = inorderSuccessor(root.right, p);

return right == null ? root : right;

}

}

## 272\_ClosestBinarySearchTreeValueII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.

Note:

Given target value is a floating point.

You may assume k is always valid, that is: k ≤ total nodes.

You are guaranteed to have only one unique set of k values in the BST that are closest to the target.

Example:

Input: root = [4,2,5,1,3], target = 3.714286, and k = 2

4

/ \

2 5

/ \

1 3

Output: [4,3]

Follow up:

Assume that the BST is balanced, could you solve it in less than O(n) runtime (where n = total nodes)?

Answer: binary search O(h)

Method 1:

O(n)

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> closestKValues(TreeNode root, double target, int k) {

Queue<Integer> result = new LinkedList<>();

inOrder(result, root, target, k);

List<Integer> res = new ArrayList<>();

for (int : result){

res.add(int);

}

return res;

}

private void inOrder(LinkedList<Integer> result, TreeNode root, double target, int k){

if (root == null){

return;

}

inOrder(result, root.left, target, k);

if (result.size() == k){

if (Math.abs(result.peek() - target) <= Math.abs(root.val - target)){

return;

}else{

result.poll();

}

}

result.offer(root.val);

inOrder(result, root.right, target, k);

}

}

Best solution:

Method 2: BST inorder traversal

public List<Integer> closestKValues(TreeNode root, double target, int k){

List<Integer> res = new ArrayList<>();

Stack<TreeNode> stack = new Stack<>();

TreeNode node = root;

Queue<Integer> queue = new LinkedList<>();

while (node != null || !stack.isEmpty()){

while (node != null){

stack.push(node);

node = node.left;

}

TreeNode curr = stack.pop();

if (queue.size() < k){

queue.offer(curr.val);

}else{

if (Math.abs(queue.peek().val - target) > Math.abs(curr.val - target)){

queue.poll();

queue.offer(curr.val);

}else{

break;

}

}

node = curr.right;

}

for (int : queue){

res.add(int);

}

return res;

}

## 297\_SerializeandDeserializeBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored

in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or

another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your

serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized

to a string and this string can be deserialized to the original tree structure.

For example, you may serialize the following tree

1

/ \

2 3

/ \

4 5

as "[1,2,3,null,null,4,5]", just the same as how LeetCode OJ serializes a binary tree. You do not necessarily need

to follow this format, so please be creative and come up with different approaches yourself.

Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms

should be stateless.

Method 1:

Level order traversal:

the code string will be 1,2,3,null,null,4,5,null,null,null,null

Note that: linkelist/arraylist can add null

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Codec {

// Encodes a tree to a single string.

public String serialize(TreeNode root) {

StringBuilder ans = new StringBuilder();

if (root == null){

return "";

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

TreeNode node = queue.poll();

if (node != null){

ans.append(String.valueOf(node.val) + ",");

queue.offer(node.left);

queue.offer(node.right);

}else{

ans.append("null,");

}

}

return ans.toString();

}

// Decodes your encoded data to tree.

public TreeNode deserialize(String data) {

if (data.equals("")){

return null;

}

String[] strs = data.split(",");

Queue<TreeNode> queue = new LinkedList<>();

TreeNode root = new TreeNode(Integer.parseInt(strs[0]));

queue.offer(root);

for (int i = 1; i < strs.length; i++){

TreeNode parent = queue.poll();

if (!strs[i].equals("null")){

int num = Integer.parseInt(strs[i]);

TreeNode left = new TreeNode(num);

parent.left = left;

queue.offer(left);

}

if (!strs[++i].equals("null")){

int num = Integer.parseInt(strs[i]);

TreeNode right = new TreeNode(num);

parent.right = right;

queue.offer(right);

}

}

return root;

}

}

// Your Codec object will be instantiated and called as such:

// Codec codec = new Codec();

// codec.deserialize(codec.serialize(root));

Method 2:

preorder traversal

the order string will be 1,2,null,null,3,4,null,null,5,null,null

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Codec {

// Encodes a tree to a single string.

public String serialize(TreeNode root) {

if (root == null){

return "";

}

StringBuilder ans = new StringBuilder();

dfsSe(root, ans);

return ans.toString();

}

private void dfsSe(TreeNode root, StringBuilder ans){

if (root == null){

ans.append("null,");

return;

}

ans.append(root.val + ",");

dfsSe(root.left, ans);

dfsSe(root.right, ans);

}

// Decodes your encoded data to tree.

public TreeNode deserialize(String data) {

if (data.equals("")){

return null;

}

String[] strs = data.split(",");

int[] d = new int[1];

return dfsDe(strs, d);

}

private TreeNode dfsDe(String[] strs, int[] d){

if (strs[d[0]].equals("null")){

d[0]++;

return null;

}

TreeNode root = new TreeNode(Integer.parseInt(strs[d[0]]));

d[0]++;

root.left = dfsDe(strs, d);

root.right = dfsDe(strs, d);

return root;

}

}

// Your Codec object will be instantiated and called as such:

// Codec codec = new Codec();

// codec.deserialize(codec.serialize(root));

Method 3:

Best solution: preOrder with queue

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Codec {

// Encodes a tree to a single string.

public String serialize(TreeNode root) {

if (root == null){

return "";

}

StringBuilder sb = new StringBuilder();

serializePreOrder(root, sb);

return sb.toString();

}

private void serializePreOrder(TreeNode root, StringBuilder sb){

if (root == null){

sb.append("null,");

return;

}

sb.append(root.val + ",");

serializePreOrder(root.left, sb);

serializePreOrder(root.right, sb);

}

// Decodes your encoded data to tree.

public TreeNode deserialize(String data) {

if (data.length() == 0){

return null;

}

String[] strs = data.split(",");

Queue<String> queue = new LinkedList<>(Arrays.asList(strs));

return deserializePreOrder(queue);

}

private TreeNode deserializePreOrder(Queue<String> queue){

String str = queue.poll();

if (str.equals("null")){

return null;

}

TreeNode root = new TreeNode(Integer.parseInt(str));

root.left = deserializePreOrder(queue);

root.right = deserializePreOrder(queue);

return root;

}

}

// Your Codec object will be instantiated and called as such:

// Codec codec = new Codec();

// codec.deserialize(codec.serialize(root));

Best version: preorder

Using preOrder traversal could be generalized to all the similar serialize and deserialize tree question, including BST, n-ary tree.

Level Order can't do the generalization.

Note that in deseralization, you have to create int[] pt in order to track the pointer and move forward. Because in Java function,

it is passed by copying value or reference. It won't work if copying value. Hence, we have to construct int[] to enable copying by

reference.

```

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Codec {

// Encodes a tree to a single string.

public String serialize(TreeNode root) {

StringBuilder res = new StringBuilder();

preOrderSerial(root, res);

return res.toString();

}

private void preOrderSerial(TreeNode root, StringBuilder res){

if (root == null){

res.append("null,");

return;

}

res.append(root.val + ",");

preOrderSerial(root.left, res);

preOrderSerial(root.right, res);

}

// Decodes your encoded data to tree.

public TreeNode deserialize(String data) {

String[] strs = data.split(",");

int[] pt = new int[1];

pt[0] = 0;

return preOrderDeserial(strs, pt);

}

private TreeNode preOrderDeserial(String[] strs, int[] pt){

String str = strs[pt[0]];

pt[0]++;

if (str.equals("null")){

return null;

}

TreeNode root = new TreeNode(Integer.parseInt(str));

root.left = preOrderDeserial(strs, pt);

root.right = preOrderDeserial(strs, pt);

return root;

}

}

// Your Codec object will be instantiated and called as such:

// Codec codec = new Codec();

// codec.deserialize(codec.serialize(root));

```

## 331\_VerifyPreorderSerializationOfaBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #.

\_9\_

/ \

3 2

/ \ / \

4 1 # 6

/ \ / \ / \

# # # # # #

For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where # represents a null node.

Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.

Each comma separated value in the string must be either an integer or a character '#' representing null pointer.

You may assume that the input format is always valid, for example it could never contain two consecutive commas such as "1,,3".

Example 1:

"9,3,4,#,#,1,#,#,2,#,6,#,#"

Return true

Example 2:

"1,#"

Return false

Example 3:

"9,#,#,1"

Return false

when you see two consecutive “#” characters on stack, pop both of them and replace the topmost element on the stack with “#”.

The basic intuition was to collapse the entire tree into the root node.

Method 1: stack

class Solution {

public boolean isValidSerialization(String preorder) {

Stack<String> stack = new Stack<>();

String[] strs = preorder.split(",");

for (String str : strs){

while (str.equals("#") && !stack.isEmpty() && stack.peek().equals("#")){

stack.pop();

if (stack.isEmpty()){

return false;

}

stack.pop();

}

stack.push(str);

}

return stack.size() == 1 && stack.peek().equals("#");

}

}

Method 2: Best solution

https://leetcode.com/problems/verify-preorder-serialization-of-a-binary-tree/discuss/78551/7-lines-Easy-Java-Solution

in a tree, for all the edges, the number of indegree = the number of outdegree

For a valid tree, (number of "#") = (number of nodes) + 1

class Solution {

public boolean isValidSerialization(String preorder) {

String[] strs = preorder.split(",");

int diff = 1;//diff = outdegree - indegree

for (String str : strs){

diff--;

if (diff < 0){

return false;

}

if (!str.equals("#")){

diff += 2;

}

}

return diff == 0;

}

}

Method 3:

If we treat null's as leaves, then the binary tree will always be full. A full binary tree has a good property

that # of leaves = # of nonleaves + 1. Since we are given a pre-order serialization, we just need to find the

shortest prefix of the serialization sequence satisfying the property above. If such prefix does not exist,

then the serialization is definitely invalid; otherwise, the serialization is valid if and only if the

prefix is the entire sequence.

class Solution {

public boolean isValidSerialization(String preorder) {

int leaves = 0;

int nonLeaves = 0;

String[] strs = preorder.split(",");

for (String str : strs){

if (leaves == nonLeaves + 1){

return false;

}

if (str.equals("#")){

leaves++;

}else{

nonLeaves++;

}

}

return leaves == nonLeaves + 1;

}

}

## 333\_LargestBSTSubTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest

number of nodes in it.

Note:

A subtree must include all of its descendants.

Here's an example:

10

/ \

5 15

/ \ \

1 8 7

The Largest BST Subtree in this case is the highlighted one.

The return value is the subtree's size, which is 3.

Follow up:

Can you figure out ways to solve it with O(n) time complexity?

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

class Result {

int size;

int lower;

int upper;

public Result (int size, int lower, int upper){

this.size = size;

this.upper = upper;

this.lower = lower;

}

}

int max = -1;

public int largestBSTSubtree(TreeNode root) {

if (root == null){

return 0;

}

postOrder(root);

return max;

}

private Result postOrder(TreeNode root){

if (root == null){

return new Result(0, Integer.MAX\_VALUE, Integer.MIN\_VALUE);

}

Result left = postOrder(root.left);

Result right = postOrder(root.right);

if (left.size == -1 || right.size == -1 || root.val <= left.upper || root.val >= right.lower){

return new Result(-1, 0, 0);

}

int size = left.size + right.size + 1;

max = Math.max(max, size);

return new Result(size, Math.min(left.lower, root.val), Math.max(right.upper, root.val));

}

}

## 404\_SumOfAllLeftLeaves.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the sum of all left leaves in a given binary tree.

Example:

3

/ \

9 20

/ \

15 7

There are two left leaves in the binary tree, with values 9 and 15 respectively. Return 24.

Method 1: tree traversal

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

int sum;

public int sumOfLeftLeaves(TreeNode root) {

dfs(root, root);

return sum;

}

private void dfs(TreeNode root, TreeNode parent){

if (root == null){

return;

}

if (root == parent.left && root.left == null && root.right == null){

sum += root.val;

}

dfs(root.left, root);

dfs(root.right, root);

}

}

class Solution {

int sum = 0;

public int sumOfLeftLeaves(TreeNode root) {

dfs(root);

return sum;

}

private void dfs(TreeNode root){

if (root == null){

return;

}

if (root.left != null && root.left.left == null && root.left.right == null){

sum += root.left.val;

}

dfs(root.left);

dfs(root.right);

}

}

Method 2: divide and conquer

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int sumOfLeftLeaves(TreeNode root) {

if (root == null){

return 0;

}

int sum = 0;

if (root.left != null && root.left.left == null && root.left.right == null){

sum += root.left.val;

}

sum += sumOfLeftLeaves(root.left);

sum += sumOfLeftLeaves(root.right);

return sum;

}

}

Method 3:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int sumOfLeftLeaves(TreeNode root) {

if (root == null){

return 0;

}

int sum = 0;

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

TreeNode curr = queue.poll();

if (curr.left != null){

queue.offer(curr.left);

if (curr.left.left == null && curr.left.right == null){

sum += curr.left.val;

}

}

if (curr.right != null){

queue.offer(curr.right);

}

}

return sum;

}

}

## 726\_CheckFullBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes.

Conversely, there is no node in a full binary tree, which has one child node. More information about

full binary trees can be found here.

Full Binary Tree

1

/ \

2 3

/ \

4 5

Not a Full Binary Tree

1

/ \

2 3

/

4

Have you met this question in a real interview? Yes

Example

Given tree {1,2,3}, return true

Given tree {1,2,3,4}, return false

Given tree {1,2,3,4,5} return true

public class Solution {

/\*

\* @param : the given tree

\* @return: Whether it is a full tree

\*/

public boolean isFullTree(TreeNode root) {

if (root == null){

return true;

}

if (root.left == null && root.right != null || root.right == null && root.left != null){

return false;

}

boolean left = isFullTree(root.left);

boolean right = isFullTree(root.right);

if (left && right){

return true;

}

return false;

}

}

# Graph

## 133\_CloneGraph.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

OJ's undirected graph serialization:

Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.

As an example, consider the serialized graph {0,1,2#1,2#2,2}.

The graph has a total of three nodes, and therefore contains three parts as separated by #.

First node is labeled as 0. Connect node 0 to both nodes 1 and 2.

Second node is labeled as 1. Connect node 1 to node 2.

Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.

Visually, the graph looks like the following:

1

/ \

/ \

0 --- 2

/ \

\\_/

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\* int label;

\* ArrayList<UndirectedGraphNode> neighbors;

\* UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

/\*

\* @param node: A undirected graph node

\* @return: A undirected graph node

\*/

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null){

return node;

}

//1. node -> all nodes

ArrayList<UndirectedGraphNode> nodes = getAllNodes(node);

//2. nodes -> new nodes

Map<UndirectedGraphNode, UndirectedGraphNode> mapping = new HashMap<>();

for (UndirectedGraphNode n : nodes){

mapping.put(n, new UndirectedGraphNode(n.label));

}

//3. edges -> new edges

for (UndirectedGraphNode n: nodes){

UndirectedGraphNode newNode = mapping.get(n);

for(UndirectedGraphNode bor : n.neighbors){

UndirectedGraphNode newbor = mapping.get(bor);

newNode.neighbors.add(newbor);

}

}

return mapping.get(node);

}

private ArrayList<UndirectedGraphNode> getAllNodes(UndirectedGraphNode node){

Queue<UndirectedGraphNode> queue = new LinkedList<>();

//Set<UndirectedGraphNode> hash = new HashSet<>();

ArrayList<UndirectedGraphNode> hash = new ArrayList<>();

queue.add(node);

hash.add(node);

while (!queue.isEmpty()){

UndirectedGraphNode head = queue.poll();

for (UndirectedGraphNode n : head.neighbors){

if (!hash.contains(n)){

hash.add(n);

queue.add(n);

}

}

}

//return new ArrayList<UndirectedGraphNode>(hash);

return hash;

}

}

Better version: BFS

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\* int label;

\* List<UndirectedGraphNode> neighbors;

\* UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null){

return null;

}

Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();

Queue<UndirectedGraphNode> queue = new LinkedList<>();

queue.offer(node);

map.put(node, new UndirectedGraphNode(node.label));

while (!queue.isEmpty()){

UndirectedGraphNode curr = queue.poll();

for (UndirectedGraphNode nei : curr.neighbors){

if (!map.containsKey(nei)){

map.put(nei, new UndirectedGraphNode(nei.label));

queue.offer(nei);//only new node will go on for BFS

}

map.get(curr).neighbors.add(map.get(nei));

}

}

return map.get(node);

}

}

public class Solution {

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null){

return null;

}

Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();

UndirectedGraphNode root = new UndirectedGraphNode(node.label);

map.put(node, root);

dfs(node, map);

return root;

}

private void dfs(UndirectedGraphNode node, Map<UndirectedGraphNode, UndirectedGraphNode> map){

if (node == null){

return;

}

for (UndirectedGraphNode nei : node.neighbors){

if (!map.containsKey(nei)){

UndirectedGraphNode newNode = new UndirectedGraphNode(nei.label);

map.put(nei, newNode);

map.get(node).neighbors.add(newNode);

dfs(nei, map);//only new node will go on for dfs

}else{

map.get(node).neighbors.add(map.get(nei));

}

}

}

}

Better version: DFS

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\* int label;

\* List<UndirectedGraphNode> neighbors;

\* UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null){

return null;

}

UndirectedGraphNode curr = new UndirectedGraphNode(node.label);

map.put(node, curr);

for (UndirectedGraphNode nei : node.neighbors){

if (!map.containsKey(nei)){

cloneGraph(nei);

}

curr.neighbors.add(map.get(nei));

}

return curr;

}

}

After updating new API

/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> neighbors;

public Node() {}

public Node(int \_val,List<Node> \_neighbors) {

val = \_val;

neighbors = \_neighbors;

}

};

\*/

class Solution {

Map<Node, Node> map = new HashMap<>();

public Node cloneGraph(Node node) {

if (node == null){

return null;

}

Node clone = new Node(node.val, new ArrayList<>());

map.put(node, clone);

for (Node nei : node.neighbors){

if (!map.containsKey(nei)){

cloneGraph(nei);

}

clone.neighbors.add(map.get(nei));

}

return clone;

}

}

Without global variable:

class Solution {

public Node cloneGraph(Node node) {

if (node == null){

return null;

}

Map<Node, Node> map = new HashMap<>();

dfs(node, map);

return map.get(node);

}

private void dfs(Node node, Map<Node, Node> map){

if (node == null){

return;

}

Node clone = new Node(node.val, new ArrayList<>());

map.put(node, clone);

for (Node nei : node.neighbors){

if (!map.containsKey(nei)){

dfs(nei, map);

}

clone.neighbors.add(map.get(nei));

}

}

}

## 178\_GraphValidTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether

these edges make up a valid tree.

Notice

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is

the same as [1, 0] and thus will not appear together in edges.

这道题给了我们一个无向图，让我们来判断其是否为一棵树，我们知道如果是树的话，所有的节点必须是连接的，也就是说必须是连通图，而且不能有环，

所以我们的焦点就变成了验证是否是连通图和是否含有环。

public class Solution {

/\*

\* @param n: An integer

\* @param edges: a list of undirected edges

\* @return: true if it's a valid tree, or false

\*/

public boolean validTree(int n, int[][] edges) {

if (n <= 0){

return false;

}

if (n != edges.length + 1){//check if it has circle

return false;

}

Map<Integer, Set<Integer>> graph = initializeGraph(n, edges);

Set<Integer> set = new HashSet<>();

Queue<Integer> queue = new LinkedList<>();

queue.add(0);

set.add(0);

while (!queue.isEmpty()){

int cur = queue.poll();

for (Integer neigbor : graph.get(cur)){

if (set.contains(neigbor)){

continue;

}

set.add(neigbor);

queue.offer(neigbor);

}

}

if (set.size() == n){

return true;

}

return false;

}

private static Map<Integer, Set<Integer>> initializeGraph(int n, int[][] edges){

Map<Integer, Set<Integer>> graph = new HashMap<>();

for (int i = 0; i < n; i++){

graph.put(i, new HashSet<Integer>());

}

for (int i = 0; i < edges.length; i++){

int u = edges[i][0];

int v = edges[i][1];

graph.get(u).add(v);

graph.get(v).add(u);

}

return graph;

}

}

Method 2: Union Find

class Solution {

class UF{

int[] id;

int[] size;

int count;

public UF(int N){

count = N;

id = new int[N];

size = new int[N];

for (int i = 0; i < N; i++){

id[i] = i;

size[i] = 1;

}

}

public int componentNum(){

return count;

}

public int root(int p){

while (p != id[p]){

p = id[p];

}

return p;

}

public boolean find(int p , int q){

return root(p) == root(q);

}

public void union(int p, int q){

int rp = root(p);

int rq = root(q);

if (rp < rq){

id[rp] = rq;

size[rq] += size[rp];

}else{

id[rq] = rp;

size[rp] += size[rq];

}

count--;

}

}

public boolean validTree(int n, int[][] edges) {

UF uf = new UF(n);

for (int[] edge : edges){

if (uf.find(edge[0], edge[1])){

return false;

}

uf.union(edge[0], edge[1]);

}

return uf.componentNum() == 1;

}

}

https://leetcode.com/problems/graph-valid-tree/discuss/69018/AC-Java-Union-Find-solution

## 332\_ReconstructItinerary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order. All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.

Note:

If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].

All airports are represented by three capital letters (IATA code).

You may assume all tickets form at least one valid itinerary.

Example 1:

tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]

Return ["JFK", "MUC", "LHR", "SFO", "SJC"].

Example 2:

tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]

Return ["JFK","ATL","JFK","SFO","ATL","SFO"].

Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"]. But it is larger in lexical order.

class Solution {

public List<String> findItinerary(String[][] tickets) {

List<String> result = new ArrayList<>();

int steps = tickets.length;

Map<String, Queue<String>> graph = new HashMap<>();

for (int i = 0; i < steps; i++){

if (!graph.containsKey(tickets[i][0])){

graph.put(tickets[i][0], new PriorityQueue<String>());

}

graph.get(tickets[i][0]).add(tickets[i][1]);

}

dfs(result, graph, "JFK");

return result;

}

private void dfs(List<String> result, Map<String, Queue<String>> graph, String start){

// result.add(start);

Queue<String> pq = graph.get(start);

while (pq != null && !pq.isEmpty()){

String str = pq.poll();

dfs(result, graph, str);

}

result.add(0, start);

}

}

# DesignQuestions

## 146\_LRUCache.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

Follow up:

Could you do both operations in O(1) time complexity?

Example:

LRUCache cache = new LRUCache( 2 /\* capacity \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // returns 1

cache.put(3, 3); // evicts key 2

cache.get(2); // returns -1 (not found)

cache.put(4, 4); // evicts key 1

cache.get(1); // returns -1 (not found)

cache.get(3); // returns 3

cache.get(4); // returns 4

Better solution:

Code that was written in the 2nd time

class LRUCache {

class LinkedNode {

int val;

int key;

LinkedNode prev;

LinkedNode next;

public LinkedNode(int key, int val){

this.key = key;

this.val = val;

prev = null;

next = null;

}

}

Map<Integer, LinkedNode> map;

LinkedNode head;

LinkedNode tail;

int capacity;

public LRUCache(int capacity) {

this.capacity = capacity;

map = new HashMap<>();

head = new LinkedNode(-1, -1);

tail = new LinkedNode(-1, -1);

head.next = tail;

tail.prev = head;

}

public int get(int key) {

if (map.containsKey(key)){

LinkedNode node = map.get(key);

prepareToMove(node);

moveToTail(node);

return node.val;

}

return -1;

}

public void put(int key, int value) {

if (map.containsKey(key)){

LinkedNode node = map.get(key);

node.val = value;

prepareToMove(node);

moveToTail(node);

}else{

if (map.size() == capacity){

LinkedNode after = head.next;

after.next.prev = head;

head.next = after.next;

after.prev = null;

after.next = null;

map.remove(after.key);

}

LinkedNode node = new LinkedNode(key, value);

moveToTail(node);

map.put(key, node);

}

}

private void moveToTail(LinkedNode node){

LinkedNode before = tail.prev;

before.next = node;

node.prev = before;

node.next = tail;

tail.prev = node;

}

private void prepareToMove(LinkedNode node){

node.prev.next = node.next;

node.next.prev = node.prev;

node.prev = null;

node.next = null;

}

}

/\*\*

\* Your LRUCache object will be instantiated and called as such:

\* LRUCache obj = new LRUCache(capacity);

\* int param\_1 = obj.get(key);

\* obj.put(key,value);

\*/

class LRUCache {

class Node{

int key;

int val;

Node next;

Node prev;

public Node(int key, int val){

this.key = key;

this.val = val;

}

}

private Map<Integer, Node> map;

private int capacity;

private Node head;

private Node tail;

public LRUCache(int capacity) {

this.map = new HashMap<>();

this.capacity = capacity;

this.head = new Node(-1, -1);

this.tail = new Node(-1, -1);

head.next = tail;

tail.prev = head;

}

public int get(int key) {

if (!map.containsKey(key)){

return -1;

}

Node cur = map.get(key);

cur.next.prev = cur.prev;

cur.prev.next = cur.next;

cur.prev = null;

cur.next = null;

moveToTail(cur);

return map.get(key).val;

}

public void put(int key, int value) {

if (!map.containsKey(key)){

Node insert = new Node(key, value);

map.put(key, insert);

if (capacity > 0){

capacity--;

}else{

map.remove(head.next.key);

head.next = head.next.next;

head.next.prev = head;

}

moveToTail(insert);

}else{

int v = get(key);

map.get(key).val = value;

}

}

private void moveToTail(Node cur){

cur.next = tail;

tail.prev.next = cur;

cur.prev = tail.prev;

tail.prev = cur;

}

}

/\*\*

\* Your LRUCache object will be instantiated and called as such:

\* LRUCache obj = new LRUCache(capacity);

\* int param\_1 = obj.get(key);

\* obj.put(key,value);

\*/

## 155\_MinStack.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

push(x) -- Push element x onto stack.

pop() -- Removes the element on top of the stack.

top() -- Get the top element.

getMin() -- Retrieve the minimum element in the stack.

Example:

MinStack minStack = new MinStack();

minStack.push(-2);

minStack.push(0);

minStack.push(-3);

minStack.getMin(); --> Returns -3.

minStack.pop();

minStack.top(); --> Returns 0.

minStack.getMin(); --> Returns -2.

Best solution

class MinStack {

Stack<Integer> stack;

int min;

/\*\* initialize your data structure here. \*/

public MinStack() {

stack = new Stack<>();

min = Integer.MAX\_VALUE;

}

public void push(int x) {

if (x <= min){ // must be <= , can't be <

stack.push(min); //save as backup

stack.push(x);

min = x;

}else{

stack.push(x);

}

}

public void pop() {

int num = stack.pop();

if (num == min){

min = stack.pop();

}

}

public int top() {

return stack.peek();

}

public int getMin() {

return min;

}

}

/\*\*

\* Your MinStack object will be instantiated and called as such:

\* MinStack obj = new MinStack();

\* obj.push(x);

\* obj.pop();

\* int param\_3 = obj.top();

\* int param\_4 = obj.getMin();

\*/

Method 2:

class MinStack {

Stack<Integer> stack;

Stack<Integer> minStack;

/\*\* initialize your data structure here. \*/

public MinStack() {

stack = new Stack<Integer>();

minStack = new Stack<>();

}

public void push(int x) {

stack.push(x);

if (minStack.isEmpty()){

minStack.push(x);

}else{

minStack.push(Math.min(minStack.peek(), x));

}

}

public void pop() {

minStack.pop();

stack.pop();

}

public int top() {

return stack.peek();

}

public int getMin() {

return minStack.peek();

}

}

Method 3:

push x and min at one stack

class MinStack {

Stack<Integer> stack;

int min;

/\*\* initialize your data structure here. \*/

public MinStack() {

stack = new Stack<>();

min = Integer.MAX\_VALUE;

}

public void push(int x) {

min = Math.min(min, x);

stack.push(x);

stack.push(min);

}

public void pop() {

stack.pop();

int temp = stack.pop();

if (min == temp){

if (stack.isEmpty()){

min = Integer.MAX\_VALUE;

}else{

min = stack.peek();

}

}

}

public int top() {

int temp = stack.pop();

int val = stack.peek();

stack.push(temp);

return val;

}

public int getMin() {

return stack.peek();

}

}

/\*\*

\* Your MinStack object will be instantiated and called as such:

\* MinStack obj = new MinStack();

\* obj.push(x);

\* obj.pop();

\* int param\_3 = obj.top();

\* int param\_4 = obj.getMin();

\*/

## 170\_Two Sum III - Data structure design.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design and implement a TwoSum class. It should support the following operations: add and find.

add - Add the number to an internal data structure.

find - Find if there exists any pair of numbers which sum is equal to the value.

Example 1:

add(1); add(3); add(5);

find(4) -> true

find(7) -> false

Example 2:

add(3); add(1); add(2);

find(3) -> true

find(6) -> false

class TwoSum {

private Map<Integer, Integer> map;

/\*\* Initialize your data structure here. \*/

public TwoSum() {

map = new HashMap<Integer, Integer>();

}

/\*\* Add the number to an internal data structure.. \*/

public void add(int number) {

map.put(number, map.getOrDefault(number, 0) + 1);

}

/\*\* Find if there exists any pair of numbers which sum is equal to the value. \*/

public boolean find(int value) {

for (int i : map.keySet()){

int j = value - i;

if (i != j && map.containsKey(j) || i == j && map.get(i) > 1){

return true;

}

}

return false;

}

}

/\*\*

\* Your TwoSum object will be instantiated and called as such:

\* TwoSum obj = new TwoSum();

\* obj.add(number);

\* boolean param\_2 = obj.find(value);

\*/

## 208\_Trie.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement a trie with insert, search, and startsWith methods.

Note:

You may assume that all inputs are consist of lowercase letters a-z.

https://leetcode.com/problems/implement-trie-prefix-tree/solution/

class Trie {

class TrieNode{

private TrieNode[] links;

private final int R = 26;

private boolean isEnd;

TrieNode(){

links = new TrieNode[26];

isEnd = false;

}

public boolean containsKey(char ch){

return links[ch - 'a'] != null;

}

public TrieNode get(char ch){

return links[ch - 'a'];

}

public void put(char ch, TrieNode node){

links[ch - 'a'] = node;

}

public void setEnd(){

isEnd = true;

}

public boolean getEnd(){

return isEnd;

}

}

private TrieNode root;

/\*\* Initialize your data structure here. \*/

public Trie() {

root = new TrieNode();

}

/\*\* Inserts a word into the trie. \*/

public void insert(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

if (!node.containsKey(word.charAt(i))){

node.put(word.charAt(i), new TrieNode());

}

node = node.get(word.charAt(i));

}

node.setEnd();

}

/\*\* Returns if the word is in the trie. \*/

public boolean search(String word) {

TrieNode node = searchPrefix(word);

return node != null && node.getEnd();

}

/\*\* Returns if there is any word in the trie that starts with the given prefix. \*/

public boolean startsWith(String prefix) {

TrieNode node = searchPrefix(prefix);

return node != null;

}

private TrieNode searchPrefix(String prefix){

TrieNode node = root;

for (int i = 0; i < prefix.length(); i++){

if (!node.containsKey(prefix.charAt(i))){

return null;

}

node = node.get(prefix.charAt(i));

}

return node;

}

}

/\*\*

\* Your Trie object will be instantiated and called as such:

\* Trie obj = new Trie();

\* obj.insert(word);

\* boolean param\_2 = obj.search(word);

\* boolean param\_3 = obj.startsWith(prefix);

\*/

Better version:

class Trie {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode (){

children = new TrieNode[26];

isEnd = false;

}

}

TrieNode root;

/\*\* Initialize your data structure here. \*/

public Trie() {

root = new TrieNode();

}

/\*\* Inserts a word into the trie. \*/

public void insert(String word) {

TrieNode node = root;

for (char c : word.toCharArray()){

if (node.children[c - 'a'] == null){

node.children[c- 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

/\*\* Returns if the word is in the trie. \*/

public boolean search(String word) {

TrieNode node = root;

for (char c : word.toCharArray()){

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return node.isEnd;

}

/\*\* Returns if there is any word in the trie that starts with the given prefix. \*/

public boolean startsWith(String prefix) {

TrieNode node = root;

for (char c : prefix.toCharArray()){

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return true;

}

}

/\*\*

\* Your Trie object will be instantiated and called as such:

\* Trie obj = new Trie();

\* obj.insert(word);

\* boolean param\_2 = obj.search(word);

\* boolean param\_3 = obj.startsWith(prefix);

\*/

## 211\_Add and Search Word - Data structure design.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a data structure that supports the following two operations:

void addWord(word)

bool search(word)

search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.

For example:

addWord("bad")

addWord("dad")

addWord("mad")

search("pad") -> false

search("bad") -> true

search(".ad") -> true

search("b..") -> true

Note:

You may assume that all words are consist of lowercase letters a-z.

https://www.jiuzhang.com/solution/add-and-search-word/

class WordDictionary {

class TrieNode {

TrieNode[] links;

int R = 26;

boolean isEnd;

public TrieNode() {

links = new TrieNode[R];

isEnd = false;

}

public boolean containsKey(char c){

return links[c - 'a'] != null;

}

public void put(char c, TrieNode node){

links[c - 'a'] = node;

}

public TrieNode get(char c){

return links[c - 'a'];

}

public void setEnd(){

isEnd = true;

}

public boolean getEnd(){

return isEnd;

}

public TrieNode[] getLinks(){

return links;

}

}

private TrieNode root;

/\*\* Initialize your data structure here. \*/

public WordDictionary() {

root = new TrieNode();

}

/\*\* Adds a word into the data structure. \*/

public void addWord(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (!node.containsKey(c)){

node.put(c, new TrieNode());

}

node = node.get(c);

}

node.setEnd();

}

/\*\* Returns if the word is in the data structure. A word could contain the dot character '.' to represent any one letter. \*/

public boolean search(String word) {

return search(word, root, 0);

}

private boolean search(String word, TrieNode node, int start) {

if (start == word.length()){

return node.getEnd();

}

char c = word.charAt(start);

if (c == '.'){

for (TrieNode link : node.getLinks()){

if (link != null && search(word, link, start+1)){

return true;

}

}

return false;

}else if (node.containsKey(c)){

node = node.get(c);

return search(word, node, start + 1);

} else{

return false;

}

}

}

/\*\*

\* Your WordDictionary object will be instantiated and called as such:

\* WordDictionary obj = new WordDictionary();

\* obj.addWord(word);

\* boolean param\_2 = obj.search(word);

\*/

class WordDictionary {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode (){

children = new TrieNode[26];

isEnd = false;

}

}

/\*\* Initialize your data structure here. \*/

TrieNode root;

public WordDictionary() {

root = new TrieNode();

}

/\*\* Adds a word into the data structure. \*/

public void addWord(String word) {

TrieNode node = root;

for (char c : word.toCharArray()){

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

/\*\* Returns if the word is in the data structure. A word could contain the dot character '.' to represent any one letter. \*/

public boolean search(String word) {

return search(word, root, 0);

}

private boolean search(String word, TrieNode node, int pos){

if (pos == word.length()){

return node.isEnd;

}

char c = word.charAt(pos);

if (c == '.'){

for (TrieNode child : node.children){

if (child != null && search(word, child, pos + 1)){

return true;

}

}

return false;

}else if (node.children[c - 'a'] != null){

node = node.children[c - 'a'];

return search(word, node, pos+1);

}

return false;

}

}

/\*\*

\* Your WordDictionary object will be instantiated and called as such:

\* WordDictionary obj = new WordDictionary();

\* obj.addWord(word);

\* boolean param\_2 = obj.search(word);

\*/

## 225\_Implement Stack using Queues.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement the following operations of a stack using queues.

push(x) -- Push element x onto stack.

pop() -- Removes the element on top of the stack.

top() -- Get the top element.

empty() -- Return whether the stack is empty.

Notes:

You must use only standard operations of a queue -- which means only push to back, peek/pop from front, size, and is empty operations are valid.

Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.

You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

Method 1:

Push: O(n)

class MyStack {

Queue<Integer> q;

/\*\* Initialize your data structure here. \*/

public MyStack() {

q = new LinkedList<Integer>();

}

/\*\* Push element x onto stack. \*/

public void push(int x) {

q.offer(x);

for (int i = 0; i < q.size() - 1; i++){

q.offer(q.poll());

}

}

/\*\* Removes the element on top of the stack and returns that element. \*/

public int pop() {

return q.poll();

}

/\*\* Get the top element. \*/

public int top() {

return q.peek();

}

/\*\* Returns whether the stack is empty. \*/

public boolean empty() {

return q.isEmpty();

}

}

/\*\*

\* Your MyStack object will be instantiated and called as such:

\* MyStack obj = new MyStack();

\* obj.push(x);

\* int param\_2 = obj.pop();

\* int param\_3 = obj.top();

\* boolean param\_4 = obj.empty();

\*/

Two queues:

class MyStack {

Queue<Integer> main;

Queue<Integer> backup;

/\*\* Initialize your data structure here. \*/

public MyStack() {

main = new LinkedList<>();

backup = new LinkedList<>();

}

/\*\* Push element x onto stack. \*/

public void push(int x) {

backup.offer(x);

while (!main.isEmpty()){

backup.offer(main.poll());

}

Queue<Integer> temp = new LinkedList<>();

temp = backup;

backup = main;

main = temp;

}

/\*\* Removes the element on top of the stack and returns that element. \*/

public int pop() {

return main.poll();

}

/\*\* Get the top element. \*/

public int top() {

return main.peek();

}

/\*\* Returns whether the stack is empty. \*/

public boolean empty() {

return main.isEmpty();

}

}

/\*\*

\* Your MyStack object will be instantiated and called as such:

\* MyStack obj = new MyStack();

\* obj.push(x);

\* int param\_2 = obj.pop();

\* int param\_3 = obj.top();

\* boolean param\_4 = obj.empty();

\*/

## 232\_ImplementQueueUsingStacks.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement the following operations of a queue using stacks.

push(x) -- Push element x to the back of queue.

pop() -- Removes the element from in front of queue.

peek() -- Get the front element.

empty() -- Return whether the queue is empty.

Example:

MyQueue queue = new MyQueue();

queue.push(1);

queue.push(2);

queue.peek(); // returns 1

queue.pop(); // returns 1

queue.empty(); // returns false

class MyQueue {

Stack<Integer> masterStack;

Stack<Integer> slaveStack;

/\*\* Initialize your data structure here. \*/

public MyQueue() {

masterStack = new Stack<>();

slaveStack = new Stack<>();

}

/\*\* Push element x to the back of queue. \*/

public void push(int x) {

slaveStack.push(x);

}

/\*\* Removes the element from in front of queue and returns that element. \*/

public int pop() {

if (!masterStack.isEmpty()){

return masterStack.pop();

}

while (!slaveStack.isEmpty()){

masterStack.push(slaveStack.pop());

}

return masterStack.pop();

}

/\*\* Get the front element. \*/

public int peek() {

if (!masterStack.isEmpty()){

return masterStack.peek();

}

while (!slaveStack.isEmpty()){

masterStack.push(slaveStack.pop());

}

return masterStack.peek();

}

/\*\* Returns whether the queue is empty. \*/

public boolean empty() {

return masterStack.isEmpty() && slaveStack.isEmpty();

}

}

/\*\*

\* Your MyQueue object will be instantiated and called as such:

\* MyQueue obj = new MyQueue();

\* obj.push(x);

\* int param\_2 = obj.pop();

\* int param\_3 = obj.peek();

\* boolean param\_4 = obj.empty();

\*/

## 348\_Design\_Tic\_Tac.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a Tic-tac-toe game that is played between two players on a n x n grid.

You may assume the following rules:

A move is guaranteed to be valid and is placed on an empty block.

Once a winning condition is reached, no more moves is allowed.

A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Example:

Given n = 3, assume that player 1 is "X" and player 2 is "O" in the board.

TicTacToe toe = new TicTacToe(3);

toe.move(0, 0, 1); -> Returns 0 (no one wins)

|X| | |

| | | | // Player 1 makes a move at (0, 0).

| | | |

toe.move(0, 2, 2); -> Returns 0 (no one wins)

|X| |O|

| | | | // Player 2 makes a move at (0, 2).

| | | |

toe.move(2, 2, 1); -> Returns 0 (no one wins)

|X| |O|

| | | | // Player 1 makes a move at (2, 2).

| | |X|

toe.move(1, 1, 2); -> Returns 0 (no one wins)

|X| |O|

| |O| | // Player 2 makes a move at (1, 1).

| | |X|

toe.move(2, 0, 1); -> Returns 0 (no one wins)

|X| |O|

| |O| | // Player 1 makes a move at (2, 0).

|X| |X|

toe.move(1, 0, 2); -> Returns 0 (no one wins)

|X| |O|

|O|O| | // Player 2 makes a move at (1, 0).

|X| |X|

toe.move(2, 1, 1); -> Returns 1 (player 1 wins)

|X| |O|

|O|O| | // Player 1 makes a move at (2, 1).

|X|X|X|

Follow up:

Could you do better than O(n2) per move() operation?

Method 1:

Time complexity: O(n)

Space complexity: O(n^2)

class TicTacToe {

private int[][] board;

/\*\* Initialize your data structure here. \*/

public TicTacToe(int n) {

board = new int[n][n];

}

/\*\* Player {player} makes a move at ({row}, {col}).

@param row The row of the board.

@param col The column of the board.

@param player The player, can be either 1 or 2.

@return The current winning condition, can be either:

0: No one wins.

1: Player 1 wins.

2: Player 2 wins. \*/

public int move(int row, int col, int player) {

if (player == 1){

board[row][col] = 1;

}else{

board[row][col] = 2;

}

return checkWin(row, col, player);

}

private int checkWin(int row, int col, int player){

int n = board.length;

boolean win = true;

for (int i = 0; i < n; i++){

if (board[i][col] != player){

win = false;

break;

}

}

if (win == true){

return player;

}

win = true;

for (int j = 0; j < n; j++){

if (board[row][j] != player){

win = false;

break;

}

}

if (win == true){

return player;

}

if (row - col == 0){

win = true;

for (int i = 0; i < n; i++){

if (board[i][i] != player){

win = false;

break;

}

}

if (win == true){

return player;

}

}

if (row + col == n - 1){

win = true;

for (int i = 0; i < n; i++){

if (board[i][n - 1 - i] != player){

win = false;

break;

}

}

if (win == true){

return player;

}

}

return 0;

}

}

/\*\*

\* Your TicTacToe object will be instantiated and called as such:

\* TicTacToe obj = new TicTacToe(n);

\* int param\_1 = obj.move(row,col,player);

\*/

Method 2:

Time complexity: O(1)

Space complexity: O(n)

The key observation is that in order to win Tic-Tac-Toe you must have the entire row or column.

Thus, we don’t need to keep track of an entire n^2 board. We only need to keep a count for each row and column.

If at any time a row or column matches the size of the board then that player has won.

To keep track of which player, I add one for Player1 and -1 for Player2. There are two additional variables to

keep track of the count of the diagonals. Each time a player places a piece we just need to check the count of that row,

column, diagonal and anti-diagonal.

public class TicTacToe {

private int[] rows;

private int[] cols;

private int diagonal;

private int antiDiagonal;

/\*\* Initialize your data structure here. \*/

public TicTacToe(int n) {

rows = new int[n];

cols = new int[n];

}

/\*\* Player {player} makes a move at ({row}, {col}).

@param row The row of the board.

@param col The column of the board.

@param player The player, can be either 1 or 2.

@return The current winning condition, can be either:

0: No one wins.

1: Player 1 wins.

2: Player 2 wins. \*/

public int move(int row, int col, int player) {

int toAdd = player == 1 ? 1 : -1;

rows[row] += toAdd;

cols[col] += toAdd;

if (row == col)

{

diagonal += toAdd;

}

if (col == (cols.length - row - 1))

{

antiDiagonal += toAdd;

}

int size = rows.length;

if (Math.abs(rows[row]) == size ||

Math.abs(cols[col]) == size ||

Math.abs(diagonal) == size ||

Math.abs(antiDiagonal) == size)

{

return player;

}

return 0;

}

## 353\_DesignSnakeGame.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a Snake game that is played on a device with screen size = width x height. Play the game online if you are not familiar with

the game.

The snake is initially positioned at the top left corner (0,0) with length = 1 unit.

You are given a list of food's positions in row-column order. When a snake eats the food, its length and the game's score both

increase by 1.

Each food appears one by one on the screen. For example, the second food will not appear until the first food was eaten by the snake.

When a food does appear on the screen, it is guaranteed that it will not appear on a block occupied by the snake.

Example:

Given width = 3, height = 2, and food = [[1,2],[0,1]].

Snake snake = new Snake(width, height, food);

Initially the snake appears at position (0,0) and the food at (1,2).

|S| | |

| | |F|

snake.move("R"); -> Returns 0

| |S| |

| | |F|

snake.move("D"); -> Returns 0

| | | |

| |S|F|

snake.move("R"); -> Returns 1 (Snake eats the first food and right after that, the second food appears at (0,1) )

| |F| |

| |S|S|

snake.move("U"); -> Returns 1

| |F|S|

| | |S|

snake.move("L"); -> Returns 2 (Snake eats the second food)

| |S|S|

| | |S|

snake.move("U"); -> Returns -1 (Game over because snake collides with border)

class SnakeGame {

int[][] food;

int width;

int height;

int foodIndex;

Queue<Integer> queue;

int row; //head location

int col; //head location

/\*\* Initialize your data structure here.

@param width - screen width

@param height - screen height

@param food - A list of food positions

E.g food = [[1,1], [1,0]] means the first food is positioned at [1,1], the second is at [1,0]. \*/

public SnakeGame(int width, int height, int[][] food) {

this.food = food;

this.width = width;

this.height = height;

foodIndex = 0;

queue = new LinkedList<>();

queue.offer(0);

row = 0;

col = 0;

}

/\*\* Moves the snake.

@param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D' = Down

@return The game's score after the move. Return -1 if game over.

Game over when snake crosses the screen boundary or bites its body. \*/

public int move(String direction) {

if (direction.equals("U")){

row--;

}else if (direction.equals("D")){

row++;

}else if (direction.equals("L")){

col--;

}else{

col++;

}

if (row < height && row >= 0 && col < width && col >= 0){

int key = row \* width + col;

if (queue.contains(key) && queue.peek() != key){//if new head and tail are the point, it is okay

return -1;

}

queue.offer(key);

if (foodIndex < food.length && row == food[foodIndex][0] && col== food[foodIndex][1]){

foodIndex++;

}else{

queue.poll();

}

return foodIndex;

}

return -1;

}

}

/\*\*

\* Your SnakeGame object will be instantiated and called as such:

\* SnakeGame obj = new SnakeGame(width, height, food);

\* int param\_1 = obj.move(direction);

\*/

## 362\_DesignHitCounter.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a hit counter which counts the number of hits received in the past 5 minutes.

Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system

in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1.

It is possible that several hits arrive roughly at the same time.

Example:

HitCounter counter = new HitCounter();

// hit at timestamp 1.

counter.hit(1);

// hit at timestamp 2.

counter.hit(2);

// hit at timestamp 3.

counter.hit(3);

// get hits at timestamp 4, should return 3.

counter.getHits(4);

// hit at timestamp 300.

counter.hit(300);

// get hits at timestamp 300, should return 4.

counter.getHits(300);

// get hits at timestamp 301, should return 3.

counter.getHits(301);

Follow up:

What if the number of hits per second could be very large? Does your design scale?

O(s) s is total seconds in given time interval, in this case 300.

basic ideal is using buckets. 1 bucket for every second because we only need to keep the recent hits info for 300 seconds.

hit[] array is wrapped around by mod operation. Each hit bucket is associated with times[] bucket

which record current time. If it is not current time, it means it is 300s or 600s... ago and need to reset to 1.

O(1) hit() O(s) getHits()

Method 1: for large data

class HitCounter {

private int[] times;

private int[] hits;

private int N;

/\*\* Initialize your data structure here. \*/

public HitCounter() {

N = 300;

times = new int[N];

hits = new int[N];

}

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

public void hit(int timestamp) {

int index = timestamp % N;

if (times[index] != timestamp){

times[index] = timestamp;

hits[index] = 1;

}else{

hits[index]++;

}

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

public int getHits(int timestamp) {

int sum = 0;

for (int i = 0; i < N; i++){

if (timestamp - times[i] < N){

sum += hits[i];

}

}

return sum;

}

}

/\*\*

\* Your HitCounter object will be instantiated and called as such:

\* HitCounter obj = new HitCounter();

\* obj.hit(timestamp);

\* int param\_2 = obj.getHits(timestamp);

\*/

Method 2: for small data

class HitCounter {

Queue<Integer> queue;

/\*\* Initialize your data structure here. \*/

public HitCounter() {

queue = new LinkedList<>();

}

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

public void hit(int timestamp) {

queue.offer(timestamp);

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

public int getHits(int timestamp) {

while (!queue.isEmpty() && timestamp - queue.peek() >= 300){

queue.poll();

}

return queue.size();

}

}

## 379\_[Design Phone Directory](https://leetcode.com/problems/design-phone-directory).java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a Phone Directory which supports the following operations:

get: Provide a number which is not assigned to anyone.

check: Check if a number is available or not.

release: Recycle or release a number.

Example:

// Init a phone directory containing a total of 3 numbers: 0, 1, and 2.

PhoneDirectory directory = new PhoneDirectory(3);

// It can return any available phone number. Here we assume it returns 0.

directory.get();

// Assume it returns 1.

directory.get();

// The number 2 is available, so return true.

directory.check(2);

// It returns 2, the only number that is left.

directory.get();

// The number 2 is no longer available, so return false.

directory.check(2);

// Release number 2 back to the pool.

directory.release(2);

// Number 2 is available again, return true.

directory.check(2);

class PhoneDirectory {

Queue<Integer> queue = new LinkedList<>();

Set<Integer> set = new HashSet<>();

int maxNumbers;

/\*\* Initialize your data structure here

@param maxNumbers - The maximum numbers that can be stored in the phone directory. \*/

public PhoneDirectory(int maxNumbers) {

this.maxNumbers = maxNumbers;

for (int i = 0; i < maxNumbers; ++i){

queue.offer(i);

}

}

/\*\* Provide a number which is not assigned to anyone.

@return - Return an available number. Return -1 if none is available. \*/

public int get() {

if (!queue.isEmpty()){

int val = queue.poll();

set.add(val);

return val;

}

return -1;

}

/\*\* Check if a number is available or not. \*/

public boolean check(int number) {

if (number >= maxNumbers || number < 0 || set.contains(number)){

return false;

}

return true;

}

/\*\* Recycle or release a number. \*/

public void release(int number) {

if (set.contains(number)){

set.remove(number);

queue.offer(number);

}

}

}

/\*\*

\* Your PhoneDirectory object will be instantiated and called as such:

\* PhoneDirectory obj = new PhoneDirectory(maxNumbers);

\* int param\_1 = obj.get();

\* boolean param\_2 = obj.check(number);

\* obj.release(number);

\*/

## 419\_Battleships.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an 2D board, count how many battleships are in it. The battleships are represented with 'X's, empty slots are represented with '.'s. You may assume the following rules:

You receive a valid board, made of only battleships or empty slots.

Battleships can only be placed horizontally or vertically. In other words, they can only be made of the shape 1xN (1 row, N columns) or Nx1 (N rows, 1 column), where N can be of any size.

At least one horizontal or vertical cell separates between two battleships - there are no adjacent battleships.

Example:

X..X

...X

...X

In the above board there are 2 battleships.

Invalid Example:

...X

XXXX

...X

This is an invalid board that you will not receive - as battleships will always have a cell separating between them.

Follow up:

Could you do it in one-pass, using only O(1) extra memory and without modifying the value of the board?

Method 1: BFS

Time complexity: O(mn)

Space complexity: O(mn)

class Solution {

public int countBattleships(char[][] board) {

int count = 0;

int m = board.length;

int n = board[0].length;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (board[i][j] == 'X'){

board[i][j] = '1';

count++;

bfs(board, i, j);

}

}

}

return count;

}

private void bfs(char[][] board, int i, int j){

int m = board.length;

int n = board[0].length;

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

int[] dx = new int[]{0, 1, 0, -1};

int[] dy = new int[]{-1, 0, 1, 0};

qx.offer(i);

qy.offer(j);

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

for (int k = 0; k < dx.length; k++){

int nx = cx + dx[k];

int ny = cy + dy[k];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && board[nx][ny] == 'X'){

board[nx][ny] = '1';

qx.offer(nx);

qy.offer(ny);

}

}

}

}

}

Method 2: Best solution

only count top left one cell as one ship

https://leetcode.com/problems/battleships-in-a-board/discuss/90902/Simple-Java-Solution

class Solution {

public int countBattleships(char[][] board) {

int m = board.length;

int n = board[0].length;

int count = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (board[i][j] == '.'){

continue;

}

if (i > 0 && board[i-1][j] == 'X'){

continue;

}

if (j > 0 && board[i][j-1] == 'X'){

continue;

}

count++;

}

}

return count;

}

}

## 460\_LFUCache.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design and implement a data structure for Least Frequently Used (LFU) cache. It should support the

following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

put(key, value) - Set or insert the value if the key is not already present. When the cache reaches its

capacity, it should invalidate the least frequently used item before inserting a new item. For the purpose of this

problem, when there is a tie (i.e., two or more keys that have the same frequency), the least recently used key would

be evicted.

Follow up:

Could you do both operations in O(1) time complexity?

Example:

LFUCache cache = new LFUCache( 2 /\* capacity \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // returns 1

cache.put(3, 3); // evicts key 2

cache.get(2); // returns -1 (not found)

cache.get(3); // returns 3.

cache.put(4, 4); // evicts key 1.

cache.get(1); // returns -1 (not found)

cache.get(3); // returns 3

cache.get(4); // returns 4

Method 1:

Time complexity: O(logn)

class LFUCache {

class Pair{

int key;

int val;

int freq;

int stamp;

public Pair(int key, int value, int freq, int stamp){

this.key = key;

this.val = value;

this.freq = freq;

this.stamp = stamp;

}

}

private Map<Integer, Pair> map;

private Queue<Pair> pq;

private int capacity;

private int num;

private int stamp;

public LFUCache(int capacity) {

this.num = 0;

this.stamp = 0;

this.capacity = capacity;

this.map = new HashMap<Integer, Pair>();

this.pq = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare(Pair p1, Pair p2){

if (p1.freq == p2.freq){

return p1.stamp - p2.stamp;

}

return p1.freq - p2.freq;

}

});

}

public int get(int key) {

if (capacity == 0){

return -1;

}

int value = -1;

if (map.containsKey(key)){

Pair oldPair = map.get(key);

pq.remove(oldPair); //O(n) for priorityQueu, O(logn) for treemap

value = oldPair.val;

Pair newPair = new Pair(key, value, oldPair.freq + 1, stamp++);

map.put(key, newPair);

pq.offer(newPair);

}

return value;

}

public void put(int key, int value) {

if (capacity == 0){

return;

}

if (!map.containsKey(key)){

if (num == capacity){

Pair oldPair = pq.poll();

map.remove(oldPair.key);

}else{

num++;

}

Pair newPair = new Pair(key, value, 1, stamp++);

map.put(key, newPair);

pq.offer(newPair);

}else{

Pair oldPair = map.get(key);

pq.remove(oldPair);

Pair newPair = new Pair(key, value, oldPair.freq + 1, stamp++);

map.put(key, newPair);

pq.offer(newPair);

}

}

}

/\*\*

\* Your LFUCache object will be instantiated and called as such:

\* LFUCache obj = new LFUCache(capacity);

\* int param\_1 = obj.get(key);

\* obj.put(key,value);

\*/

## 526\_LoadBalancer.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement a load balancer for web servers. It provide the following functionality:

Add a new server to the cluster => add(server\_id).

Remove a bad server from the cluster => remove(server\_id).

Pick a server in the cluster randomly with equal probability => pick().

Have you met this question in a real interview? Yes

Example

At beginning, the cluster is empty => {}.

add(1)

add(2)

add(3)

pick()

>> 1 // the return value is random, it can be either 1, 2, or 3.

pick()

>> 2

pick()

>> 1

pick()

>> 3

remove(1)

pick()

>> 2

pick()

>> 3

pick()

>> 3

思路:

• 要在o(1)的时间内插入删除,只能hash。那hash可以getRandom吗?

– 不太好做

• 什么数据结构比较好getRandom?

– 数组

• 考虑hash与数组结合起来用,hash插入一个,数组也插入一个。那么问题来

了,数组删除元素怎么办? – 与最后插入的一个元素交换

• 那怎么o(1)时间在数组中找到要删除元素(要交换)的位置? – 用hash将元素的位置记下来

算法:

• 插入:

– 数组末尾加入这个元素

– Hash这个元素存下数组中的下标

• 删除:

– 通过hash找到这个元素在数组中的位置

– 数数组中这个元素和数组的末尾元素交换,交换后删除

– Hash中删除这个元素,更新数组原末尾元素现在在数组中的位置

• Pick:

– 数组中random一个返回

public class LoadBalancer {

int num;

Map<Integer, Integer> map;

List<Integer> list;

Random rand;

public LoadBalancer() {

map = new HashMap<>();

list = new ArrayList<>();

num = 0;

rand = new Random();

}

/\*

\* @param server\_id: add a new server to the cluster

\* @return: nothing

\*/

public void add(int server\_id) {

if (!map.containsKey(server\_id)){

list.add(server\_id);

map.put(server\_id, num);

num++;

}

}

/\*

\* @param server\_id: server\_id remove a bad server from the cluster

\* @return: nothing

\*/

public void remove(int server\_id) {

if (map.containsKey(server\_id)){

int idx = map.get(server\_id);

int lastItem = list.get(num - 1);

map.put(lastItem, idx);

list.set(idx, lastItem);

map.remove(server\_id);

list.remove(num - 1);

num--;

}

}

/\*

\* @return: pick a server in the cluster randomly with equal probability

\*/

public int pick() {

return list.get(rand.nextInt(num));

}

}

## 622\_DesignCircularQueue.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Your implementation should support following operations:

MyCircularQueue(k): Constructor, set the size of the queue to be k.

Front: Get the front item from the queue. If the queue is empty, return -1.

Rear: Get the last item from the queue. If the queue is empty, return -1.

enQueue(value): Insert an element into the circular queue. Return true if the operation is successful.

deQueue(): Delete an element from the circular queue. Return true if the operation is successful.

isEmpty(): Checks whether the circular queue is empty or not.

isFull(): Checks whether the circular queue is full or not.

Example:

MyCircularQueue circularQueue = new MycircularQueue(3); // set the size to be 3

circularQueue.enQueue(1); // return true

circularQueue.enQueue(2); // return true

circularQueue.enQueue(3); // return true

circularQueue.enQueue(4); // return false, the queue is full

circularQueue.Rear(); // return 3

circularQueue.isFull(); // return true

circularQueue.deQueue(); // return true

circularQueue.enQueue(4); // return true

circularQueue.Rear(); // return 4

Note:

All values will be in the range of [0, 1000].

The number of operations will be in the range of [1, 1000].

Please do not use the built-in Queue library.

class MyCircularQueue {

private int[] queue;

private int next;

/\*\* Initialize your data structure here. Set the size of the queue to be k. \*/

public MyCircularQueue(int k) {

queue = new int[k];

next = 0;

}

/\*\* Insert an element into the circular queue. Return true if the operation is successful. \*/

public boolean enQueue(int value) {

if (isFull()){

return false;

}

queue[next] = value;

next++;

return true;

}

/\*\* Delete an element from the circular queue. Return true if the operation is successful. \*/

public boolean deQueue() {

if (isEmpty()){

return false;

}

for (int i = 0; i < next - 1; i++){

queue[i] = queue[i+1];

}

queue[next-1] = 0;

next--;

return true;

}

/\*\* Get the front item from the queue. \*/

public int Front() {

if (isEmpty()){

return -1;

}

return queue[0];

}

/\*\* Get the last item from the queue. \*/

public int Rear() {

if (isEmpty()){

return -1;

}

return queue[next-1];

}

/\*\* Checks whether the circular queue is empty or not. \*/

public boolean isEmpty() {

return next == 0;

}

/\*\* Checks whether the circular queue is full or not. \*/

public boolean isFull() {

return next == queue.length;

}

}

/\*\*

\* Your MyCircularQueue object will be instantiated and called as such:

\* MyCircularQueue obj = new MyCircularQueue(k);

\* boolean param\_1 = obj.enQueue(value);

\* boolean param\_2 = obj.deQueue();

\* int param\_3 = obj.Front();

\* int param\_4 = obj.Rear();

\* boolean param\_5 = obj.isEmpty();

\* boolean param\_6 = obj.isFull();

\*/

# 1001\_Grid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a N x N grid of cells, each cell (x, y) with 0 <= x < N and 0 <= y < N has a lamp.

Initially, some number of lamps are on. lamps[i] tells us the location of the i-th lamp that is on. Each lamp that is on illuminates

every square on its x-axis, y-axis, and both diagonals (similar to a Queen in chess).

For the i-th query queries[i] = (x, y), the answer to the query is 1 if the cell (x, y) is illuminated, else 0.

After each query (x, y) [in the order given by queries], we turn off any lamps that are at cell (x, y) or are adjacent 8-directionally

(ie., share a corner or edge with cell (x, y).)

Return an array of answers. Each value answer[i] should be equal to the answer of the i-th query queries[i].

Example 1:

Input: N = 5, lamps = [[0,0],[4,4]], queries = [[1,1],[1,0]]

Output: [1,0]

Explanation:

Before performing the first query we have both lamps [0,0] and [4,4] on.

The grid representing which cells are lit looks like this, where [0,0] is the top left corner, and [4,4] is the bottom right corner:

1 1 1 1 1

1 1 0 0 1

1 0 1 0 1

1 0 0 1 1

1 1 1 1 1

Then the query at [1, 1] returns 1 because the cell is lit. After this query, the lamp at [0, 0] turns off, and the grid now looks

like this:

1 0 0 0 1

0 1 0 0 1

0 0 1 0 1

0 0 0 1 1

1 1 1 1 1

Before performing the second query we have only the lamp [4,4] on. Now the query at [1,0] returns 0, because the cell is no longer

lit.

Note:

1 <= N <= 10^9

0 <= lamps.length <= 20000

0 <= queries.length <= 20000

lamps[i].length == queries[i].length == 2

https://leetcode.com/problems/grid-illumination/discuss/243076/Java-Clean-Code-O(N)-Time-and-O(N)-Space-Beats-100

The basic idea is:

The row, column or diagonal will remain illuminated if there are >= 1 lamp

all the diagonals with 1 slope, x= y+c i.e. x-y = constant

all the diagonals with -1 slope, x= -y+c i.e x+y = constant

store the counts in separate maps

When a lamp is turned off, the count in respective row, column or diagonal decreases by 1

class Solution {

public int[] gridIllumination(int N, int[][] lamps, int[][] queries) {

Map<Integer, Integer> mapX = new HashMap<>();

Map<Integer, Integer> mapY = new HashMap<>();

Map<Integer, Integer> mapSlopePos = new HashMap<>();

Map<Integer, Integer> mapSlopeNeg = new HashMap<>();

Set<Integer> setLamp = new HashSet<>();

for (int[] lamp : lamps){

int x = lamp[0];

int y = lamp[1];

mapX.put(x, mapX.getOrDefault(x, 0) + 1);

mapY.put(y, mapY.getOrDefault(y, 0) + 1);

mapSlopePos.put(x+y, mapSlopePos.getOrDefault(x+y, 0) + 1);

mapSlopeNeg.put(x-y, mapSlopeNeg.getOrDefault(x-y, 0) + 1);

setLamp.add(x\*N+y);

}

int m = queries.length;

int[] res = new int[m];

for (int i = 0; i < m; i++){

int x = queries[i][0];

int y = queries[i][1];

if (mapX.getOrDefault(x, 0) > 0 || mapY.getOrDefault(y, 0) > 0 || mapSlopePos.getOrDefault(x+y, 0) > 0 || mapSlopeNeg.getOrDefault(x-y,0) > 0){

res[i] = 1;

}

for (int r = -1; r <= 1; r++){

for (int c = -1; c <= 1; c++){

int nx = x + r;

int ny = y + c;

if (nx >= 0 && nx < N && ny >= 0 && ny < N && setLamp.contains(nx\*N+ny)){

setLamp.remove(nx\*N+ny);

mapX.put(nx, mapX.getOrDefault(nx, 0) - 1);

mapY.put(ny, mapY.getOrDefault(ny, 0) - 1);

mapSlopePos.put(nx+ny, mapSlopePos.getOrDefault(nx+ny, 0) - 1);

mapSlopeNeg.put(nx-ny, mapSlopeNeg.getOrDefault(nx-ny, 0) - 1);

}

}

}

}

return res;

}

}

class Solution {

public int[] gridIllumination(int N, int[][] lamps, int[][] queries) {

int m = queries.length;

int[] res = new int[m];

//construct

Map<Integer, Integer> map = new HashMap<>();

Map<Integer, Set<Integer>> lits = new HashMap<>();

for (int[] lamp : lamps){

int x = lamp[0];

int y = lamp[1];

int key = x \* N + y;

map.put(key, map.getOrDefault(key, 0) + 1);

if (!lits.containsKey(key)){

lits.put(key, new HashSet<>());

}

//lit up y-axis

Set<Integer> set = lits.get(key);

for (int i = 0; i < N; i++){

if (i != x){

map.put(i\*N+y, map.getOrDefault(i\*N+y, 0) + 1);

set.add(i\*N+y);

}

}

//lit up x-axis

for (int j = 0; j < N; j++){

if (j != y){

map.put(x\*N+j, map.getOrDefault(x\*N+j, 0) + 1);

set.add(x\*N+j);

}

}

//lit up diagonal

int[] dx = {1, 1, -1, -1};

int[] dy = {1, -1, 1, -1};

for (int i = 0; i < dx.length; i++){

int nx = x + dx[i];

int ny = y + dy[i];

while (nx >= 0 && nx < N && ny >= 0 && ny < N){

map.put(nx\*N+ny, map.getOrDefault(nx\*N+ny, 0) + 1);

nx += dx[i];

ny += dy[i];

set.add(nx\*N+ny);

}

}

}

for (int k = 0; k < m; k++){

int x = queries[k][0];

int y = queries[k][1];

if (map.getOrDefault(x\*N+y, 0) > 0){

res[k] = 1;

for (int r = -1; r <= 1; r++){

for (int c = -1; c <= 1; c++){

int nx = x + r;

int ny = y + c;

if (nx >= 0 && nx < N && ny >= 0 && ny < N && lits.containsKey(nx\*N+ny)){

Set<Integer> set = lits.get(nx\*N+ny);

for (int on : set){

if (map.getOrDefault(on, 0) > 0){

map.put(on, map.get(on) - 1);

}

}

lits.remove(nx\*N+ny);

}

}

}

}

}

return res;

}

}

# 1002\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of strings made only from lowercase letters, return a list of all characters that show up in all strings within

the list (including duplicates). For example, if a character occurs 3 times in all strings but not 4 times, you need to include that

character three times in the final answer.

You may return the answer in any order.

Example 1:

Input: ["bella","label","roller"]

Output: ["e","l","l"]

Example 2:

Input: ["cool","lock","cook"]

Output: ["c","o"]

Note:

1 <= A.length <= 100

1 <= A[i].length <= 100

A[i][j] is a lowercase letter

class Solution {

public List<String> commonChars(String[] A) {

List<String> res = new ArrayList<>();

String first = A[0];

int len = A.length;

int[] hashA = new int[26];

for (char c : first.toCharArray()){

hashA[c - 'a']++;

if (len == 1){

res.add(String.valueOf(c));

}

}

if (len == 1){

return res;

}

for (int i = 1; i < len; i++){

String str = A[i];

int[] hash = new int[26];

for (char c : str.toCharArray()){

hash[c - 'a']++;

}

for (int j = 0; j < 26; j++){

hashA[j] = Math.min(hashA[j], hash[j]);

}

}

for (int i = 0; i < 26; i++){

while (hashA[i] > 0){

res.add(String.valueOf((char)(i + 'a')));

hashA[i]--;

}

}

return res;

}

}

Similar as Leetcode 916 word subset: https://github.com/optimisea/Leetcode/blob/master/Java/916\_Word.java

Better version:

class Solution {

public List<String> commonChars(String[] A) {

int[] minCount = new int[26];

for (int i = 0; i < 26; i++){

minCount[i] = Integer.MAX\_VALUE;

}

for (String str : A){

int[] hash = new int[26];

for (char c : str.toCharArray()){

hash[c - 'a']++;

}

for (int i = 0; i < 26; i++){

minCount[i] = Math.min(minCount[i], hash[i]);

}

}

List<String> res = new ArrayList<>();

for (int i = 0; i < 26; i++){

for (int j = 0; j < minCount[i]; j++){

res.add(String.valueOf((char)(i+'a')));

}

}

return res;

}

}

# 1003\_Check.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given that the string "abc" is valid.

From any valid string V, we may split V into two pieces X and Y such that X + Y (X concatenated with Y) is equal to V.

(X or Y may be empty.) Then, X + "abc" + Y is also valid.

If for example S = "abc", then examples of valid strings are: "abc", "aabcbc", "abcabc", "abcabcababcc".

Examples of invalid strings are: "abccba", "ab", "cababc", "bac".

Return true if and only if the given string S is valid.

Example 1:

Input: "aabcbc"

Output: true

Explanation:

We start with the valid string "abc".

Then we can insert another "abc" between "a" and "bc", resulting in "a" + "abc" + "bc" which is "aabcbc".

Example 2:

Input: "abcabcababcc"

Output: true

Explanation:

"abcabcabc" is valid after consecutive insertings of "abc".

Then we can insert "abc" before the last letter, resulting in "abcabcab" + "abc" + "c" which is "abcabcababcc".

Example 3:

Input: "abccba"

Output: false

Example 4:

Input: "cababc"

Output: false

Note:

1 <= S.length <= 20000

S[i] is 'a', 'b', or 'c'

Best solution: The same as 20. Valid Parentheses

https://leetcode.com/problems/valid-parentheses/

class Solution {

public boolean isValid(String S) {

Stack<Character> stack = new Stack<>();

for (char c : S.toCharArray()){

if (c == 'c'){

if (stack.isEmpty() || stack.pop() != 'b'){

return false;

}

if (stack.isEmpty() || stack.pop() != 'a'){

return false;

}

}else{

stack.push(c);

}

}

return stack.isEmpty();

}

}

Wrong solution below:

try aabbcc, should be false

class Solution {

public boolean isValid(String S) {

int numA = 0;

int numB = 0;

int numC = 0;

for (char c : S.toCharArray()){

if (c == 'a'){

numA++;

}else if (c == 'b'){

numB++;

}else{

numC++;

}

if (numA < numB || numA < numC || numB < numC){

return false;

}

}

return numA == numB && numB == numC;

}

}

# 1004\_Max.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of 0s and 1s, we may change up to K values from 0 to 1.

Return the length of the longest (contiguous) subarray that contains only 1s.

Example 1:

Input: A = [1,1,1,0,0,0,1,1,1,1,0], K = 2

Output: 6

Explanation:

[1,1,1,0,0,1,1,1,1,1,1]

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

Example 2:

Input: A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3

Output: 10

Explanation:

[0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

Note:

1 <= A.length <= 20000

0 <= K <= A.length

A[i] is 0 or 1

Sliding Window: the same as I & II

https://github.com/optimisea/Leetcode/blob/master/Java/485\_Max.java

https://github.com/optimisea/Leetcode/blob/master/Java/487\_Max.java

Also check Leetcode 926 Flip String to Monotone Increasing

https://github.com/optimisea/Leetcode/blob/master/Java/926\_Flip.java

class Solution {

public int longestOnes(int[] A, int K) {

int start = 0;

int end = 0;

int count = 0;

int res = 0;

while (end < A.length){

if (A[end] == 0){

count++;

}

end++;

while (count > K){

if (A[start] == 0){

count--;

}

start++;

}

res = Math.max(res, end - start);

}

return res;

}

}

# 1005\_Maximize.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of integers, we must modify the array in the following way: we choose an i and replace A[i] with -A[i], and we

repeat this process K times in total. (We may choose the same index i multiple times.)

Return the largest possible sum of the array after modifying it in this way.

Example 1:

Input: A = [4,2,3], K = 1

Output: 5

Explanation: Choose indices (1,) and A becomes [4,-2,3].

Example 2:

Input: A = [3,-1,0,2], K = 3

Output: 6

Explanation: Choose indices (1, 2, 2) and A becomes [3,1,0,2].

Example 3:

Input: A = [2,-3,-1,5,-4], K = 2

Output: 13

Explanation: Choose indices (1, 4) and A becomes [2,3,-1,5,4].

Note:

1 <= A.length <= 10000

1 <= K <= 10000

-100 <= A[i] <= 100

Time complexity: O(K\*logN)

class Solution {

public int largestSumAfterKNegations(int[] A, int K) {

Queue<Integer> minPQ = new PriorityQueue<>();

for (int i : A){

minPQ.offer(i);

}

while (K > 0){

int curr = minPQ.poll();

minPQ.offer(-curr);

K--;

}

int res = 0;

while (!minPQ.isEmpty()){

res += minPQ.poll();

}

return res;

}

}

# 1006\_Clumsy.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Normally, the factorial of a positive integer n is the product of all positive integers less than or equal to n. For example,

factorial(10) = 10 \* 9 \* 8 \* 7 \* 6 \* 5 \* 4 \* 3 \* 2 \* 1.

We instead make a clumsy factorial: using the integers in decreasing order, we swap out the multiply operations for a fixed

rotation of operations: multiply (\*), divide (/), add (+) and subtract (-) in this order.

For example, clumsy(10) = 10 \* 9 / 8 + 7 - 6 \* 5 / 4 + 3 - 2 \* 1. However, these operations are still applied using the usual

order of operations of arithmetic: we do all multiplication and division steps before any addition or subtraction steps, and

multiplication and division steps are processed left to right.

Additionally, the division that we use is floor division such that 10 \* 9 / 8 equals 11. This guarantees the result is an integer.

Implement the clumsy function as defined above: given an integer N, it returns the clumsy factorial of N.

Example 1:

Input: 4

Output: 7

Explanation: 7 = 4 \* 3 / 2 + 1

Example 2:

Input: 10

Output: 12

Explanation: 12 = 10 \* 9 / 8 + 7 - 6 \* 5 / 4 + 3 - 2 \* 1

Note:

1 <= N <= 10000

-2^31 <= answer <= 2^31 - 1 (The answer is guaranteed to fit within a 32-bit integer.)

Method: Iteration

class Solution {

public int clumsy(int N) {

if (N == 4){

return 4\*3/2 + 1;

}else if (N == 3){

return 3\*2/1;

}else if (N == 2){

return 2\*1;

}else if (N == 1){

return 1;

}

int res = N \* (N-1)/(N-2) + (N-3);

N -= 4;

while (N > 4){

res += - N \* (N-1)/(N-2) + N-3;

N -= 4;

}

if (N == 4){

res += -4\*3/2 + 1;

}else if (N == 3){

res += -3\*2/1;

}else if (N == 2){

res += -2\*1;

}else if (N == 1){

res += -1;

}

return res;

}

}

Method 2: recursion

class Solution {

public int clumsy(int N) {

if (N == 4){

return 4\*3/2 + 1;

}else if (N == 3){

return 3\*2/1;

}else if (N == 2){

return 2\*1;

}else if (N == 1){

return 1;

}

return N \* (N-1) / (N-2) + (N-3) + helper(N-4);

}

private int helper(int N){

if (N == 4){

return -4\*3/2 + 1;

}else if (N == 3){

return -3\*2/1;

}else if (N == 2){

return -2\*1;

}else if (N == 1){

return -1;

}

return -N \* (N-1) / (N-2) + (N-3) + helper(N-4);

}

}

# 1007\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a row of dominoes, A[i] and B[i] represent the top and bottom halves of the i-th domino. (A domino is a tile with two numbers

from 1 to 6 - one on each half of the tile.)

We may rotate the i-th domino, so that A[i] and B[i] swap values.

Return the minimum number of rotations so that all the values in A are the same, or all the values in B are the same.

If it cannot be done, return -1.

Example 1:

Input: A = [2,1,2,4,2,2], B = [5,2,6,2,3,2]

Output: 2

Explanation:

The first figure represents the dominoes as given by A and B: before we do any rotations.

If we rotate the second and fourth dominoes, we can make every value in the top row equal to 2, as indicated by the second figure.

Example 2:

Input: A = [3,5,1,2,3], B = [3,6,3,3,4]

Output: -1

Explanation:

In this case, it is not possible to rotate the dominoes to make one row of values equal.

Note:

1 <= A[i], B[i] <= 6

2 <= A.length == B.length <= 20000

Method: Greedy

class Solution {

public int minDominoRotations(int[] A, int[] B) {

int[] count = new int[7];

int n = A.length;

for (int i = 0; i < n; i++){

count[A[i]]++;

if (A[i] != B[i]){

count[B[i]]++;

}

}

int num = -1;

for (int i = 1; i <= 6; i++){

if (count[i] == n){

num = i;

break;

}

}

if (num == -1){

return -1;

}

int a = 0;

int b = 0;

for (int i = 0; i < n; i++){

if (A[i] == num){

a++;

}

if (B[i] == num){

b++;

}

}

return n - Math.max(a, b);

}

}

# 1012\_Complement.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Every non-negative integer N has a binary representation. For example, 5 can be represented as "101" in binary, 11 as "1011" in binary,

and so on. Note that except for N = 0, there are no leading zeroes in any binary representation.

The complement of a binary representation is the number in binary you get when changing every 1 to a 0 and 0 to a 1. For example,

the complement of "101" in binary is "010" in binary.

For a given number N in base-10, return the complement of it's binary representation as a base-10 integer.

Example 1:

Input: 5

Output: 2

Explanation: 5 is "101" in binary, with complement "010" in binary, which is 2 in base-10.

Example 2:

Input: 7

Output: 0

Explanation: 7 is "111" in binary, with complement "000" in binary, which is 0 in base-10.

Example 3:

Input: 10

Output: 5

Explanation: 10 is "1010" in binary, with complement "0101" in binary, which is 5 in base-10.

Note:

0 <= N < 10^9

https://leetcode.com/problems/complement-of-base-10-integer/discuss/256740/JavaC%2B%2BPython-Find-111.....1111-greater-N

Let's find the first number X that X = 1111....1 > N

And also, it has to be noticed that,

N = 0 is a corner case expecting1 as result.

Best solution 1 :

N + bitwiseComplement(N) = 11....11 = X

Then bitwiseComplement(N) = X - N

class Solution {

public int bitwiseComplement(int N) {

int X = 1;

while (N > X){

X = X \* 2 + 1;

}

return X - N;

}

}

Best soluion 2:

N ^ bitwiseComplement(N) = 11....11 = X

bitwiseComplement(N) = N ^ X

class Solution {

public int bitwiseComplement(int N) {

int X = 1;

while (N > X){

X = X \* 2 + 1;

}

return X ^ N;

}

}

Method 1: find the first

class Solution {

public int bitwiseComplement(int N) {

if (N == 0){

return 1;

}

int count = 0;

int M = N;

while (M > 0){

M /= 2;

count++;

}

int res = 0;

for (int i = 0; i < count; i++){

res |= (1 ^ ((N >> i) & 1)) << i;

}

return res;

}

}

Method 2:

class Solution {

public int bitwiseComplement(int N) {

if (N == 0){

return 1;

}

int res = 0;

int curr = 1;

while (N != 0){

if (N % 2 == 0){

res += curr;

}

N /= 2;

curr \*= 2;

}

return res;

}

}

# 1013\_Pairs.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a list of songs, the i-th song has a duration of time[i] seconds.

Return the number of pairs of songs for which their total duration in seconds is divisible by 60. Formally, we want the number of

indices i < j with (time[i] + time[j]) % 60 == 0.

Example 1:

Input: [30,20,150,100,40]

Output: 3

Explanation: Three pairs have a total duration divisible by 60:

(time[0] = 30, time[2] = 150): total duration 180

(time[1] = 20, time[3] = 100): total duration 120

(time[1] = 20, time[4] = 40): total duration 60

Example 2:

Input: [60,60,60]

Output: 3

Explanation: All three pairs have a total duration of 120, which is divisible by 60.

Note:

1 <= time.length <= 60000

1 <= time[i] <= 500

Best solution:

Similar as Leetcode 1: Two Sum; Target is 60, one number is t%60, the other is (60 - t%60)%60

class Solution {

public int numPairsDivisibleBy60(int[] time) {

Map<Integer, Integer> map = new HashMap<>();

int res = 0;

for (int t : time){

int key = (60 - t%60)%60;// the second %60 to for the corner case, e.g., 60, 60, 60

if (map.containsKey(key)){

res += map.get(key);

}

map.put(t%60, map.getOrDefault(t%60, 0) + 1);

}

return res;

}

}

Method 1:

class Solution {

public int numPairsDivisibleBy60(int[] time) {

Map<Integer, Integer> map = new HashMap<>();

int count = 0;

for (int t : time){

map.put(t%60, map.getOrDefault(t%60, 0) + 1);

}

for (int key : map.keySet()){

int cand = 60 - key;

if (map.containsKey(cand)){

if (key != cand){

count += map.get(key) \* map.get(cand);

}else{

int n = map.get(key);

count += n \* (n-1);

}

}else if (key == 0){

int n = map.get(key);

count += n \* (n-1);

}

}

return count/2;

}

}

class Solution {

public int numPairsDivisibleBy60(int[] time) {

Map<Integer, Integer> map = new HashMap<>();

int count = 0;

for (int t : time){

map.put(t%60, map.getOrDefault(t%60, 0) + 1);

}

for (int key : map.keySet()){

int cand = (60 - key) % 60;

if (map.containsKey(cand)){

if (key != cand){

count += map.get(key) \* map.get(cand);

}else{

int n = map.get(key);

count += n \* (n-1);

}

}

}

return count/2;

}

}

# 1014\_Capacity.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A conveyor belt has packages that must be shipped from one port to another within D days.

The i-th package on the conveyor belt has a weight of weights[i]. Each day, we load the ship with packages on the conveyor belt (in the order given by weights). We may not load more weight than the maximum weight capacity of the ship.

Return the least weight capacity of the ship that will result in all the packages on the conveyor belt being shipped within D days.

Example 1:

Input: weights = [1,2,3,4,5,6,7,8,9,10], D = 5

Output: 15

Explanation:

A ship capacity of 15 is the minimum to ship all the packages in 5 days like this:

1st day: 1, 2, 3, 4, 5

2nd day: 6, 7

3rd day: 8

4th day: 9

5th day: 10

Note that the cargo must be shipped in the order given, so using a ship of capacity 14 and splitting the packages into parts like (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) is not allowed.

Example 2:

Input: weights = [3,2,2,4,1,4], D = 3

Output: 6

Explanation:

A ship capacity of 6 is the minimum to ship all the packages in 3 days like this:

1st day: 3, 2

2nd day: 2, 4

3rd day: 1, 4

Example 3:

Input: weights = [1,2,3,1,1], D = 4

Output: 3

Explanation:

1st day: 1

2nd day: 2

3rd day: 3

4th day: 1, 1

Note:

1 <= D <= weights.length <= 50000

1 <= weights[i] <= 500

Similar as 875 Koko Eating Bananas

https://github.com/optimisea/Leetcode/blob/master/Java/875\_Koko.java

Simiar as 410 Split Array Largest Sum

https://github.com/optimisea/Leetcode/blob/master/Java/410\_Split.java

class Solution {

public int shipWithinDays(int[] weights, int D) {

int sum = 0;

int max = 0;

for (int i = 0; i < weights.length; i++){

sum += weights[i];

max = Math.max(max, weights[i]);

}

int start = max;

int end = sum;

while (start <= end){

int mid = start + (end - start) / 2;

int num = daysNeeded(weights, mid);

if (num <= D){

end = mid - 1;

}else{

start = mid + 1;

}

}

return start;

}

private int daysNeeded(int[] weights, int mid){

int count = 0;

int sum = 0;

for (int i : weights){

sum += i;

if (sum > mid){

count++;

sum = i;

}

}

return count + 1;

}

}

class Solution {

public int shipWithinDays(int[] weights, int D) {

int sum = 0;

int max = 0;

for (int i = 0; i < weights.length; i++){

sum += weights[i];

max = Math.max(max, weights[i]);

}

int start = max;

int end = sum;

while (start <= end){

int mid = start + (end - start) / 2;

if (lessThanNum(weights, mid, D)){

end = mid - 1;

}else{

start = mid + 1;

}

}

return start;

}

private boolean lessThanNum(int[] weights, int mid, int D){

int count = 0;

int sum = 0;

for (int i : weights){

sum += i;

if (sum > mid){

count++;

sum = i;

if (count >= D){

return false;

}

}

}

return true;

}

}

# 1020\_Partition.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of integers, return true if and only if we can partition the array into three non-empty parts with equal sums.

Formally, we can partition the array if we can find indexes i+1 < j with (A[0] + A[1] + ... + A[i] == A[i+1] + A[i+2] + ... + A[j-1] == A[j] + A[j-1] + ... + A[A.length - 1])

Example 1:

Input: [0,2,1,-6,6,-7,9,1,2,0,1]

Output: true

Explanation: 0 + 2 + 1 = -6 + 6 - 7 + 9 + 1 = 2 + 0 + 1

Example 2:

Input: [0,2,1,-6,6,7,9,-1,2,0,1]

Output: false

Example 3:

Input: [3,3,6,5,-2,2,5,1,-9,4]

Output: true

Explanation: 3 + 3 = 6 = 5 - 2 + 2 + 5 + 1 - 9 + 4

Note:

3 <= A.length <= 50000

-10000 <= A[i] <= 10000

class Solution {

public boolean canThreePartsEqualSum(int[] A) {

int sum = 0;

for (int i : A){

sum += i;

}

if (sum % 3 != 0){

return false;

}

int target = sum / 3;

int first = 0;

int firstInd = -1;

for (int i = 0; i < A.length; i++){

first += A[i];

if (first == target){

firstInd = i;

break;

}

}

int second = 0;

int secondInd = -1;

for (int i = firstInd + 1; i< A.length; i++){

second += A[i];

if (second == target){

secondInd = i;

break;

}

}

return firstInd != -1 && secondInd != -1;

}

}

# 1021\_Best.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of positive integers, A[i] represents the value of the i-th sightseeing spot, and two sightseeing spots i and j have distance j - i between them.

The score of a pair (i < j) of sightseeing spots is (A[i] + A[j] + i - j) : the sum of the values of the sightseeing spots, minus the distance between them.

Return the maximum score of a pair of sightseeing spots.

Example 1:

Input: [8,1,5,2,6]

Output: 11

Explanation: i = 0, j = 2, A[i] + A[j] + i - j = 8 + 5 + 0 - 2 = 11

Note:

2 <= A.length <= 50000

1 <= A[i] <= 1000

Method 1:

Time complexity: O(N)

Space complexity: O(N)

class Solution {

public int maxScoreSightseeingPair(int[] A) {

int max = Integer.MIN\_VALUE;

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < A.length; i++){

while (!stack.isEmpty() && A[i] - i > A[stack.peek()] - stack.peek()){

int ind = stack.pop();

max = Math.max(max, A[i] + A[ind] + ind - i);

}

stack.push(i);

}

while (stack.size() > 1){

int curr = stack.pop();

int next = stack.peek();

max = Math.max(max, A[curr] + A[next] + next - curr);

}

return max;

}

}

Method 2:

https://leetcode.com/problems/best-sightseeing-pair/discuss/260850/JavaC%2B%2BPython-One-Pass

Time complexity: O(N)

Space complexity: O(1)

class Solution {

public int maxScoreSightseeingPair(int[] A) {

int max = Integer.MIN\_VALUE;

int curr = 0;

for (int i : A){

max = Math.max(max, curr + i);

curr = Math.max(curr, i) - 1;

}

return max;

}

}

# 1022\_Smallest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer K, you need find the smallest positive integer N such that N is divisible by K, and N only contains the

digit 1.

Return the length of N. If there is no such N, return -1.

Example 1:

Input: 1

Output: 1

Explanation: The smallest answer is N = 1, which has length 1.

Example 2:

Input: 2

Output: -1

Explanation: There is no such positive integer N divisible by 2.

Example 3:

Input: 3

Output: 3

Explanation: The smallest answer is N = 111, which has length 3.

Note:

1 <= K <= 10^5

(A\*B+C)%K = (A%K \* B%K + C%K) %K

It is only necessary to iterate K times because we get a distinct result in each iteration, if the module is 0 then the number of iterations is the answer.

class Solution {

public int smallestRepunitDivByK(int K) {

if (K % 2 == 0 || K % 5 == 0){

return -1;

}

int N = 0;

for (int i = 1; i <= K; i++){

N = (N\*10 + 1) % K;

if (N == 0){

return i;

}

}

return -1;

}

}

# 1022\_Sum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, each node has value 0 or 1. Each root-to-leaf path represents a binary number starting with the most significant bit. For example, if the path is 0 -> 1 -> 1 -> 0 -> 1, then this could represent 01101 in binary, which is 13.

For all leaves in the tree, consider the numbers represented by the path from the root to that leaf.

Return the sum of these numbers.

Example 1:

Input: [1,0,1,0,1,0,1]

Output: 22

Explanation: (100) + (101) + (110) + (111) = 4 + 5 + 6 + 7 = 22

Note:

The number of nodes in the tree is between 1 and 1000.

node.val is 0 or 1.

The answer will not exceed 2^31 - 1.

class Solution {

int res = 0;

public int sumRootToLeaf(TreeNode root) {

if (root == null){

return 0;

}

dfs(root, 0);

return res;

}

private void dfs(TreeNode root, int preSum){

if (root == null){

return;

}

preSum = preSum \* 2 + root.val;

if (root.left == null && root.right == null){

res += preSum;

return;

}

dfs(root.left, preSum);

dfs(root.right, preSum);

}

}

# 1023\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary string S (a string consisting only of '0' and '1's) and a positive integer N, return true if and only if for every integer X from 1 to N, the binary representation of X is a substring of S.

Example 1:

Input: S = "0110", N = 3

Output: true

Example 2:

Input: S = "0110", N = 4

Output: false

Note:

1 <= S.length <= 1000

1 <= N <= 10^9

class Solution {

public boolean queryString(String S, int N) {

for (int i = 1; i <= N; i++){

String str = format(i);

if (S.indexOf(str) < 0){

return false;

}

}

return true;

}

private String format(int N){

StringBuilder sb = new StringBuilder();

while (N != 0){

sb.append(N & 1);

N >>= 1;

}

return sb.reverse().toString();

}

}

# 1023\_Camelcase.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A query word matches a given pattern if we can insert lowercase letters to the pattern word so that it equals the query. (We may insert each character at any position, and may insert 0 characters.)

Given a list of queries, and a pattern, return an answer list of booleans, where answer[i] is true if and only if queries[i] matches the pattern.

Example 1:

Input: queries = ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern = "FB"

Output: [true,false,true,true,false]

Explanation:

"FooBar" can be generated like this "F" + "oo" + "B" + "ar".

"FootBall" can be generated like this "F" + "oot" + "B" + "all".

"FrameBuffer" can be generated like this "F" + "rame" + "B" + "uffer".

Example 2:

Input: queries = ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern = "FoBa"

Output: [true,false,true,false,false]

Explanation:

"FooBar" can be generated like this "Fo" + "o" + "Ba" + "r".

"FootBall" can be generated like this "Fo" + "ot" + "Ba" + "ll".

Example 3:

Input: queries = ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern = "FoBaT"

Output: [false,true,false,false,false]

Explanation:

"FooBarTest" can be generated like this "Fo" + "o" + "Ba" + "r" + "T" + "est".

Note:

1 <= queries.length <= 100

1 <= queries[i].length <= 100

1 <= pattern.length <= 100

All strings consists only of lower and upper case English letters.

Better solution:

class Solution {

public List<Boolean> camelMatch(String[] queries, String pattern) {

List<Boolean> res = new ArrayList<>();

for (String query : queries){

if (isMatch(query, pattern)){

res.add(true);

}else{

res.add(false);

}

}

return res;

}

private boolean isMatch(String query, String pattern){

int j = 0;

for (int i = 0; i < query.length(); i++){

if (j < pattern.length() && query.charAt(i) == pattern.charAt(j)){

j++;

}else if (Character.isUpperCase(query.charAt(i))){

return false;

}

}

return j == pattern.length();

}

}

class Solution {

public List<Boolean> camelMatch(String[] queries, String pattern) {

List<Boolean> res = new ArrayList<>();

for (String query : queries){

if (isMatch(query, pattern)){

res.add(true);

}else{

res.add(false);

}

}

return res;

}

private boolean isMatch(String query, String pattern){

int i = 0;

int j = 0;

while (i < pattern.length()){

while (j < query.length()){

if (query.charAt(j) == pattern.charAt(i)){

i++;

j++;

break;

}

if (Character.isUpperCase(query.charAt(j))){

return false;

}

j++;

}

if (j == query.length()){

break;

}

}

while (j < query.length()){

if (Character.isUpperCase(query.charAt(j))){

return false;

}

j++;

}

return i == pattern.length();

}

}

# 1024\_Video.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a series of video clips from a sporting event that lasted T seconds. These video clips can be overlapping with each other and have varied lengths.

Each video clip clips[i] is an interval: it starts at time clips[i][0] and ends at time clips[i][1]. We can cut these clips into segments freely: for example, a clip [0, 7] can be cut into segments [0, 1] + [1, 3] + [3, 7].

Return the minimum number of clips needed so that we can cut the clips into segments that cover the entire sporting event ([0, T]). If the task is impossible, return -1.

Example 1:

Input: clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], T = 10

Output: 3

Explanation:

We take the clips [0,2], [8,10], [1,9]; a total of 3 clips.

Then, we can reconstruct the sporting event as follows:

We cut [1,9] into segments [1,2] + [2,8] + [8,9].

Now we have segments [0,2] + [2,8] + [8,10] which cover the sporting event [0, 10].

Example 2:

Input: clips = [[0,1],[1,2]], T = 5

Output: -1

Explanation:

We can't cover [0,5] with only [0,1] and [0,2].

Example 3:

Input: clips = [[0,1],[6,8],[0,2],[5,6],[0,4],[0,3],[6,7],[1,3],[4,7],[1,4],[2,5],[2,6],[3,4],[4,5],[5,7],[6,9]], T = 9

Output: 3

Explanation:

We can take clips [0,4], [4,7], and [6,9].

Example 4:

Input: clips = [[0,4],[2,8]], T = 5

Output: 2

Explanation:

Notice you can have extra video after the event ends.

Note:

1 <= clips.length <= 100

0 <= clips[i][0], clips[i][1] <= 100

0 <= T <= 100

Method 1:

Time complexity: O(N\*T)

Space complexity: O(T)

class Solution {

public int videoStitching(int[][] clips, int T) {

int[] dp = new int[T+1];

Arrays.fill(dp, T+1);

dp[0] = 0;

for (int i = 0; i <= T; i++){

for (int[] clip : clips){

if (clip[0] <= i && i <= clip[1]){

dp[i] = Math.min(dp[i], dp[clip[0]] + 1);

}

}

if (dp[i] == T+1){

return -1;

}

}

return dp[T];

}

}

# 1025\_Divisor.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Alice and Bob take turns playing a game, with Alice starting first.

Initially, there is a number N on the chalkboard. On each player's turn, that player makes a move consisting of:

Choosing any x with 0 < x < N and N % x == 0.

Replacing the number N on the chalkboard with N - x.

Also, if a player cannot make a move, they lose the game.

Return True if and only if Alice wins the game, assuming both players play optimally.

Example 1:

Input: 2

Output: true

Explanation: Alice chooses 1, and Bob has no more moves.

Example 2:

Input: 3

Output: false

Explanation: Alice chooses 1, Bob chooses 1, and Alice has no more moves.

Note:

1 <= N <= 1000

class Solution {

Map<Integer, Boolean> map = new HashMap<>();

public boolean divisorGame(int N) {

if (map.containsKey(N)){

return map.get(N);

}

for (int i = 1; i < N; i++){

if (N % i == 0){

if (!divisorGame(N-i)){

return true;

}

}

}

map.put(N, false);

return false;

}

}

# 1026\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the root of a binary tree, find the maximum value V for which there exists different nodes A and B where V = |A.val - B.val| and A is an ancestor of B.

(A node A is an ancestor of B if either: any child of A is equal to B, or any child of A is an ancestor of B.)

Example 1:

Input: [8,3,10,1,6,null,14,null,null,4,7,13]

Output: 7

Explanation:

We have various ancestor-node differences, some of which are given below :

|8 - 3| = 5

|3 - 7| = 4

|8 - 1| = 7

|10 - 13| = 3

Among all possible differences, the maximum value of 7 is obtained by |8 - 1| = 7.

Note:

The number of nodes in the tree is between 2 and 5000.

Each node will have value between 0 and 100000.

Method 1: Best Top down

class Solution {

public int maxAncestorDiff(TreeNode root) {

return dfs(root, root.val, root.val);

}

private int dfs(TreeNode root, int min, int max){

if (root == null){

return 0;

}

int res = Math.max(Math.abs(root.val-min), Math.abs(root.val - max));

max = Math.max(max, root.val);

min = Math.min(min, root.val);

int left = dfs(root.left, min, max);

int right = dfs(root.right, min, max);

return Math.max(res, Math.max(left, right));

}

}

Method 2: Bottom Up

class Solution {

class Node{

int min;

int max;

public Node(int max, int min){

this.min = min;

this.max = max;

}

}

int res = Integer.MIN\_VALUE;

public int maxAncestorDiff(TreeNode root) {

if (root == null){

return 0;

}

dfs(root);

return res;

}

private Node dfs(TreeNode root){

if (root == null){

return new Node(Integer.MIN\_VALUE, Integer.MAX\_VALUE);

}

Node left = dfs(root.left);

Node right = dfs(root.right);

int diff = Integer.MIN\_VALUE;

if (left.max == Integer.MIN\_VALUE && right.max == Integer.MIN\_VALUE){

return new Node(root.val, root.val);

}

if (left.max != Integer.MIN\_VALUE && left.min != Integer.MAX\_VALUE){

diff = Math.max(diff, Math.max(Math.abs(root.val - left.max), Math.abs(root.val - left.min)));

}

if (right.max != Integer.MIN\_VALUE && right.min != Integer.MAX\_VALUE){

diff = Math.max(diff, Math.max(Math.abs(root.val - right.max), Math.abs(root.val - right.min)));

}

res = Math.max(res, diff);

int currMax = Math.max(root.val, Math.max(left.max, right.max));

int currMin = Math.min(root.val, Math.min(left.min, right.min));

return new Node(currMax, currMin);

}

}

# 1028\_Convert.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a number N, return a string consisting of "0"s and "1"s that represents its value in base -2 (negative two).

The returned string must have no leading zeroes, unless the string is "0".

Example 1:

Input: 2

Output: "110"

Explantion: (-2) ^ 2 + (-2) ^ 1 = 2

Example 2:

Input: 3

Output: "111"

Explantion: (-2) ^ 2 + (-2) ^ 1 + (-2) ^ 0 = 3

Example 3:

Input: 4

Output: "100"

Explantion: (-2) ^ 2 = 4

Note:

0 <= N <= 10^9

https://www.geeksforgeeks.org/convert-number-negative-base-representation/

class Solution {

public String baseNeg2(int N) {

// If n is zero then in any base it will be 0 only

if (N == 0){

return "0";

}

String converted = "";

while (N != 0){

// Get remainder by negative base, it can be negative also

int remainder = N % (-2);

N /= -2;

// if remainder is negative, add abs(base) to it and add 1 to n

if (remainder < 0){

remainder += 2;

N += 1;

}

// convert remainder to string add into the result

converted = String.valueOf(remainder) + converted;

}

return converted;

}

}

General format:

class Solution {

public String baseNeg2(int N) {

// If n is zero then in any base it will be 0 only

if (N == 0){

return "0";

}

StringBuilder sb = new StringBuilder();

int negBase = -2;

while (N != 0){

// Get remainder by negative base, it can be negative also

int remainder = N % negBase;

N /= negBase;

// if remainder is negative, add abs(base) to it and add 1 to n

if (remainder < 0){

remainder += -1 \* negBase;

N += 1;

}

// convert remainder to string add into the result

sb.append(remainder);

}

return sb.reverse().toString();

}

}

5 % 2 = 1

5 % (-2) = -1

(-5) % 2 = -1

(-5) % (-2) = 1

Note that the sign of the result equals the sign of the dividend.

Says it in Java specs:

https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.17.3

# 1029\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of 0s and 1s, consider N\_i: the i-th subarray from A[0] to A[i] interpreted as a binary number (from most-significant-bit to least-significant-bit.)

Return a list of booleans answer, where answer[i] is true if and only if N\_i is divisible by 5.

Example 1:

Input: [0,1,1]

Output: [true,false,false]

Explanation:

The input numbers in binary are 0, 01, 011; which are 0, 1, and 3 in base-10. Only the first number is divisible by 5, so answer[0] is true.

Example 2:

Input: [1,1,1]

Output: [false,false,false]

Example 3:

Input: [0,1,1,1,1,1]

Output: [true,false,false,false,true,false]

Example 4:

Input: [1,1,1,0,1]

Output: [false,false,false,false,false]

Note:

1 <= A.length <= 30000

A[i] is 0 or 1

class Solution {

public List<Boolean> prefixesDivBy5(int[] A) {

List<Boolean> res = new ArrayList<>();

int n = A.length;

int sum = 0;

for (int i = 0; i < n; i++){

sum = (sum << 1) + A[i];

sum %= 5;

if (sum % 5 == 0){

res.add(true);

}else{

res.add(false);

}

}

return res;

}

}

# 1029\_Two.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are 2N people a company is planning to interview. The cost of flying the i-th person to city A is costs[i][0], and the cost of flying the i-th person to city B is costs[i][1].

Return the minimum cost to fly every person to a city such that exactly N people arrive in each city.

Example 1:

Input: [[10,20],[30,200],[400,50],[30,20]]

Output: 110

Explanation:

The first person goes to city A for a cost of 10.

The second person goes to city A for a cost of 30.

The third person goes to city B for a cost of 50.

The fourth person goes to city B for a cost of 20.

The total minimum cost is 10 + 30 + 50 + 20 = 110 to have half the people interviewing in each city.

Note:

1 <= costs.length <= 100

It is guaranteed that costs.length is even.

1 <= costs[i][0], costs[i][1] <= 1000

Method 1: Greedy

https://leetcode.com/problems/two-city-scheduling/discuss/278716/C%2B%2B-O(n-log-n)-sort-by-cost-difference

Time complexity: O(NlogN)

Space complexity: O(1)

class Solution {

public int twoCitySchedCost(int[][] costs) {

Arrays.sort(costs, new Comparator<int[]>(){

public int compare (int[] c1, int[] c2){

return c1[0] - c1[1] - (c2[0] - c2[1]);

}

});

int res = 0;

for (int i = 0; i < costs.length; i++){

if (i < costs.length / 2){

res += costs[i][0];

}else{

res += costs[i][1];

}

}

return res;

}

}

Method 2: DP

dp[i][j] represents the cost when considering first (i + j) people in which i people assigned to city A and j people assigned to city B.

Time complexity: O(N^2)

Space complexity: O(N^2)

class Solution {

public int twoCitySchedCost(int[][] costs) {

int N = costs.length/2;

int[][] dp = new int[N+1][N+1];//i people went to A, j people went to B

for (int i = 1; i <= N; i++){

dp[i][0] = dp[i-1][0] + costs[i-1][0];

}

for (int j = 1; j <= N; j++){

dp[0][j] = dp[0][j-1] + costs[j-1][1];

}

for (int i = 1; i <= N; i++){

for (int j = 1; j <= N; j++){

dp[i][j] = Math.min(dp[i-1][j] + costs[i+j-1][0], dp[i][j-1] + costs[i+j-1][1]);

}

}

return dp[N][N];

}

}

# 1030\_Matrix.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given a matrix with R rows and C columns has cells with integer coordinates (r, c), where 0 <= r < R and 0 <= c < C.

Additionally, we are given a cell in that matrix with coordinates (r0, c0).

Return the coordinates of all cells in the matrix, sorted by their distance from (r0, c0) from smallest distance to largest distance. Here, the distance between two cells (r1, c1) and (r2, c2) is the Manhattan distance, |r1 - r2| + |c1 - c2|. (You may return the answer in any order that satisfies this condition.)

Example 1:

Input: R = 1, C = 2, r0 = 0, c0 = 0

Output: [[0,0],[0,1]]

Explanation: The distances from (r0, c0) to other cells are: [0,1]

Example 2:

Input: R = 2, C = 2, r0 = 0, c0 = 1

Output: [[0,1],[0,0],[1,1],[1,0]]

Explanation: The distances from (r0, c0) to other cells are: [0,1,1,2]

The answer [[0,1],[1,1],[0,0],[1,0]] would also be accepted as correct.

Example 3:

Input: R = 2, C = 3, r0 = 1, c0 = 2

Output: [[1,2],[0,2],[1,1],[0,1],[1,0],[0,0]]

Explanation: The distances from (r0, c0) to other cells are: [0,1,1,2,2,3]

There are other answers that would also be accepted as correct, such as [[1,2],[1,1],[0,2],[1,0],[0,1],[0,0]].

Note:

1 <= R <= 100

1 <= C <= 100

0 <= r0 < R

0 <= c0 < C

class Solution {

public int[][] allCellsDistOrder(int R, int C, int r0, int c0) {

int[][] res = new int[R\*C][2];

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

boolean[][] visited = new boolean[R][C];

Queue<int[]> queue = new LinkedList<>();

visited[r0][c0] = true;

queue.offer(new int[]{r0, c0});

int index = 0;

while (!queue.isEmpty()){

int[] curr = queue.poll();

res[index++] = curr;

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < R & ny >= 0 && ny < C && !visited[nx][ny]){

visited[nx][ny] = true;

queue.offer(new int[]{nx, ny});

}

}

}

return res;

}

}

# 1031\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of non-negative integers, return the maximum sum of elements in two non-overlapping (contiguous) subarrays, which have lengths L and M. (For clarification, the L-length subarray could occur before or after the M-length subarray.)

Formally, return the largest V for which V = (A[i] + A[i+1] + ... + A[i+L-1]) + (A[j] + A[j+1] + ... + A[j+M-1]) and either:

0 <= i < i + L - 1 < j < j + M - 1 < A.length, or

0 <= j < j + M - 1 < i < i + L - 1 < A.length.

Example 1:

Input: A = [0,6,5,2,2,5,1,9,4], L = 1, M = 2

Output: 20

Explanation: One choice of subarrays is [9] with length 1, and [6,5] with length 2.

Example 2:

Input: A = [3,8,1,3,2,1,8,9,0], L = 3, M = 2

Output: 29

Explanation: One choice of subarrays is [3,8,1] with length 3, and [8,9] with length 2.

Example 3:

Input: A = [2,1,5,6,0,9,5,0,3,8], L = 4, M = 3

Output: 31

Explanation: One choice of subarrays is [5,6,0,9] with length 4, and [3,8] with length 3.

Note:

L >= 1

M >= 1

L + M <= A.length <= 1000

0 <= A[i] <= 1000

Method 1: DP

Time complexity: O(N)

Space complexity: O(N)

class Solution {

public int maxSumTwoNoOverlap(int[] A, int L, int M) {

int N = A.length;

int[] preSum = new int[N+1];

for (int i = 1; i <= N; i++){

preSum[i] = preSum[i-1] + A[i-1];

}

int res = preSum[L+M] or 0; //denote the result until the first L + M elements

int maxL = preSum[L] or 0; // denote the max sum of L element before the last M element

int maxM = preSum[M] or 0; // denote the max sum of M element before the last L element

for (int i = L + M; i <= N; i++){

maxL = Math.max(maxL, preSum[i-M] - preSum[i-L-M]);

maxM = Math.max(maxM, preSum[i-L] - preSum[i-L-M]);

res = Math.max(res, Math.max(maxL + preSum[i] - preSum[i-M], maxM + preSum[i]- preSum[i-L]));

}

return res;

}

}

Method 2: Brute force:

Time complexity: O(N^2)

Space complexity: O(N)

class Solution {

public int maxSumTwoNoOverlap(int[] A, int L, int M) {

int N = A.length;

int[] preSum = new int[N+1];

for (int i = 1; i <= N; i++){

preSum[i] = preSum[i-1] + A[i-1];

}

int res = 0;

for (int i = 0; i < N - L - M + 1; i++){

int sumL = preSum[i+L] - preSum[i];

for (int j = i + L; j < N - M + 1; j++){

int sumM = preSum[j+M] - preSum[j];

res = Math.max(res, sumL + sumM);

}

}

for (int i = 0; i < N - L - M + 1; i++){

int sumM = preSum[i+M] - preSum[i];

for (int j = i + M; j < N- L + 1; j++){

int sumL = preSum[j+L] - preSum[j];

res = Math.max(res, sumL + sumM);

}

}

return res;

}

}

# 1031\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2D array A, each cell is 0 (representing sea) or 1 (representing land)

A move consists of walking from one land square 4-directionally to another land square, or off the boundary of the grid.

Return the number of land squares in the grid for which we cannot walk off the boundary of the grid in any number of moves.

Example 1:

Input: [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]

Output: 3

Explanation:

There are three 1s that are enclosed by 0s, and one 1 that isn't enclosed because its on the boundary.

Example 2:

Input: [[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]

Output: 0

Explanation:

All 1s are either on the boundary or can reach the boundary.

Note:

1 <= A.length <= 500

1 <= A[i].length <= 500

0 <= A[i][j] <= 1

All rows have the same size.

class Solution {

public int numEnclaves(int[][] A) {

int m = A.length;

int n = A[0].length;

Queue<int[]> queue = new LinkedList<>();

int[][] dirs = {{1, 0}, { 0, 1}, {-1, 0}, {0, -1}};

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

if (A[i][0] == 1 && !visited[i][0]){

queue.offer(new int[]{i, 0});

visited[i][0] = true;

}

if (A[i][n-1] == 1 && !visited[i][n-1]){

queue.offer(new int[]{i, n-1});

visited[i][n-1] = true;

}

}

for (int j = 0; j < n; j++){

if (A[0][j] == 1 && !visited[0][j]){

queue.offer(new int[]{0, j});

visited[0][j] = true;

}

if (A[m-1][j] == 1 && !visited[m-1][j]){

queue.offer(new int[]{m-1, j});

visited[m-1][j] = true;

}

}

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && A[nx][ny] == 1 && !visited[nx][ny]){

queue.offer(new int[]{nx, ny});

visited[nx][ny] = true;

}

}

}

////next bfs to count how many "1"

int res = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (!visited[i][j] && A[i][j] == 1){

res += bfs(A, visited, i, j);

}

}

}

return res;

}

private int bfs(int[][] A, boolean[][] visited, int x, int y){

int m = A.length;

int n = A[0].length;

Queue<int[]> queue = new LinkedList<>();

int[][] dirs = {{1, 0}, { 0, 1}, {-1, 0}, {0, -1}};

queue.offer(new int[]{x, y});

visited[x][y] = true;

int count = 1;

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && A[nx][ny] == 1 && !visited[nx][ny]){

queue.offer(new int[]{nx, ny});

visited[nx][ny] = true;

count++;

}

}

}

return count;

}

}

# 1032\_Stream.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement the StreamChecker class as follows:

StreamChecker(words): Constructor, init the data structure with the given words.

query(letter): returns true if and only if for some k >= 1, the last k characters queried (in order from oldest to newest, including this letter just queried) spell one of the words in the given list.

Example:

StreamChecker streamChecker = new StreamChecker(["cd","f","kl"]); // init the dictionary.

streamChecker.query('a'); // return false

streamChecker.query('b'); // return false

streamChecker.query('c'); // return false

streamChecker.query('d'); // return true, because 'cd' is in the wordlist

streamChecker.query('e'); // return false

streamChecker.query('f'); // return true, because 'f' is in the wordlist

streamChecker.query('g'); // return false

streamChecker.query('h'); // return false

streamChecker.query('i'); // return false

streamChecker.query('j'); // return false

streamChecker.query('k'); // return false

streamChecker.query('l'); // return true, because 'kl' is in the wordlist

Note:

1 <= words.length <= 2000

1 <= words[i].length <= 2000

Words will only consist of lowercase English letters.

Queries will only consist of lowercase English letters.

The number of queries is at most 40000

Method 1: brute force

class StreamChecker {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode(){

children = new TrieNode[26];

isEnd = false;

}

}

TrieNode root;

Map<String, TrieNode> prefixMap;

public StreamChecker(String[] words) {

root = new TrieNode();

prefixMap = new HashMap<>();

prefixMap.put("", root);

for (String word : words){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

}

public boolean query(char letter) {

boolean isExist = false;

Map<String, TrieNode> temp = new HashMap<>();

for (String key : prefixMap.keySet()){

TrieNode node = prefixMap.get(key);

if (node.children[letter - 'a'] != null){

node = node.children[letter - 'a'];

temp.put(key + letter, node);

if (node.isEnd){

isExist = true;

}

}

}

if (root.children[letter -'a'] != null){

temp.put(String.valueOf(letter), root.children[letter - 'a']);

}

temp.putIfAbsent("", root);

prefixMap = temp;

return isExist;

}

}

/\*\*

\* Your StreamChecker object will be instantiated and called as such:

\* StreamChecker obj = new StreamChecker(words);

\* boolean param\_1 = obj.query(letter);

\*/

Method 2: reverse order

Store the words in the trie with reverse order, and check the query string from the end

class StreamChecker {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode(){

children = new TrieNode[26];

isEnd = false;

}

}

TrieNode root;

StringBuilder sb;

public StreamChecker(String[] words) {

root = new TrieNode();

sb = new StringBuilder();

for (String word : words){

TrieNode node = root;

for (int i = word.length() - 1; i >= 0; i--){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

}

public boolean query(char letter) {

sb.append(letter);

TrieNode node = root;

for (int i = sb.length() - 1; i >= 0; i--){

char c = sb.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

if (node.isEnd){

return true;

}

}

return false;

}

}

/\*\*

\* Your StreamChecker object will be instantiated and called as such:

\* StreamChecker obj = new StreamChecker(words);

\* boolean param\_1 = obj.query(letter);

\*/

# 1033\_Moving.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Three stones are on a number line at positions a, b, and c.

Each turn, you pick up a stone at an endpoint (ie., either the lowest or highest position stone), and move it to an unoccupied position between those endpoints. Formally, let's say the stones are currently at positions x, y, z with x < y < z. You pick up the stone at either position x or position z, and move that stone to an integer position k, with x < k < z and k != y.

The game ends when you cannot make any more moves, ie. the stones are in consecutive positions.

When the game ends, what is the minimum and maximum number of moves that you could have made? Return the answer as an length 2 array: answer = [minimum\_moves, maximum\_moves]

Example 1:

Input: a = 1, b = 2, c = 5

Output: [1,2]

Explanation: Move the stone from 5 to 3, or move the stone from 5 to 4 to 3.

Example 2:

Input: a = 4, b = 3, c = 2

Output: [0,0]

Explanation: We cannot make any moves.

Example 3:

Input: a = 3, b = 5, c = 1

Output: [1,2]

Explanation: Move the stone from 1 to 4; or move the stone from 1 to 2 to 4.

class Solution {

public int[] numMovesStones(int a, int b, int c) {

int[] arr = {a, b, c};

Arrays.sort(arr);

if (arr[2] - arr[0] == 2){

return new int[]{0, 0};

}

return new int[]{Math.min(arr[2] - arr[1], arr[1] - arr[0]) <= 2 ? 1 : 2, arr[2] - arr[0] - 2};

}

}

# 1034\_Coloring.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2-dimensional grid of integers, each value in the grid represents the color of the grid square at that location.

Two squares belong to the same connected component if and only if they have the same color and are next to each other in any of the 4 directions.

The border of a connected component is all the squares in the connected component that are either 4-directionally adjacent to a square not in the component, or on the boundary of the grid (the first or last row or column).

Given a square at location (r0, c0) in the grid and a color, color the border of the connected component of that square with the given color, and return the final grid.

Example 1:

Input: grid = [[1,1],[1,2]], r0 = 0, c0 = 0, color = 3

Output: [[3, 3], [3, 2]]

Example 2:

Input: grid = [[1,2,2],[2,3,2]], r0 = 0, c0 = 1, color = 3

Output: [[1, 3, 3], [2, 3, 3]]

Example 3:

Input: grid = [[1,1,1],[1,1,1],[1,1,1]], r0 = 1, c0 = 1, color = 2

Output: [[2, 2, 2], [2, 1, 2], [2, 2, 2]]

Note:

1 <= grid.length <= 50

1 <= grid[0].length <= 50

1 <= grid[i][j] <= 1000

0 <= r0 < grid.length

0 <= c0 < grid[0].length

1 <= color <= 1000

Best solution:

class Solution {

public int[][] colorBorder(int[][] grid, int r0, int c0, int color) {

int m = grid.length;

int n = grid[0].length;

Queue<int[]> queue = new LinkedList<>();

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

queue.offer(new int[]{r0, c0});

boolean[][] visited = new boolean[m][n];

visited[r0][c0] = true;

int orig = grid[r0][c0];

while (!queue.isEmpty()){

int[] curr = queue.poll();

int x = curr[0];

int y = curr[1];

if (x == 0 || x == m - 1 || y == 0 || y == n - 1){

grid[x][y] = color;

}

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

if (grid[nx][ny] == orig){

queue.offer(new int[]{nx, ny});

visited[nx][ny] = true;

}else{

grid[x][y] = color;

}

}

}

}

return grid;

}

}

Method 1: BFS

Time complexity: O(mn)

Space complexity: O(mn)

class Solution {

public int[][] colorBorder(int[][] grid, int r0, int c0, int color) {

int m = grid.length;

int n = grid[0].length;

int[][] res = new int[m][n];

Queue<int[]> queue = new LinkedList<>();

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

queue.offer(new int[]{r0, c0});

boolean[][] visited = new boolean[m][n];

visited[r0][c0] = true;

int orig = grid[r0][c0];

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int x = curr[0] + dir[0];

int y = curr[1] + dir[1];

if (x >= 0 && x < m && y >= 0 && y < n && !visited[x][y] && grid[x][y] == orig){

queue.offer(new int[]{x, y});

visited[x][y] = true;

}

}

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (!visited[i][j]){

res[i][j] = grid[i][j];

}else{

if (i == 0 || i == m - 1 || j == 0 || j == n - 1){

res[i][j] = color;

}else{

boolean isBorder = false;

for (int[] dir : dirs){

int x = i + dir[0];

int y = j + dir[1];

if (grid[x][y] != orig){

res[i][j] = color;

isBorder = true;

break;

}

}

if (!isBorder){

res[i][j] = grid[i][j];

}

}

}

}

}

return res;

}

}

# 1035\_Uncrossed.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We write the integers of A and B (in the order they are given) on two separate horizontal lines.

Now, we may draw connecting lines: a straight line connecting two numbers A[i] and B[j] such that:

A[i] == B[j];

The line we draw does not intersect any other connecting (non-horizontal) line.

Note that a connecting lines cannot intersect even at the endpoints: each number can only belong to one connecting line.

Return the maximum number of connecting lines we can draw in this way.

Example 1:

Input: A = [1,4,2], B = [1,2,4]

Output: 2

Explanation: We can draw 2 uncrossed lines as in the diagram.

We cannot draw 3 uncrossed lines, because the line from A[1]=4 to B[2]=4 will intersect the line from A[2]=2 to B[1]=2.

Example 2:

Input: A = [2,5,1,2,5], B = [10,5,2,1,5,2]

Output: 3

Example 3:

Input: A = [1,3,7,1,7,5], B = [1,9,2,5,1]

Output: 2

Note:

1 <= A.length <= 500

1 <= B.length <= 500

1 <= A[i], B[i] <= 2000

Longest common subsequence

Method 1: DP

Time complexity: O(mn)

Space complexity: O(mn)

class Solution {

public int maxUncrossedLines(int[] A, int[] B) {

int m = A.length;

int n = B.length;

int[][] dp = new int[m+1][n+1];

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (A[i-1] == B[j-1]){

dp[i][j] = dp[i-1][j-1] + 1;

}else{

dp[i][j] = Math.max(dp[i-1][j-1], Math.max(dp[i-1][j], dp[i][j-1]));

}

}

}

return dp[m][n];

}

}

Method 2: faster

class Solution {

public int maxUncrossedLines(int[] A, int[] B) {

int m = A.length;

int n = B.length;

int[][] dp = new int[m+1][n+1];

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (A[i-1] == B[j-1]){

dp[i][j] = dp[i-1][j-1] + 1;

}else{

dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);

}

}

}

return dp[m][n];

}

}

# 1036\_Escape.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a 1 million by 1 million grid, the coordinates of each grid square are (x, y) with 0 <= x, y < 10^6.

We start at the source square and want to reach the target square. Each move, we can walk to a 4-directionally adjacent square in the grid that isn't in the given list of blocked squares.

Return true if and only if it is possible to reach the target square through a sequence of moves.

Example 1:

Input: blocked = [[0,1],[1,0]], source = [0,0], target = [0,2]

Output: false

Explanation:

The target square is inaccessible starting from the source square, because we can't walk outside the grid.

Example 2:

Input: blocked = [], source = [0,0], target = [999999,999999]

Output: true

Explanation:

Because there are no blocked cells, it's possible to reach the target square.

Note:

0 <= blocked.length <= 200

blocked[i].length == 2

0 <= blocked[i][j] < 10^6

source.length == target.length == 2

0 <= source[i][j], target[i][j] < 10^6

source != target

https://leetcode.com/problems/escape-a-large-maze/discuss/282870/python-solution-with-picture-show-my-thoughts

need optimization to limit the bfs steps

class Solution {

public boolean isEscapePossible(int[][] blocked, int[] source, int[] target) {

return helper(blocked, source, target) && helper(blocked, target, source);

}

private boolean helper(int[][] blocked, int[] source, int[] target){

if (blocked.length == 0){

return true;

}

int maxStep = blocked.length;

Set<String> set = new HashSet<>();

for (int[] block : blocked){

set.add(block[0] + ":" + block[1]);

}

Queue<int[]> queue = new LinkedList<>();

int N = (int)Math.pow(10, 6);

Set<String> seen = new HashSet<>();

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

queue.offer(new int[]{source[0], source[1]});

seen.add(source[0] + ":" + source[1]);

int step = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

int[] curr = queue.poll();

if (curr[0] == target[0] && curr[1] == target[1]){

return true;

}

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

String key = nx + ":" + ny;

if (nx >= 0 && nx < N && ny >= 0 && ny < N && !seen.contains(key) && !set.contains(key)){

queue.offer(new int[]{nx, ny});

seen.add(key);

}

}

}

step++;

if (step >= maxStep){

return true;

}

}

return false;

}

}

# 1037\_Valid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A boomerang is a set of 3 points that are all distinct and not in a straight line.

Given a list of three points in the plane, return whether these points are a boomerang.

Example 1:

Input: [[1,1],[2,3],[3,2]]

Output: true

Example 2:

Input: [[1,1],[2,2],[3,3]]

Output: false

Note:

points.length == 3

points[i].length == 2

0 <= points[i][j] <= 100

class Solution {

public boolean isBoomerang(int[][] points) {

return (points[1][1] - points[0][1]) \* (points[2][0] - points[0][0]) !=

(points[2][1] - points[0][1]) \* (points[1][0] - points[0][0]);

}

}

# 1038\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the root of a binary search tree with distinct values, modify it so that every node has a new value equal to the sum of the values of the original tree that are greater than or equal to node.val.

As a reminder, a binary search tree is a tree that satisfies these constraints:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

Example 1:

Input: [4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]

Output: [30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]

Note:

The number of nodes in the tree is between 1 and 100.

Each node will have value between 0 and 100.

The given tree is a binary search tree.

Method 1:

with global variable

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

int sum = 0;

public TreeNode bstToGst(TreeNode root) {

reverseInorder(root);

return root;

}

private void reverseInorder(TreeNode root){

if (root == null){

return;

}

reverseInorder(root.right);

sum += root.val;

root.val = sum;

reverseInorder(root.left);

}

}

Method 2:

without global variable

class Solution {

public TreeNode bstToGst(TreeNode root) {

int[] sum = new int[1];

reverseInorder(root, sum);

return root;

}

private void reverseInorder(TreeNode root, int[] sum){

if (root == null){

return;

}

reverseInorder(root.right, sum);

sum[0] += root.val;

root.val = sum[0];

reverseInorder(root.left, sum);

}

}

1039\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given N, consider a convex N-sided polygon with vertices labelled A[0], A[i], ..., A[N-1] in clockwise order.

Suppose you triangulate the polygon into N-2 triangles. For each triangle, the value of that triangle is the product of the labels of the vertices, and the total score of the triangulation is the sum of these values over all N-2 triangles in the triangulation.

Return the smallest possible total score that you can achieve with some triangulation of the polygon.

Example 1:

Input: [1,2,3]

Output: 6

Explanation: The polygon is already triangulated, and the score of the only triangle is 6.

Example 2:

Input: [3,7,4,5]

Output: 144

Explanation: There are two triangulations, with possible scores: 3\*7\*5 + 4\*5\*7 = 245, or 3\*4\*5 + 3\*4\*7 = 144. The minimum score is 144.

Example 3:

Input: [1,3,1,4,1,5]

Output: 13

Explanation: The minimum score triangulation has score 1\*1\*3 + 1\*1\*4 + 1\*1\*5 + 1\*1\*1 = 13.

Note:

3 <= A.length <= 50

1 <= A[i] <= 100

Without loss of generality, there is a triangle that uses adjacent vertices A[0] and A[N-1]

(where N = A.length). Depending on your choice K of it, this breaks down the triangulation into

two subproblems A[1:K] and A[K+1:N-1].

class Solution {

public int minScoreTriangulation(int[] A) {

int N = A.length;

int[][] dp = new int[N][N];

for (int d = 2; d < N; d++){

for (int i = 0; i + d < N; i++){

int j = i + d;

dp[i][j] = Integer.MAX\_VALUE;

for (int k = i + 1; k < j; k++){

dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k][j] + A[i] \* A[k] \* A[j]);

}

}

}

return dp[0][N-1];

}

}

# 107\_WordBreak.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s and a dictionary of words dict, determine if s can be break into a space-separated sequence of

one or more dictionary words.

Have you met this question in a real interview? Yes

Example

Given s = "lintcode", dict = ["lint", "code"].

Return true because "lintcode" can be break as "lint code".

Tags

String Dynamic Programming

public class Solution {

/\*

\* @param s: A string

\* @param dict: A dictionary of words dict

\* @return: A boolean

\*/

public boolean wordBreak(String s, Set<String> dict) {

if (s == null || s.length() == 0){

return true;

}

int len = s.length();

boolean[] canBreak = new boolean[len + 1];

canBreak[0] = true;

for (int i = 1; i <= len; i++){

for (int lastWord = 1; lastWord <= i && lastWord <= getMaxLen(dict); lastWord++){

if (!canBreak[i-lastWord]){

continue;

}

String str = s.substring(i-lastWord, i);

if (dict.contains(str)){

canBreak[i] = true;

break;

}

}

}

return canBreak[len];

}

private int getMaxLen(Set<String> dict){

int max = Integer.MIN\_VALUE;

for (String s : dict){

max = Math.max(max, s.length());

}

return max;

}

}

Method 2:

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

boolean[] dp = new boolean[s.length() + 1];

dp[0] = true;

for (int i = 1; i <= s.length(); i++){

for (int j = 0; j < i; j++){

if (dp[j] && wordDict.contains(s.substring(j, i))){

dp[i] = true;

break;

}

}

}

return dp[s.length()];

}

}

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

Set<String> set = new HashSet<>();

int min = Integer.MAX\_VALUE;

for (String word : wordDict){

set.add(word);

min = Math.min(min, word.length());

}

int n = s.length();

boolean[] dp = new boolean[n+1];

dp[0] = true;

for (int i = min; i <= n; i++){

for (int j = i - min; j >= 0; j--){

if (set.contains(s.substring(j,i))){

dp[i] = dp[i] || dp[j] ;

}

}

}

return dp[n];

}

}

Best solution

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

Set<String> set = new HashSet<>();

int min = Integer.MAX\_VALUE;

for (String word : wordDict){

set.add(word);

min = Math.min(min, word.length());

}

int n = s.length();

boolean[] dp = new boolean[n+1];

dp[0] = true;

for (int i = min; i <= n; i++){

for (int j = i - min; j >= 0; j--){

if (set.contains(s.substring(j,i)) && dp[j]){

dp[i] = true;

break;

}

}

}

return dp[n];

}

}

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

Set<String> set = new HashSet<>();

int min = Integer.MAX\_VALUE;

for (String word : wordDict){

set.add(word);

min = Math.min(min, word.length());

}

int n = s.length();

boolean[] dp = new boolean[n+1];

dp[0] = true;

for (int i = min; i <= n; i++){

for (int j = 0; j <= i - min; j++){

if (set.contains(s.substring(j,i)) && dp[j]){

dp[i] = true;

break;

}

}

}

return dp[n];

}

}

## 140\_WordBreakII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, add spaces

in s to construct a sentence where each word is a valid dictionary word. You may assume the dictionary

does not contain duplicate words.

Return all such possible sentences.

For example, given

s = "catsanddog",

dict = ["cat", "cats", "and", "sand", "dog"].

A solution is ["cats and dog", "cat sand dog"].

DP = recursion + memorization

Method 1: DP, the same as Methodd 2

class Solution {

Map<String, List<String>> map = new HashMap<>();

public List<String> wordBreak(String s, List<String> wordDict) {

List<String> result = new ArrayList<>();

if (s == null || s.length() == 0){

return result;

}

if (map.containsKey(s)){

return map.get(s);

}

if (wordDict.contains(s)){//must have, like initialization

result.add(s);

}

for (int i = 1; i < s.length(); i++){

String sub = s.substring(i);

if (wordDict.contains(sub)){

List<String> temp = wordBreak(s.substring(0, i), wordDict);

if (temp.size() != 0){

for (String str : temp){//if size == 0, won't go into loop

result.add(str +" " + sub);

}

}

}

}

map.put(s, result);

return result;

}

}

Method 2: DP: best solution

note the difference with Leetcode 131 Palindrome Partition

https://github.com/optimisea/Leetcode/blob/master/Java/131\_PalindromePartitioning.java

In this case: the return result could be empty due to no valid break.

class Solution {

public List<String> wordBreak(String s, List<String> wordDict) {

Map<String, List<String>> map = new HashMap<>();

Set<String> set = new HashSet<>();

for (String str : wordDict){

set.add(str);

}

return dfs(s, set, map);

}

private List<String> dfs(String s, Set<String> set, Map<String, List<String>> map){

if (map.containsKey(s)){

return map.get(s);

}

List<String> res = new ArrayList<>();

if (set.contains(s)){

res.add(s);

}

for (int i = 1; i < s.length(); i++){

String sub = s.substring(0, i);

if (set.contains(sub)){

List<String> temp = dfs(s.substring(i), set, map);

// if (temp.size() != 0){

for (String str : temp){

res.add(sub + " " + str);

}

// }

}

}

map.put(s, res);

return res;

}

}

Method 3: Backtracking TLE

class Solution {

public List<String> wordBreak(String s, List<String> wordDict) {

List<String> res = new ArrayList<>();

Set<String> set = new HashSet<>();

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;

for (String st : wordDict){

set.add(st);

min = Math.min(min, st.length());

max = Math.max(max, st.length());

}

backtrack(res, new ArrayList<String>(), s, set, min, max, 0);

return res;

}

private void backtrack(List<String> res, List<String> item, String s, Set<String> set, int min, int max, int pos){

if (pos == s.length()){

String str = String.join(" ", item);

res.add(str);

return;

}

// System.out.println(pos);

for (int i = pos; i <= s.length(); i++){

if (i - pos < min){

continue;

}

if (i - pos > max){

break;

}

String sub = s.substring(pos, i);

// System.out.println(sub);

if (set.contains(sub)){

item.add(sub);

backtrack(res, item, s, set, min, max, i);

item.remove(item.size() - 1);

}

}

}

}

class Solution {

public List<String> wordBreak(String s, List<String> wordDict) {

List<String> res = new ArrayList<>();

Set<String> set = new HashSet<>();

int min = 0;

int max = 0;

for (String str : wordDict){

set.add(str);

min = Math.min(min, str.length());

max = Math.max(max, str.length());

}

dfs(res, s, "", set, 0, min, max);

return res;

}

private void dfs(List<String> res, String s, String item, Set<String> set, int start, int min, int max){

if (start == s.length()){

res.add(item.trim());

return;

}

for (int i = start; i < s.length(); i++){

String sub = s.substring(start, i+1);

if (sub.length() < min){

continue;

}

if (sub.length() > max){

break;

}

if (set.contains(sub)){

dfs(res, s, item + " " + sub, set, i+1, min, max);

}

}

}

}

Best solution:

class Solution {

public List<String> wordBreak(String s, List<String> wordDict) {

Set<String> set = new HashSet<>();

for (String str : wordDict){

set.add(str);

}

Map<String, List<String>> map = new HashMap<>();

return dfs(s, set, map);

}

private List<String> dfs(String s, Set<String> set, Map<String, List<String>> map){

if (map.containsKey(s)){

return map.get(s);

}

List<String> res = new ArrayList<>();

if (set.contains(s)){

res.add(s);

}

for (int i = 1; i < s.length(); i++){

String sub = s.substring(0, i);

if (set.contains(sub)){

List<String> list = dfs(s.substring(i), set, map);

for (String str : list){

res.add(sub + " " + str);

}

}

}

map.put(s, res);

return res;

}

}

# 109\_Triangle.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

Notice

Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

Have you met this question in a real interview? Yes

Example

Given the following triangle:

[

[2],

[3,4],

[6,5,7],

[4,1,8,3]

]

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

public class Solution {

/\*

\* @param triangle: a list of lists of integers

\* @return: An integer, minimum path sum

\*/

public int minimumTotal(int[][] triangle) {

int n = triangle.length;

int[][] f = new int[n][n];

f[0][0] = triangle[0][0];

for (int i = 1; i < n; i++){

f[i][0] = f[i-1][0] + triangle[i][0];

f[i][i] = f[i-1][i-1] + triangle[i][i];

}

for (int i = 1; i < n; i++){

for (int j = 1; j < i ; j++){

f[i][j] = Math.min(f[i-1][j-1], f[i-1][j]) + triangle[i][j];

}

}

int ans = Integer.MAX\_VALUE;

for (int i = 0; i < n; i++){

ans = Math.min(ans, f[n-1][i]);

}

return ans;

}

}

# 111\_ClimbingStairs.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Have you met this question in a real interview? Yes

Example

Given an example n=3 , 1+1+1=2+1=1+2=3

return 3

public class Solution {

/\*\*

\* @param n: An integer

\* @return: An integer

\*/

public int climbStairs(int n) {

if (n < 2){

return n;

}

int[] f = new int[n+1];

f[0] = 0;

f[1] = 1;

f[2] = 2;

for (int i = 3; i <=n; i++){

f[i] = f[i-1] + f[i-2];

}

return f[n];

}

}

class Solution {

public int climbStairs(int n) {

int[] dp = new int[n+1];

dp[0] = 1;

dp[1] = 1;

for (int i = 2; i <= n; i++){

dp[i] = dp[i-1] + dp[i-2];

}

return dp[n];

}

}

# 115\_Distinct.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S and a string T, count the number of distinct subsequences of S which equals T.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters

without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

Example 1:

Input: S = "rabbbit", T = "rabbit"

Output: 3

Explanation:

As shown below, there are 3 ways you can generate "rabbit" from S.

(The caret symbol ^ means the chosen letters)

rabbbit

^^^^ ^^

rabbbit

^^ ^^^^

rabbbit

^^^ ^^^

Example 2:

Input: S = "babgbag", T = "bag"

Output: 5

Explanation:

As shown below, there are 5 ways you can generate "bag" from S.

(The caret symbol ^ means the chosen letters)

babgbag

^^ ^

babgbag

^^ ^

babgbag

^ ^^

babgbag

^ ^^

babgbag

^^^

Method: Dynamic Programming

class Solution {

public int numDistinct(String s, String t) {

int m = t.length();

int n = s.length();

int[][] dp = new int[m+1][n+1];//the number of cases before and including index j of String t for index i of String t

for (int j = 0; j <= n; j++){

dp[0][j] = 1;

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (t.charAt(i-1) == s.charAt(j-1)){

dp[i][j] = dp[i-1][j-1] + dp[i][j-1];

}else{

dp[i][j] = dp[i][j-1];

}

}

}

return dp[m][n];

}

}

class Solution {

public int numDistinct(String s, String t) {

int m = t.length();

int n = s.length();

int[][] dp = new int[m+1][n+1];

for (int j = 0; j <= n; j++){

dp[0][j] = 1;

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (t.charAt(i-1) == s.charAt(j-1)){

dp[i][j] += dp[i-1][j-1];

}

dp[i][j] += dp[i][j-1];

}

}

return dp[m][n];

}

}

Best solution:

dp[i][j] denotes the number of distinct subsequence with the first i substring from s and the first j substring from t

when s.charAt(i-1) == t.charAt(j-1).

Two options: if include s.charAt(i-1), then dp[i][j] += dp[i-1][j-1]

if not include s.charAt(i-1), then dp[i][j] += dp[i-1][j]

when s.charAt(i-1) != t.charAt(j-1)

only one option: not include s.charAt(i-1), then dp[i][j] = dp[i-1][j];

class Solution {

public int numDistinct(String s, String t) {

int m = s.length();

int n = t.length();

int[][] dp = new int[m+1][n+1];

for (int i = 0; i <= m; i++){

dp[i][0] = 1;//there is only one way to match empty string t

}

for (int i = 1; i <= m; i++){

for (int j =1; j <= n; j++){

if (s.charAt(i-1) == t.charAt(j-1)){

dp[i][j] += dp[i-1][j-1];

}

dp[i][j] += dp[i-1][j];

}

}

return dp[m][n];

}

}

class Solution {

public int numDistinct(String s, String t) {

int m = s.length();

int n = t.length();

int[][] dp = new int[m+1][n+1];

for (int i = 0; i <= m; i++){

dp[i][0] = 1;

}

for (int i = 1; i <= m; i++){

for (int j =1; j <= n; j++){

if (s.charAt(i-1) == t.charAt(j-1)){

dp[i][j] = dp[i-1][j-1] + dp[i-1][j];

}else{

dp[i][j] = dp[i-1][j];

}

}

}

return dp[m][n];

}

}

# 116\_JumpGame.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

Notice

This problem have two method which is Greedy and Dynamic Programming.

The time complexity of Greedy method is O(n).

The time complexity of Dynamic Programming method is O(n^2).

We manually set the small data set to allow you pass the test in both ways. This is just to let you learn

how to use this problem in dynamic programming ways. If you finish it in dynamic programming ways,

you can try greedy method to make it accept again.

Method 1:

dynamic programming

Time complexity: O(n^2)

public class Solution {

/\*

\* @param A: A list of integers

\* @return: A boolean

\*/

public boolean canJump(int[] A) {

if (A == null || A.length == 0){

return false;

}

boolean[] can = new boolean[A.length];

can[0] = true;

for (int i = 1; i < A.length; i++){

for (int j = 0; j <i; j++){

if (can[j] && j + A[j] >= i){

can[i] = true;

break;

}

}

}

return can[A.length - 1];

}

}

Method 2: Greedy

Time complexity: O(n)

class Solution {

public boolean canJump(int[] nums) {

if (nums == null || nums.length == 0){

return false;

}

int max = 0;

for (int i = 0; i < nums.length; i++){

if (max < i){

return false;

}

max = Math.max(max, i + nums[i]);

}

return true;

}

}

class Solution {

public boolean canJump(int[] nums) {

int farthest = 0;

for (int i = 0; i < nums.length; i++){

if (i > farthest){

return false;

}

farthest = Math.max(farthest, i + nums[i]);

if (farthest >= nums.length - 1){

return true;

}

}

return false;

}

}

# 116\_Populating.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree

struct TreeLinkNode {

TreeLinkNode \*left;

TreeLinkNode \*right;

TreeLinkNode \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Note:

You may only use constant extra space.

You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,

Given the following perfect binary tree,

1

/ \

2 3

/ \ / \

4 5 6 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ / \

4->5->6->7 -> NULL

Method 1: iteration LinkedList

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void connect(TreeLinkNode root) {

if (root == null){

return;

}

TreeLinkNode head = root;

while (head.left != null && head.right != null){

TreeLinkNode cur = head;

while (cur != null){

cur.left.next = cur.right;

cur.right.next = cur.next == null ? null : cur.next.left;

cur = cur.next;

}

head = head.left;

}

}

}

Method 2: recursion Tree

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void connect(TreeLinkNode root) {

if (root == null){

return;

}

connect(root.left, root.right);

}

private void connect(TreeLinkNode node1, TreeLinkNode node2){

if (node1 != null && node2 != null){

node1.next = node2;

connect(node1.left, node1.right);

connect(node2.left, node2.right);

connect(node1.right, node2.left);

}

}

}

Method 3: BFS Tree

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void connect(TreeLinkNode root) {

if (root == null){

return;

}

Queue<TreeLinkNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

TreeLinkNode node = queue.poll();

if (i == size - 1){

node.next = null;

}else{

node.next = queue.peek();

}

if (node.left != null){

queue.offer(node.left);

}

if (node.right != null){

queue.offer(node.right);

}

}

}

}

}

Method 4: 117. general form

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void connect(TreeLinkNode root) {

if (root == null){

return;

}

TreeLinkNode head = root;

while (head != null){

TreeLinkNode curr = head;

TreeLinkNode childDummy = new TreeLinkNode(-1);

TreeLinkNode child = childDummy;

while (curr != null){

if (curr.left != null){

child.next = curr.left;

child = child.next;

}

if (curr.right != null){

child.next = curr.right;

child = child.next;

}

curr = curr.next;

}

head = childDummy.next;

}

}

}

# 117\_JumpGameII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

Have you met this question in a real interview? Yes

Example

Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

Method 1: like Coin Change O(n^2)

public class Solution {

/\*

\* @param A: A list of integers

\* @return: An integer

\*/

public int jump(int[] A) {

if (A== null || A.length == 0){

return 0;

}

int[] minSteps = new int[A.length];

for (int i = 0; i < A.length; i++){

minSteps[i] = Integer.MAX\_VALUE;

}

minSteps[0] = 0;

for (int i = 1; i < A.length; i++){

for (int j = 0; j < i; j++){

if (minSteps[j] != Integer.MAX\_VALUE && A[j] + j >= i){

minSteps[i] = Math.min(minSteps[i], minSteps[j] + 1);

}

}

}

return minSteps[A.length - 1];

}

}

Method 2: Like BFS O(n)

http://www.allenlipeng47.com/blog/index.php/2016/09/12/jump-game-ii/

class Solution {

public int jump(int[] nums) {

int level = 0;

int nextMaxIndex = 0;

int currMaxIndex = 0;

for (int i = 0; i < nums.length; i++){

if (currMaxIndex >= nums.length - 1){

return level;

}

nextMaxIndex = Math.max(nextMaxIndex, i + nums[i]);

if (i == currMaxIndex){

level++;

currMaxIndex = nextMaxIndex;

}

}

return 0;

}

}

Method 3: Greedy O(n) like Jum Game I

The main idea is based on greedy. Let's say the range of the current jump is [curBegin, curEnd], curFarthest is the farthest point

that all points in [curBegin, curEnd] can reach. Once the current point reaches curEnd, then trigger another jump,

and set the new curEnd with curFarthest, then keep the above steps, as the following:

class Solution {

public int jump(int[] nums) {

int steps = 0;

int curBegin = 0;

int curEnd = 0;

int curFarthest = 0;

while (curFarthest < nums.length - 1){

for (int i = curBegin; i <= curEnd; i++){

curFarthest = Math.max(curFarthest, i + nums[i]);

}

steps++;

curBegin = curEnd+1;

curEnd = curFarthest;

}

return steps;

}

}

class Solution {

public int jump(int[] nums) {

int minStep = 0;

int currStart = 0;

int currEnd = 0;

int currFarthest = 0;

boolean canReach = true;

while (currFarthest < nums.length - 1){

for (int i = currStart; i <= currEnd; i++){

currFarthest = Math.max(currFarthest, i + nums[i]);

}

if (currFarthest > currEnd){

minStep++;

currStart = currEnd;

currEnd = currFarthest;

}else{//can't reach the last index

canReach = false;

break;

}

}

return canReach ? minStep : -1;

}

}

# 117\_Populating.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Follow up for problem "Populating Next Right Pointers in Each Node".

What if the given tree could be any binary tree? Would your previous solution still work?

Note:

You may only use constant extra space.

For example,

Given the following binary tree,

1

/ \

2 3

/ \ \

4 5 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ \

4-> 5 -> 7 -> NULL

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

Time complexity: O(n)

Space complexity: O(1)

Level order traversal

head: the leftmost node of the current level

cur: traversal node of the current level

dummyChildHead: the dummy head of the next level

child: traversal node of the next level

public class Solution {

public void connect(TreeLinkNode root) {

if (root == null){

return;

}

TreeLinkNode head = root;

TreeLinkNode dummyChildHead = new TreeLinkNode(-1);

while (head != null){

TreeLinkNode cur = head;

TreeLinkNode child = dummyChildHead;

while (cur != null){

if (cur.left != null){

child.next = cur.left;

child = child.next;

}

if (cur.right != null){

child.next = cur.right;

child = child.next;

}

cur = cur.next;

}

head = dummyChildHead.next;

dummyChildHead.next = null;

}

}

}

Better version:

public class Solution {

public void connect(TreeLinkNode root) {

if (root == null){

return;

}

TreeLinkNode head = root;

while (head != null){

TreeLinkNode curr = head;

TreeLinkNode childDummy = new TreeLinkNode(-1);

TreeLinkNode child = childDummy;

while (curr != null){

if (curr.left != null){

child.next = curr.left;

child = child.next;

}

if (curr.right != null){

child.next = curr.right;

child = child.next;

}

curr = curr.next;

}

head = childDummy.next;

}

}

}

# 118\_Pascal\_Triangle.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given numRows, generate the first numRows of Pascal's triangle.

For example, given numRows = 5,

Return

[

[1],

[1,1],

[1,2,1],

[1,3,3,1],

[1,4,6,4,1]

]

Method 1:

DP

Time complexity: O(n^2)

Space complexity: O(n^2)

class Solution {

public List<List<Integer>> generate(int numRows) {

List<List<Integer>> result = new ArrayList<>();

int[][] dp = new int[numRows+1][numRows+1];

for (int i = 1; i <= numRows; i++){

List<Integer> item = new ArrayList<>();

for (int j = 1; j <= i; j++){

if (j == 1 || j == i){

dp[i][j] = 1;

}else{

dp[i][j] = dp[i-1][j-1] + dp[i-1][j];

}

item.add(dp[i][j]);

}

result.add(item);

}

return result;

}

}

Method 2:

DP + rolling array

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public List<List<Integer>> generate(int numRows) {

List<List<Integer>> result = new ArrayList<>();

int[][] dp = new int[2][numRows+1];

for (int i = 1; i <= numRows; i++){

List<Integer> item = new ArrayList<>();

for (int j = 1; j <= i; j++){

if (j == 1 || j == i){

dp[i%2][j] = 1;

}else{

dp[i%2][j] = dp[(i-1)%2][j-1] + dp[(i-1)%2][j];

}

item.add(dp[i%2][j]);

}

result.add(item);

}

return result;

}

}

class Solution {

public List<List<Integer>> generate(int numRows) {

List<List<Integer>> res = new ArrayList<>();

if (numRows == 0){

return res;

}

List<Integer> l = new ArrayList<>();

l.add(1);

res.add(l);

if (numRows == 1){

return res;

}

l = new ArrayList<>();

l.add(1);

l.add(1);

res.add(l);

if (numRows == 2){

return res;

}

for (int i = 2; i < numRows; i++){

List<Integer> list = new ArrayList<>();

list.add(1);

for (int j = 1; j < i; j++){

list.add(res.get(i-1).get(j-1) + res.get(i-1).get(j));

}

list.add(1);

res.add(list);

}

return res;

}

}

# 119\_Pascal'sTriangleII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an index k, return the kth row of the Pascal's triangle.

For example, given k = 3,

Return [1,3,3,1].

Note:

Could you optimize your algorithm to use only O(k) extra space?

Method 1:

Time complexity: O(k^2)

Space complexity: O(1)

class Solution {

public List<Integer> getRow(int rowIndex) {

List<Integer> result = new ArrayList<>();

if (rowIndex < 0){

return result;

}

for (int i = 0; i <= rowIndex; i++){

result.add(1);

for (int j = i - 1; j > 0; j--){

result.set(j, result.get(j) + result.get(j-1));

}

}

return result;

}

}

Method 2:

Time complexity: O(k^2)

Space complexity: O(2\*k);

class Solution {

public List<Integer> getRow(int rowIndex) {

List<Integer> result = new ArrayList<>();

int[][] f = new int[2][rowIndex+1];

f[0][0] = 1;

for (int i = 1; i <= rowIndex ; i++){

for (int j = 0; j <= rowIndex; j++){

if (j == 0 || j == rowIndex){

f[i%2][j] = 1;

}else{

f[i%2][j] = f[(i-1)%2][j-1] + f[(i-1)%2][j];

}

}

}

for (int i = 0; i <= rowIndex; i++){

result.add(f[rowIndex%2][i]);

}

return result;

}

}

class Solution {

public List<Integer> getRow(int rowIndex) {

List<Integer> res = new ArrayList<>();

int[][] dp = new int[2][rowIndex+1];

for (int i = 0; i <= rowIndex; i++){

for (int j = 0; j <= i; j++){

if (j == 0 || j == i){

dp[i%2][j] = 1;

}else{

dp[i%2][j] = dp[(i-1)%2][j-1] + dp[(i-1)%2][j];

}

if (i == rowIndex){

res.add(dp[i%2][j]);

}

}

}

return res;

}

}

# 125\_BackpackII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n items with size Ai and value Vi, and a backpack with size m. What's the maximum value can you put into the backpack?

Notice

You cannot divide item into small pieces and the total size of items you choose should smaller or equal to m.

Have you met this question in a real interview?

Example

Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2, 4], and a backpack with size 10. The maximum value is 9.

Challenge

O(n x m) memory is acceptable, can you do it in O(m) memory?

State: f[i][j]

Function: f[i][j] = max(f[i-1][j], f[i-1][j-A[i-1]] + V[i-1]);

Intialization • f[0][0]=0;

Answer • f[n][s]

Time complexity: O(n\*m)

Space complexity: O(m)

public class Solution {

/\*\*

\* @param m: An integer m denotes the size of a backpack

\* @param A: Given n items with size A[i]

\* @param V: Given n items with value V[i]

\* @return: The maximum value

\*/

public int backPackII(int m, int[] A, int[] V) {

int n = A.length;

int[][] dp = new int[2][m+1];

for (int i = 1; i <= n; i++){

for (int j = 1; j <= m; j++){

if (j >= A[i-1]){

dp[i%2][j] = Math.max(dp[(i-1)%2][j], dp[(i-1)%2][j-A[i-1]] + V[i-1]);

}else{

dp[i%2][j] = dp[(i-1)%2][j];

}

}

}

int max = 0;

for (int i = m; i >= 0; i--){

max = Math.max(max, dp[n%2][i]);

}

return max;

}

}

# 126\_Word.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two words (beginWord and endWord), and a dictionary's word list, find all shortest transformation sequence(s) from

beginWord to endWord, such that:

Only one letter can be changed at a time

Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

Note:

Return an empty list if there is no such transformation sequence.

All words have the same length.

All words contain only lowercase alphabetic characters.

You may assume no duplicates in the word list.

You may assume beginWord and endWord are non-empty and are not the same.

Example 1:

Input:

beginWord = "hit",

endWord = "cog",

wordList = ["hot","dot","dog","lot","log","cog"]

Output:

[

["hit","hot","dot","dog","cog"],

["hit","hot","lot","log","cog"]

]

Example 2:

Input:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]

Output: []

Explanation: The endWord "cog" is not in wordList, therefore no possible transformation.

TLE

class Solution {

public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

List<List<String>> res = new ArrayList<>();

Set<String> wordSet = new HashSet<>(wordList);

Set<String> seen = new HashSet<>();

List<String> item = new ArrayList<String>();

int min = bfs(wordSet, beginWord, endWord);

System.out.println(min);

seen.add(beginWord);

item.add(beginWord);

dfs(res, item, wordSet, seen, beginWord, endWord, min, 0);

return res;

}

private int bfs(Set<String> wordSet, String beginWord, String endWord){

Queue<String> queue = new LinkedList<>();

Set<String> seen = new HashSet<>();

queue.offer(beginWord);

seen.add(beginWord);

int res = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int j = 0; j < size; j++){

String curr = queue.poll();

if (curr.equals(endWord)){

return res;

}

char[] currArr = curr.toCharArray();

for (int i = 0; i < curr.length(); i++){

char ch = currArr[i];

for (char c = 'a'; c <= 'z'; c++){

if (ch != c){

currArr[i] = c;

String str = new String(currArr);

if (wordSet.contains(str) && !seen.contains(str)){

queue.offer(str);

seen.add(str);

}

currArr[i] = ch;

}

}

}

}

res++;

}

return res;

}

private void dfs(List<List<String>> res, List<String> item, Set<String> wordSet, Set<String> seen, String begin, String end, int min, int steps){

if (steps == min){

if (begin.equals(end)){

res.add(new ArrayList<>(item));

}

return;

}

for (int i = 0; i < begin.length(); i++){

if (steps >= min){

break;

}

char[] charArr = begin.toCharArray();

char c = charArr[i];

for (char ch = 'a'; ch <= 'z'; ch++){

if (ch != c){

charArr[i] = ch;

String next = new String(charArr);

if (wordSet.contains(next) && !seen.contains(next)){

seen.add(next);

item.add(next);

dfs(res, item, wordSet, seen, next, end, min, steps + 1);

seen.remove(next);

item.remove(item.size() - 1);

}

charArr[i] = c;

}

}

}

}

}

TLE

class Solution {

public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {

List<List<String>> res = new ArrayList<>();

Set<String> wordSet = new HashSet<>(wordList);

Set<String> seen = new HashSet<>();

List<String> item = new ArrayList<String>();

seen.add(beginWord);

item.add(beginWord);

dfs(res, item, wordSet, seen, beginWord, endWord);

int min = Integer.MAX\_VALUE;

for(List<String> cand : res){

int size = cand.size();

if (size < min){

min = size;

}

}

List<List<String>> result = new ArrayList<>();

for (List<String> cand : res){

int size = cand.size();

if (size == min){

result.add(cand);

}

}

return result;

}

private void dfs(List<List<String>> res, List<String> item, Set<String> wordSet, Set<String> seen, String begin, String end){

if (begin.equals(end)){

res.add(new ArrayList<>(item));

return;

}

for (int i = 0; i < begin.length(); i++){

char[] charArr = begin.toCharArray();

char c = charArr[i];

for (char ch = 'a'; ch <= 'z'; ch++){

if (ch == c){

continue;

}

charArr[i] = ch;

String next = new String(charArr);

if (wordSet.contains(next) && !seen.contains(next)){

seen.add(next);

item.add(next);

dfs(res, item, wordSet, seen, next, end);

seen.remove(next);

item.remove(item.size() - 1);

}

}

}

}

}

# 127\_WordLadder.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two words (beginWord and endWord), and a dictionary's word list, find the length of shortest transformation sequence

from beginWord to endWord, such that:

Only one letter can be changed at a time.

Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

For example,

Given:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log","cog"]

As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",

return its length 5.

Note:

Return 0 if there is no such transformation sequence.

All words have the same length.

All words contain only lowercase alphabetic characters.

You may assume no duplicates in the word list.

You may assume beginWord and endWord are non-empty and are not the same.

UPDATE (2017/1/20):

The wordList parameter had been changed to a list of strings (instead of a set of strings).

Please reload the code definition to get the latest changes.

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

if (wordList == null || wordList.size() == 0 || !wordList.contains(endWord)){

return 0;

}

if (beginWord.equals(endWord)){

return 1;

}

int result = 1;

Queue<String> queue = new LinkedList<>();

Set<String> set = new HashSet<>();

queue.offer(beginWord);

set.add(beginWord);

while (!queue.isEmpty()){

int size = queue.size();

result++;

for (int i = 0; i < size; i++){

String str = queue.poll();

for(String word : getNextWord(wordList, str)){

if (word.equals(endWord)){

return result;

}

if (!set.contains(word)){

queue.offer(word);

set.add(word);

}

}

}

}

return 0;

}

private List<String> getNextWord(List<String> wordList, String str){

Set<String> set = new HashSet<>(wordList); //convert list to set so that contains function takes O(1) instead of O(n)

List<String> result = new ArrayList<>();

for (int i = 0; i < str.length(); i++){

for (char j = 'a'; j <= 'z'; j++){

if (j != str.charAt(i)){

String word = replace(str, j, i);

if (set.contains(word)){

result.add(word);

}

}

}

}

return result;

}

private String replace(String str, char j, int i){

char[] charArray = str.toCharArray();

charArray[i] = j;

// return String.valueOf(charArray);

return new String(charArray);

}

}

Combined to one function

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

if (wordList == null || wordList.size() == 0 || !wordList.contains(endWord)){

return 0;

}

if (beginWord.equals(endWord)){

return 1;

}

int result = 1;

Queue<String> queue = new LinkedList<>();

Set<String> set = new HashSet<>();

queue.offer(beginWord);

set.add(beginWord);

Set<String> wordSet = new HashSet<>(wordList);

while (!queue.isEmpty()){

int size = queue.size();

result++;

for (int k = 0; k < size; k++){

String str = queue.poll();

for (int i = 0; i < str.length(); i++){

char[] charArray = str.toCharArray();

for (char j = 'a'; j <= 'z'; j++){

if (str.charAt(i) == j){

continue;

}

charArray[i] = j;

String word = new String(charArray);

if (word.equals(endWord)){

return result;

}

if (wordSet.contains(word) && !set.contains(word)){

queue.offer(word);

set.add(word);

}

}

}

}

}

return 0;

}

}

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

Queue<String> queue = new LinkedList<>();

Set<String> seen = new HashSet<>();

queue.offer(beginWord);

seen.add(beginWord);

Set<String> wordSet = new HashSet<>(wordList);

int res = 0;

while (!queue.isEmpty()){

res++;

int size = queue.size();

for (int i = 0; i < size; i++){

String curr = queue.poll();

if (curr.equals(endWord)){

return res;

}

char[] charArray = curr.toCharArray();

for (int j = 0; j < curr.length(); j++){

for (char ch = 'a'; ch <= 'z'; ch++){

char c = charArray[j];

if (ch != c){

charArray[j] = ch;

String str = new String(charArray);

if (!seen.contains(str) && wordSet.contains(str)){

queue.offer(str);

seen.add(str);

}

charArray[j] = c;

}

}

}

}

}

return 0;

}

}

Better:

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

int res = 0;

Set<String> set = new HashSet<>();

for (String s : wordList){

set.add(s);

}

Queue<String> queue = new LinkedList<>();

Set<String> seen = new HashSet<>();

queue.offer(beginWord);

seen.add(beginWord);

while (!queue.isEmpty()){

int size = queue.size();

res++;

for (int i = 0; i < size; i++){

String str = queue.poll();

if (endWord.equals(str)){

return res;

}

char[] charArr = str.toCharArray();

for (int j = 0; j < str.length(); j++){

char c = charArr[j];

for (char k = 'a'; k <= 'z'; k++){

if (c == k){

continue;

}

charArr[j] = k;

String cand = new String(charArr);

if (!seen.contains(cand) && set.contains(cand)){

seen.add(cand);

queue.offer(cand);

}

}

charArr[j] = c;

}

}

}

return 0;

}

}

# 129\_SumRootLeafNumbers.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers.

For example,

1

/ \

2 3

The root-to-leaf path 1->2 represents the number 12.

The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

best solution:

preorder traversal

class Solution {

public int sumNumbers(TreeNode root) {

return helper(root, 0);

}

private int helper(TreeNode root, int sum){

if (root == null){

return 0;

}

if (root.left == null && root.right == null){

return sum \* 10 + root.val;

}

sum = sum \* 10 + root.val;

return helper(root.right, sum) + helper(root.left, sum);

}

}

Better solution:

class Solution {

int res = 0;

public int sumNumbers(TreeNode root) {

preorder(root, 0);

return res;

}

private void preorder(TreeNode root, int sum){

if (root == null){

return;

}

if (root.left == null && root.right == null){

res += sum \* 10 + root.val;

return;

}

sum = sum \* 10 + root.val;

preorder(root.left, sum);

preorder(root.right, sum);

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

int res = 0;

public int sumNumbers(TreeNode root) {

if (root == null){

return 0;

}

dfs(root, 0);

return res;

}

private void dfs(TreeNode root, int sum){

if (root.left == null && root.right == null){

res += sum \* 10 + root.val;

return;

}

sum = sum \* 10 + root.val;

if (root.left != null){

dfs(root.left, sum);

}

if (root.right != null){

dfs(root.right, sum);

}

}

}

# 134\_GasStation.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are N gas stations along a circular route, where the amount of gas at station i is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to

its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note:

The solution is guaranteed to be unique.

1, if sum of gas is more than sum of cost, then there must be a solution.

And the question guaranteed that the solution is unique(The first one I found is the right one).

2, The tank should never be negative, so restart whenever there is a negative number.

Method 1: O(N)

class Solution {

public int canCompleteCircuit(int[] gas, int[] cost) {

int start = 0;

int gasSum = 0;

int costSum = 0;

int tank = 0;

for (int i = 0; i < gas.length; i++){

gasSum += gas[i];

costSum += cost[i];

tank += gas[i] - cost[i];

if (tank < 0){

start = i + 1;

tank = 0;

}

}

if (gasSum < costSum){

return -1;

}

return start;

}

}

Method 2: O(N^2)

class Solution {

public int canCompleteCircuit(int[] gas, int[] cost) {

int n = gas.length;

for (int i = 0; i < n; i++){

boolean found = true;

int tank = 0;

for (int j = 0; j < n; j++){

int index = (i + j) % n;

if (gas[index] + tank < cost[index]){

found = false;

break;

}

tank += gas[index] - cost[index];

}

if (found){

return i;

}

}

return -1;

}

}

# 136\_SingleNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers, every element appears twice except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Method:

Use XOR

XOR has Commutative property and associative property

a ^ a = 0, a ^ 0 = a

class Solution {

public int singleNumber(int[] nums) {

int ans = 0;

for (int n : nums){

ans ^= n;

}

return ans;

}

}

Generalization format as Single II

class Solution {

public int singleNumber(int[] nums) {

int res = 0;

for (int i = 0; i < 32; i++){

int sum = 0;

for (int j = 0; j < nums.length; j++){

if (((nums[j] >> i) & 1) == 1){

sum++;

sum %= 2;

}

}

if (sum != 0){

res |= (sum << i);

}

}

return res;

}

}

class Solution {

public int singleNumber(int[] nums) {

int res = 0;

for (int i = 0; i < 32; i++){

int temp = 0;

for (int num : nums){

temp += ((num >> i) & 1);

}

if (temp % 2 == 1){

res |= (1 << i);

}

}

return res;

}

}

# 137\_Single.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty array of integers, every element appears three times except for one, which appears

exactly once. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Example 1:

Input: [2,2,3,2]

Output: 3

Example 2:

Input: [0,1,0,1,0,1,99]

Output: 99

https://leetcode.com/problems/single-number-ii/discuss/43297/Java-O(n)-easy-to-understand-solution-easily-extended-to-any-times-of-occurance

class Solution {

public int singleNumber(int[] nums) {

int ans = 0;

for (int i = 0; i < 32; i++){

int sum = 0;

for (int j = 0; j < nums.length; j++){

if (((nums[j] >> i) & 1) == 1){

sum++;

// sum %= 3;

}

}

sum %= 3;

if (sum != 0){

ans |= sum << i;

}

}

return ans;

}

}

class Solution {

public int singleNumber(int[] nums) {

int res = 0;

for (int i = 0; i < 32; i++){

int temp = 0;

for (int num : nums){

temp += ((num >> i) & 1);

}

if (temp % 3 == 1){

res |= (1 << i);

}

}

return res;

}

}

# 138\_SubarraySum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array, find a subarray where the sum of numbers is zero.

Your code should return the index of the first number and the index of the last number.

Notice

There is at least one subarray that it's sum equals to zero.

Have you met this question in a real interview? Yes

Example

Given [-3, 1, 2, -3, 4], return [0, 2] or [1, 3].

Method 1: best solution

public class Solution {

/\*

\* @param nums: A list of integers

\* @return: A list of integers includes the index of the first number and the index of the last number

\*/

public List<Integer> subarraySum(int[] nums) {

List<Integer> result = new ArrayList<>();

if (nums == null || nums.length == 0){

return result;

}

HashMap<Integer,Integer> map = new HashMap<>();

map.put(0,-1);

int sum = 0;

for (int i = 0; i < nums.length; i++){

sum += nums[i];

if (map.containsKey(sum)){

result.add(map.get(sum) + 1);

result.add(i);

return result;

}

map.put(sum, i);

}

return result;

}

}

Method 2: use prefix Sum

public class Solution {

/\*\*

\* @param nums: A list of integers

\* @return: A list of integers includes the index of the first number and the index of the last number

\*/

public List<Integer> subarraySum(int[] nums) {

List<Integer> result = new ArrayList<>();

int[] pre = new int[nums.length+1];

pre[0] = 0;

for (int i = 1; i <= nums.length; i++){

pre[i] = pre[i-1] + nums[i-1];

}

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < pre.length; i++){

if (map.containsKey(pre[i])){

result.add(map.get(pre[i]));

result.add(i-1);

return result;

}

map.put(pre[i], i);

}

return result;

}

}

# 139\_SubarraySumClosest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array, find a subarray with sum closest to zero. Return the indexes of the first number and last number.

Have you met this question in a real interview? Yes

Example

Given [-3, 1, 1, -3, 5], return [0, 2], [1, 3], [1, 1], [2, 2] or [0, 4].

Challenge

O(nlogn) time

public class Solution {

/\*

\* @param nums: A list of integers

\* @return: A list of integers includes the index of the first number and the index of the last number

\*/

private class Pair{

int sum;

int index;

public Pair(int sum, int index){

this.sum = sum;

this.index = index;

}

}

public int[] subarraySumClosest(int[] nums) {

if (nums == null || nums.length == 0){

return nums;

}

int[] result = new int[2];

if (nums.length == 1){

result[0] = result[1] = 0;

return result;

}

Pair[] preSumPair = new Pair[nums.length + 1];

preSumPair[0] = new Pair(0, 0);

int preSum = 0;

for (int i = 1; i <= nums.length; i++){

preSum = preSum + nums[i - 1];

preSumPair[i] = new Pair(preSum, i);

}

Arrays.sort(preSumPair, new Comparator<Pair>(){

public int compare(Pair a, Pair b){

return a.sum - b.sum;

}

});

int diff = Integer.MAX\_VALUE;

for (int i = 1; i <= nums.length; i++){

if (diff > preSumPair[i].sum - preSumPair[i - 1].sum){

diff = preSumPair[i].sum - preSumPair[i - 1].sum;

result[0] = Math.min(preSumPair[i].index, preSumPair[i - 1].index);

result[1] = Math.max(preSumPair[i].index, preSumPair[i - 1].index) - 1;

}

}

return result;

}

}

# 141\_Sqrt.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement int sqrt(int x).

Compute and return the square root of x.

Have you met this question in a real interview?

Example

sqrt(3) = 1

sqrt(4) = 2

sqrt(5) = 2

sqrt(10) = 3

Challenge

O(log(x))

public class Solution {

/\*\*

\* @param x: An integer

\* @return: The sqrt of x

\*/

public int sqrt(int x) {

long start = 1;

long end = x;

while (start + 1 < end){

long mid = start + (end - start) / 2;

if (mid \* mid == x){

return (int) mid;

}else if (mid \* mid < x){

start = mid;

}else{

end = mid;

}

}

if (end \* end == x){

return (int) end;

}

return (int) start;

}

}

class Solution {

public int mySqrt(int x) {

long start = 1;

long end = x;

while (start <= end){

long mid = start + (end - start) / 2;

if (mid \* mid == x){

return (int)mid;

}else if (mid \* mid < x){

start = mid + 1;

}else {

end = mid - 1;

}

}

return (int) Math.min(start, end);

}

}

# 143\_SortColorsII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of n objects with k different colors (numbered from 1 to k), sort them so that objects of

the same color are adjacent, with the colors in the order 1, 2, ... k.

Notice

You are not suppose to use the library's sort function for this problem.

k <= n

Have you met this question in a real interview? Yes

Example

Given colors=[3, 2, 2, 1, 4], k=4, your code should sort colors in-place to [1, 2, 2, 3, 4].

Challenge

A rather straight forward solution is a two-pass algorithm using counting sort. That will cost O(k) extra memory.

Can you do it without using extra memory?

Rainbow sort

O(Nlogk)

public class Solution {

/\*

\* @param colors: A list of integer

\* @param k: An integer

\* @return: nothing

\*/

public void sortColors2(int[] colors, int k) {

if (colors == null || colors.length == 0){

return;

}

sortColor(colors, 0, colors.length - 1, 1, k);

}

private void sortColor(int[] colors, int start, int end, int colorFrom, int colorTo){

if (start >= end){

return;

}

if (colorFrom == colorTo){

return;

}

int left = start;

int right = end;

int colorMid = (colorFrom + colorTo) / 2;

while (left < right){

while (left < right && colors[left] <= colorMid){

left++;

}

while (left < right && colors[right] > colorMid){

right--;

}

if (left < right){

int temp = colors[left];

colors[left] = colors[right];

colors[right] = temp;

left++;

right--;

}

}

sortColor(colors, start, right, colorFrom, colorMid);

sortColor(colors, left, end, colorMid + 1, colorTo);

}

}

Time complexity: O(nk)

Space complexity: O(1)

https://aaronice.gitbooks.io/lintcode/high\_frequency/sort\_colors\_ii.html

class Solution {

public void sortColors(int[] nums, int k) {

int left = 0;

int right = nums.length - 1;

int i = 0;

int min = 1;

int max = k;

while (min < max){

while (i <= right){

if (nums[i] == min){

swap(nums, i, left);

i++;

left++;

}else if (nums[i] == max){

swap(nums, i, right);

right--;

}else{

i++;

}

}

i = left;

min++;

max--;

}

}

private void swap(int[] nums, int x, int y){

int temp = nums[x];

nums[x] = nums[y];

nums[y] = temp;

}

}

# 144\_InterleavingPositiveandNegativeNumbers.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array with positive and negative integers. Re-range it to interleaving with positive and negative integers.

Notice

You are not necessary to keep the original order of positive integers or negative integers.

Have you met this question in a real interview? Yes

Example

Given [-1, -2, -3, 4, 5, 6], after re-range, it will be [-1, 5, -2, 4, -3, 6] or any other reasonable answer.

public class Solution {

/\*

\* @param A: An integer array.

\* @return: nothing

\*/

public void rerange(int[] A) {

if(A == null || A.length == 0){

return;

}

int start = 0;

int end = A.length - 1;

while (start < end){

while (start < end && A[start] < 0){

start++;

}

while (start < end && A[end] > 0){

end--;

}

if (start < end){

swap(A, start, end);

start++;

end--;

}

}

if (A.length % 2 == 0){

swapPosNeg(A, 0, A.length - 1);

}else{

int mid = A.length / 2;

if (A[mid] % 2 < 0){

swapPosNeg(A, 1, A.length - 1);

}else{

swapPosNeg(A, 0, A.length - 2);

}

}

}

private void swapPosNeg(int[] A, int left, int right){

while (left < right){

swap(A, left, right);

left = left + 2;

right = right - 2;

}

}

private void swap (int[] A, int a, int b){

int temp = A[a];

A[a] = A[b];

A[b] = temp;

}

}

# 149\_Max.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

Example 1:

Input: [[1,1],[2,2],[3,3]]

Output: 3

Explanation:

^

|

| o

| o

| o

+------------->

0 1 2 3 4

Example 2:

Input: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]

Output: 4

Explanation:

^

|

| o

| o o

| o

| o o

+------------------->

0 1 2 3 4 5 6

https://leetcode.com/problems/max-points-on-a-line/discuss/47113/A-java-solution-with-notes

/\*

\* A line is determined by two factors,say y=ax+b

\*

\* If two points(x1,y1) (x2,y2) are on the same line(Of course).

\* Consider the gap between two points.

\* We have (y2-y1)=a(x2-x1),a=(y2-y1)/(x2-x1) a is a rational, b is canceled since b is a constant

\* If a third point (x3,y3) are on the same line. So we must have y3=ax3+b

\* Thus,(y3-y1)/(x3-x1)=(y2-y1)/(x2-x1)=a

\* Since a is a rational, there exists y0 and x0, y0/x0=(y3-y1)/(x3-x1)=(y2-y1)/(x2-x1)=a

\* So we can use y0&x0 to track a line;

\*/

Best Solution:

/\*\*

\* Definition for a point.

\* class Point {

\* int x;

\* int y;

\* Point() { x = 0; y = 0; }

\* Point(int a, int b) { x = a; y = b; }

\* }

\*/

class Solution {

public int maxPoints(Point[] points) {

if (points == null || points.length == 0){

return 0;

}

int res = 0;

for (int i = 0; i < points.length; i++){

Map<String, Integer> map = new HashMap<>();

int dup = 0;

int max = 0;

for (int j = i + 1; j < points.length; j++){

Point p1 = points[i];

Point p2 = points[j];

if (p1.x == p2.x && p1.y == p2.y){

dup++;

continue;

}

int deltaX = p1.x - p2.x;

int deltaY = p1.y - p2.y;

int gcd = gcd(deltaX, deltaY);

int dx = deltaX / gcd;

int dy = deltaY / gcd;

String key = dx + ":" + dy;

map.put(key, map.getOrDefault(key, 0) + 1);

max = Math.max(max, map.get(key));

}

res = Math.max(res, max + dup + 1);//+1 means to include itself

}

return res;

}

private int gcd (int a, int b){

if (b == 0){

return a;

}

return gcd (b, a%b);

}

}

# 150\_EvaluateReversePolishNotation.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, \*, /. Each operand may be an integer or another expression.

Some examples:

["2", "1", "+", "3", "\*"] -> ((2 + 1) \* 3) -> 9

["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

class Solution {

public int evalRPN(String[] tokens) {

Stack<Integer> number = new Stack<>();

int first, second;

for (int i = 0; i < tokens.length; i++){

switch (tokens[i]){

case "+":

first = number.pop();

second = number.pop();

number.push(first + second);

break;

case "-":

first = number.pop();

second = number.pop();

number.push(second - first);

break;

case "\*":

first = number.pop();

second = number.pop();

number.push(first \* second);

break;

case "/":

first = number.pop();

second = number.pop();

number.push(second / first);

break;

default:

number.push(Integer.parseInt(tokens[i]));

}

}

return number.pop();

}

}

class Solution {

public int evalRPN(String[] tokens) {

Stack<String> stack = new Stack<>();

for (String token : tokens){

char c = token.charAt(token.length()-1);

if (Character.isDigit(c)){

stack.push(token);

}else{

int s2 = Integer.parseInt(stack.pop());

int s1 = Integer.parseInt(stack.pop());

if (c == '+'){

stack.push(String.valueOf(s1 + s2));

}else if (c == '-'){

stack.push(String.valueOf(s1 - s2));

}else if (c == '\*'){

stack.push(String.valueOf(s1 \* s2));

}else{

stack.push(String.valueOf(s1 / s2));

}

}

}

return Integer.parseInt(stack.pop());

}

}

# 152\_MaximumProductSubarray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2,3,-2,4],

the contiguous subarray [2,3] has the largest product = 6.

class Solution {

public int maxProduct(int[] nums) {

int maxPre = 1;

int minPre = 1;

int maxCur = nums[0];

int minCur = nums[0];

int result = Integer.MIN\_VALUE;

for (int i = 0; i < nums.length; i++){

maxCur = Math.max(maxPre \* nums[i], minPre \* nums[i]);

minCur = Math.min(maxPre \* nums[i], minPre \* nums[i]);

maxCur = Math.max(maxCur, nums[i]);

minCur = Math.min(minCur, nums[i]);

result = Math.max(result, maxCur);

maxPre = maxCur;

minPre = minCur;

}

return result;

}

}

Similar to maximum subarray :https://github.com/optimisea/Leetcode/blob/master/Java/53\_MaximumSubarray.java

DP

class Solution {

public int maxProduct(int[] nums) {

int global = Integer.MIN\_VALUE;

int maxCurr = 0;

int minCurr = 0;

int maxPrev = 1;

int minPrev = 1;

for (int num : nums){

maxCurr = Math.max(num, Math.max(maxPrev \* num, minPrev \* num));

minCurr = Math.min(num, Math.min(maxPrev \* num, minPrev \* num));

global = Math.max(global, maxCurr);

maxPrev = maxCurr;

minPrev = minCurr;

}

return global;

}

}

# 153\_Find Minimum in Rotated Sorted Array.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

class Solution {

public int findMin(int[] nums) {

int target = nums[nums.length-1];

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] <= target){

end = mid;

}else{

start = mid;

}

}

if (nums[start] < nums[end]){

return nums[start];

}

return nums[end];

}

}

# 154\_ Find Minimum in Rotated Sorted ArrayII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Follow up for "Find Minimum in Rotated Sorted Array":

What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

The array may contain duplicates.

Method: similar concept as 338 search target in rotated array II

worset case: O(n)

average case: O(logn)

class Solution {

public int findMin(int[] nums) {

int target = nums[nums.length - 1];

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start)/2;

if (nums[mid] == nums[start]){//this is the only difference compared to search minimium I due to duplicates

start++;

}else if (nums[mid] <= target){

end = mid;

}else if (nums[mid] > target){

start = mid;

}

}

if (nums[start] < nums[end]){

return nums[start];

}

return nums[end];

}

}

class Solution {

public int findMin(int[] nums) {

int start = 0;

int end = nums.length - 1;

int target = nums[end];

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == nums[start]){//when eqaul, we don't know which direction to do binary

start++;

}else if (nums[mid] == nums[end]){//when eqaul, we don't know which direction to do binary

end--;

}else if (nums[mid] <= target){

end = mid;

}else{

start = mid;

}

}

return Math.min(nums[start], nums[end]);

}

}

# Intervals

## 156\_MergeIntervals.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a collection of intervals, merge all overlapping intervals.

Have you met this question in a real interview? Yes

Example

Given intervals => merged intervals:

[ [

[1, 3], [1, 6],

[2, 6], => [8, 10],

[8, 10], [15, 18]

[15, 18] ]

]

• 区间左端点从小到大排个序,从左往右扫一遍 :

– 不能合并 ->直接下一个

– 能合并 -> 就合并

/\*\*

\* Definition of Interval:

\* public class Interval {

\* int start, end;

\* Interval(int start, int end) {

\* this.start = start;

\* this.end = end;

\* }

\*/

public class Solution {

/\*

\* @param intervals: interval list.

\* @return: A new interval list.

\*/

public List<Interval> merge(List<Interval> intervals) {

List<Interval> result = new ArrayList<Interval>();

if (intervals == null || intervals.size() == 0){

return result;

}

Collections.sort(intervals, new Comparator<Interval>(){

public int compare(Interval a, Interval b){

return a.start - b.start;

}

});

Interval last = null;

for(Interval item : intervals){

if (last == null || last.end < item.start){

result.add(item);

last = item;

}else{

last.end = Math.max(last.end, item.end); // Modify the element already in list

}

}

return result;

}

}

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public List<Interval> merge(List<Interval> intervals) {

List<Interval> res = new ArrayList<>();

Collections.sort(intervals, new Comparator<Interval>(){

public int compare (Interval i1, Interval i2){

return i1.start - i2.start;

}

});

Interval last = null;

for (Interval interval : intervals){

if (last == null || last.end < interval.start){

res.add(interval);

last = interval;

}else{

last.end = Math.max(last.end, interval.end);

}

}

return res;

}

}

# 157\_Read.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The API: int read4(char \*buf) reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the read4 API, implement the function int read(char \*buf, int n) that reads n characters from the file.

Example 1:

Input: buf = "abc", n = 4

Output: "abc"

Explanation: The actual number of characters read is 3, which is "abc".

Example 2:

Input: buf = "abcde", n = 5

Output: "abcde"

Note:

The read function will only be called once for each test case.

Note that

思路：文件中的字符数可能小于n，所以不断借助read4读取字符的时候需要判断是否已经读到文件末尾。每次将read4中字符复制到buf中时，

也需要判断好能够复制的上限。

注意， n也可能小于文件的字数

The meaning here is that read4() function will read 4 characters at a time from a file and then put the characters that has been read

into this buf variable.

So read() function is reading at most n characters from a file ( we don’t know what file and how it’s reading from the file), and

put x characters into char[] buf.

https://www.jianshu.com/p/b06b90a52617?utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation

buf: the destination buffer

n: max number of char to read

return the number of char that was read

public int read(char[] buf, int n) {

boolean eof = false; // end of file flag

int total = 0; // total bytes have read

char[] tmp = new char[4]; // temp buffer

while (!eof && total < n) {

int count = read4(tmp);

// check if it's the end of the file

eof = count < 4;

// get the actual needed count to put into buf

count = Math.min(count, n - total);

// copy from temp buffer to buf

for (int i = 0; i < count; i++)

buf[total++] = tmp[i];

}

return total;

}

# 158\_Read.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The API: int read4(char \*buf) reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the read4 API, implement the function int read(char \*buf, int n) that reads n characters from the file.

Note:

The read function may be called multiple times.

https://www.youtube.com/watch?v=5gO5syMOKnI

https://www.lintcode.com/problem/read-n-characters-given-read4-ii-call-multiple-times/description

Public class Solution Extends Read4{

@buffer: destination buffer

@n: max character that can read

@return the number of character can be read

private char[] tmp = new char[4];

private tmpCnt = 0; //pointer in tmp

private tmpPtr = 0; //how many char is read in tmp

Format 1:

public int read(char[] buff, int n){

int total = 0;

while (total < n){

if (tmpPtr == 0){//either not start or already read all tmp character

tmpCnt = read4(tmp);

}

if (tmpCnt == 0){//end of file

break;

}

while (total < n && tmpPtr < tmpCnt){

buff[total++] = tmp[tmpPtr++];

}

if (tmpPtr == tmpCnt){

tmpPtr = 0;

}

if (tmpCnt < 4){

break;

}

}

return total;

}

}

Format 2:

public class Solution extends Reader4 {

/\*\*

\* @param buf destination buffer

\* @param n maximum number of characters to read

\* @return the number of characters read

\*/

private char[] tmp = new char[4];

private int tmpCnt = 0;

private int tmpPtr = 0;

public int read(char[] buf, int n) {

int total = 0;

while (total < n){

if (tmpPtr == 0){

tmpCnt = read4(tmp);

}

while (total < n && tmpPtr < tmpCnt){

buf[total++] = tmp[tmpPtr++];

}

if (tmpPtr == tmpCnt){

tmpPtr = 0;

}

if (tmpCnt < 4){

break;

}

}

return total;

}

}

Format 3:

/\* The read4 API is defined in the parent class Reader4.

int read4(char[] buf); \*/

public class Solution extends Reader4 {

/\*\*

\* @param buf destination buffer

\* @param n maximum number of characters to read

\* @return the number of characters read

\*/

private char[] tmp = new char[4];

private int tmpCnt = 0;

private int tmpPtr = 0;

public int read(char[] buf, int n) {

int total = 0;

while (total < n){

if (tmpPtr == 0){

tmpCnt = read4(tmp);

}

if (tmpCnt == 0){

break;

}

while (total < n && tmpPtr < tmpCnt){

buf[total++] = tmp[tmpPtr++];

}

if (tmpPtr == tmpCnt){

tmpPtr = 0;

}

}

return total;

}

}

# 161\_RotateImage.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Have you met this question in a real interview? Yes

Example

Given a matrix

[

[1,2],

[3,4]

]

rotate it by 90 degrees (clockwise), return

[

[3,1],

[4,2]

]

Challenge

Do it in-place.

• 旋转方法总结:

– 顺时针90:先上下,再对角线 – 逆时针90:先左右,再对角线

￼note the different range for different flips

public class Solution {

/\*

\* @param matrix: a lists of integers

\* @return:

\*/

public void rotate(int[][] matrix) {

if (matrix == null || matrix.length == 0){

return;

}

int n = matrix.length;

for (int i = 0; i < n/2; i++){ //1. flip along x-axis

for (int j = 0; j < n; j++){

int temp = matrix[i][j];

matrix[i][j] = matrix[n-1-i][j];

matrix[n-1-i][j] = temp;

}

}

for (int i = 0; i < n; i++){

for (int j = i; j < n; j++){ //2. flip along diagronal

int temp = matrix[i][j];

matrix[i][j] = matrix[j][i];

matrix[j][i] = temp;

}

}

}

}

class Solution {

public void rotate(int[][] matrix) {

int m = matrix.length;

int n = matrix[0].length;

for (int j = 0; j < n; j++){

for (int i = 0; i < m/2; i++){

int temp = matrix[i][j];

matrix[i][j] = matrix[m-1-i][j];

matrix[m-1-i][j] = temp;

}

}

for (int i = 0; i < m; i++){

for (int j = 0; j < i; j++){

int temp = matrix[i][j];

matrix[i][j] = matrix[j][i];

matrix[j][i] = temp;

}

}

}

}

# 162\_FindPeakElement.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A peak element is an element that is greater than its neighbors.

Given an input array where num[i] ≠ num[i+1], find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that num[-1] = num[n] = -∞.

For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

click to show spoilers.

Note:

Your solution should be in logarithmic complexity.

class Solution {

public int findPeakElement(int[] nums) {

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid-1] < nums[mid] && nums[mid] > nums[mid+1]){

return mid;

}else if (nums[mid-1] < nums[mid] && nums[mid] < nums[mid+1]){

start = mid;

}else{

end = mid;

}

}

if (nums[start] < nums[end]){

return end;

}

return start;

}

}

http://www.jiuzhang.com/solutions/find-peak-element/

public int findPeak(int[] A) {

// write your code here

int start = 1, end = A.length-2; // 1.答案在之间，2.不会出界

while(start + 1 < end) {

int mid = (start + end) / 2;

if(A[mid] < A[mid - 1]) {

end = mid;

} else if(A[mid] < A[mid + 1]) {

start = mid;

} else {

end = mid;

}

}

if(A[start] < A[end]) {

return end;

} else {

return start;

}

}

# 163\_Missing.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted integer array nums, where the range of elements are in the inclusive range [lower, upper], return its missing ranges.

Example:

Input: nums = [0, 1, 3, 50, 75], lower = 0 and upper = 99,

Output: ["2", "4->49", "51->74", "76->99"]

Single pointer

class Solution {

public List<String> findMissingRanges(int[] nums, int lower, int upper) {

List<String> res = new ArrayList<>();

int next = lower;

for (int i = 0; i < nums.length; i++){

if (next < nums[i]){

res.add(getRange(next, nums[i] - 1));

next = nums[i] + 1;

}else if (next == nums[i]){

next++;

}

// handle the case that [Integer.MAX\_VALUE], lower = 0, hi = Integer.MAX\_VALUE

if (next == Integer.MIN\_VALUE){

return res;

}

}

if (next <= upper){

res.add(getRange(next, upper));

}

return res;

}

private String getRange(int n1, int n2){

if (n1 == n2){

return String.valueOf(n1);

}

return String.format("%d->%d", n1, n2);

}

}

# 164\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Return 0 if the array contains less than 2 elements.

Example 1:

Input: [3,6,9,1]

Output: 3

Explanation: The sorted form of the array is [1,3,6,9], either

(3,6) or (6,9) has the maximum difference 3.

Example 2:

Input: [10]

Output: 0

Explanation: The array contains less than 2 elements, therefore return 0.

Note:

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

Try to solve it in linear time/space.

class Solution {

public int maximumGap(int[] nums) {

if (nums.length < 2){

return 0;

}

Arrays.sort(nums);

int max = 0;

for (int i = 1; i < nums.length; i++){

max = Math.max(max, nums[i] - nums[i-1]);

}

return max;

}

}

# 165\_Compare.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Compare two version numbers version1 and version2.

If version1 > version2 return 1; if version1 < version2 return -1;otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the . character.

The . character does not represent a decimal point and is used to separate number sequences.

For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Example 1:

Input: version1 = "0.1", version2 = "1.1"

Output: -1

Example 2:

Input: version1 = "1.0.1", version2 = "1"

Output: 1

Example 3:

Input: version1 = "7.5.2.4", version2 = "7.5.3"

Output: -1

Method 1: Two points

class Solution {

public int compareVersion(String version1, String version2) {

int p1 = 0;

int p2 = 0;

while (p1 < version1.length() && p2 < version2.length()){

int v1 = 0;

while (p1 < version1.length() && version1.charAt(p1) != '.'){

v1 = v1 \* 10 + (version1.charAt(p1) - '0');

p1++;

}

int v2 = 0;

while (p2 < version2.length() && version2.charAt(p2) != '.'){

v2 = v2 \* 10 + (version2.charAt(p2) - '0');

p2++;

}

if (v1 < v2){

return -1;

}else if (v1 > v2){

return 1;

}else if (p1 == version1.length() && p2 == version2.length()){

return 0;

}

p1++;

p2++;

}

while (p1 < version1.length() && (version1.charAt(p1) == '0' || version1.charAt(p1) == '.')){

p1++;

}

if (p1 < version1.length()){

return 1;

}

while (p2 < version2.length() && (version2.charAt(p2) == '0' || version2.charAt(p2) == '.')){

p2++;

}

if (p2 < version2.length()){

return -1;

}

return 0;

}

}

Method 2: Use split

Note that

String a = "9";

String[] arr = a.split("\\.");

System.out.println(arr.length); \\output: 1

System.out.println(arr[0]); \\output: 9

String a = "10.";

String[] arr = a.split("\\.");

System.out.println(arr.length); \\output: 1

System.out.println(arr[0]); \\output: 10

String a = ".10";

String[] arr = a.split("\\.");

System.out.println(arr.length); \\output: 2

System.out.println(arr[0]); \\output: ""

System.out.println(arr[1]); \\output: 10

class Solution {

public int compareVersion(String version1, String version2) {

String[] str1 = version1.split("\\.");

String[] str2 = version2.split("\\.");

int i = 0;

int j = 0;

while (i < str1.length || j < str2.length){

int num1 = i < str1.length ? Integer.parseInt(str1[i]) : 0;

int num2 = j < str2.length ? Integer.parseInt(str2[j]) : 0;

if (num1 > num2){

return 1;

}else if (num1 < num2){

return -1;

}

i++;

j++;

}

return 0;

}

}

Note that "." won't work because . in a regular expression means "any character". So we have to use str.split("\\.");

To split white space, use str.split"\\s+")

# 166\_Fraction.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

Given numerator = 1, denominator = 2, return "0.5".

Given numerator = 2, denominator = 1, return "2".

Given numerator = 2, denominator = 3, return "0.(6)".

No scary math, just apply elementary math knowledge. Still remember how to perform a long division?

Try a long division on 49\frac{4}{9}​9​​4​​, the repeating part is obvious. Now try 4333\frac{4}{333}​333​​4​​. Do you see a pattern?

Be wary of edge cases! List out as many test cases as you can think of and test your code thoroughly.

Intuition

The key insight here is to notice that once the remainder starts repeating, so does the divided result.

0.1661.000⎯⎯⎯⎯⎯⎯⎯106⎯⎯⎯⎯⎯⎯4036⎯⎯⎯⎯⎯⎯4⇐remainder=1, mark 1 as seen at position=0.⇐remainder=4, mark 4 as seen at position=1.⇐remainder=4 was seen before at position=1, so the fractional part which is 16 starts repeating at position=1 ⇒ 1(6).

Algorithm

You will need a hash table that maps from the remainder to its position of the fractional part. Once you found a repeating remainder,

you may enclose the reoccurring fractional part with parentheses by consulting the position from the table.

The remainder could be zero while doing the division. That means there is no repeating fractional part and you should stop right away.

Just like the question Divide Two Integers, be wary of edge cases such as negative fractions and nasty extreme case such as

−2147483648−1\frac{-2147483648}{-1}​−1​​−2147483648​​.

Here are some good test cases:

Test case Explanation

01\frac{0}{1}​1​​0​​ Numerator is zero.

10\frac{1}{0}​0​​1​​ Divisor is 0, should handle it by throwing an exception but here we ignore for simplicity sake.

204\frac{20}{4}​4​​20​​ Answer is a whole integer, should not contain the fractional part.

12\frac{1}{2}​2​​1​​ Answer is 0.5, no recurring decimal.

−14\frac{-1}{4}​4​​−1​​ or 1−4\frac{1}{-4}​−4​​1​​ One of the numerator or denominator is negative, fraction is negative.

−1−4\frac{-1}{-4}​−4​​−1​​ Both numerator and denominator are negative, should result in positive fraction.

−2147483648−1\frac{-2147483648}{-1}​−1​​−2147483648​​ Beware of overflow if you cast to positive.

class Solution {

public String fractionToDecimal(int numerator, int denominator) {

if (numerator == 0){

return "0";

}

StringBuilder fraction = new StringBuilder();

// if either one is negative not both;

if ((numerator < 0) ^ (denominator < 0)){

fraction.append("-");

}

// convert to long or else abs(-2147483648) overflows

long dividend = Math.abs(Long.valueOf(numerator));

long divisor = Math.abs(Long.valueOf(denominator));

fraction.append(String.valueOf(dividend/divisor));

long remainder = dividend % divisor;

if (remainder == 0){

return fraction.toString();

}

fraction.append(".");

Map<Long, Integer> map = new HashMap<>();

while (remainder != 0){

if (map.containsKey(remainder)){

fraction.insert(map.get(remainder), "(");

fraction.append(")");

break;

}

map.put(remainder, fraction.length());

remainder \*= 10;

fraction.append(String.valueOf(remainder/divisor));

remainder %= divisor;

}

return fraction.toString();

}

}

# Anagrams

## 171\_Anagrams.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of strings, return all groups of strings that are anagrams.

Notice

All inputs will be in lower-case

Have you met this question in a real interview? Yes

Example

Given ["lint", "intl", "inlt", "code"], return ["lint", "inlt", "intl"].

Given ["ab", "ba", "cd", "dc", "e"], return ["ab", "ba", "cd", "dc"].

Challenge

What is Anagram?

- Two strings are anagram if they can be the same after change the order of characters.

public class Solution {

/\*

\* @param strs: A list of strings

\* @return: A list of strings

\*/

public List<String> anagrams(String[] strs) {

List<String> result = new ArrayList<>();

if (strs == null || strs.length == 0){

return result;

}

Map<String, ArrayList<String>> map = new HashMap<>();

for (int i = 0; i < strs.length; i++){

char[] charArray = strs[i].toCharArray();

Arrays.sort(charArray);

String s = String.valueOf(charArray);

if (!map.containsKey(s)){

map.put(s, new ArrayList<String>());

}

map.get(s).add(strs[i]);

}

for (Map.Entry<String, ArrayList<String>> entry : map.entrySet()){

if (entry.getValue().size() > 1){

result.addAll(entry.getValue());

}

}

return result;

}

}

class Solution {

public List<List<String>> groupAnagrams(String[] strs) {

List<List<String>> res = new ArrayList<>();

Map<String, List<String>> map = new HashMap<>();

for (String str : strs){

char[] cs = str.toCharArray();

Arrays.sort(cs);

String key = new String(cs);

if (!map.containsKey(key)){

map.put(key, new ArrayList<>());

}

map.get(key).add(str);

}

for (List<String> list : map.values()){

res.add(list);

}

return res;

}

}

## 242\_ValidAnagram.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings s and t, write a function to determine if t is an anagram of s.

For example,

s = "anagram", t = "nagaram", return true.

s = "rat", t = "car", return false.

Note:

You may assume the string contains only lowercase alphabets.

Follow up:

What if the inputs contain unicode characters? How would you adapt your solution to such case?

Method 1:

Conver to alphabet array with the size of 26

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public boolean isAnagram(String s, String t) {

if (s == null || t == null){

return false;

}

if (s.length() == 0 && t.length() == 0){

return true;

}

final int MAX\_CHAR = 26;

int[] alphabet = new int[MAX\_CHAR];

for (int i = 0; i < s.length(); i++){

alphabet[s.charAt(i) - 'a']++;

}

for (int i = 0; i < t.length(); i++){

alphabet[t.charAt(i) - 'a']--;

}

for (int i = 0; i < MAX\_CHAR; i++){

if (alphabet[i] != 0){

return false;

}

}

return true;

}

}

Method 2:

Use sort and compare

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public boolean isAnagram(String s, String t) {

if (s == null || t == null){

return false;

}

if (s.length() == 0 && t.length() == 0){

return true;

}

char[] sArray = s.toCharArray();

Arrays.sort(sArray);

String s\_new = String.valueOf(sArray);

char[] tArray = t.toCharArray();

Arrays.sort(tArray);

String t\_new = String.valueOf(tArray);

if (s\_new.equals(t\_new)){

return true;

}

return false;

}

}

Method 3:

class Solution {

public boolean isAnagram(String s, String t) {

int[] alphabet = new int[256];

for (int i = 0; i < s.length(); i++){

alphabet[s.charAt(i)]++;

}

for (int i = 0; i < t.length(); i++){

alphabet[t.charAt(i)]--;

}

for (int i = 0; i < alphabet.length; i++){

if (alphabet[i] != 0){

return false;

}

}

return true;

}

}

Better version:

class Solution {

public boolean isAnagram(String s, String t) {

Map<Character, Integer> map = new HashMap<>();

for (char c : s.toCharArray()){

map.put(c, map.getOrDefault(c, 0) + 1);

}

for (char c : t.toCharArray()){

if (!map.containsKey(c)){

return false;

}else{

map.put(c, map.get(c) - 1);

}

}

for (int val : map.values()){

if (val != 0){

return false;

}

}

return true;

}

}

## 249\_Group.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, we can "shift" each of its letter to its successive letter, for example: "abc" -> "bcd". We can keep "shifting" which

forms the sequence:

"abc" -> "bcd" -> ... -> "xyz"

Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.

For example, given: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"],

A solution is:

[

["abc","bcd","xyz"],

["az","ba"],

["acef"],

["a","z"]

]

Similar as group anagram

class Solution {

public List<List<String>> groupStrings(String[] strings) {

List<List<String>> result = new ArrayList<>();

Map<String, List<String>> map = new HashMap<>();

for (String str : strings){

//first find the key by baselining every char in str

int offset = (int) (str.charAt(0) - 'a');

StringBuilder sb = new StringBuilder();

for (int i = 0; i < str.length(); i++){

char c = (char) (str.charAt(i) - offset);

if (c < 'a'){

c += 26;

}

sb.append(c);

}

String key = sb.toString();

if (!map.containsKey(key)){

map.put(key, new ArrayList<String>());

}

map.get(key).add(str);

}

for (String s : map.keySet()){

result.add(map.get(s));

}

return result;

}

}

## 438\_FindAllAnagramsinString.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s and a non-empty string p, find all the start indices of p's anagrams in s.

Strings consists of lowercase English letters only and the length of both strings s and p will not be larger than 20,100.

The order of output does not matter.

Example 1:

Input:

s: "cbaebabacd" p: "abc"

Output:

[0, 6]

Explanation:

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

Example 2:

Input:

s: "abab" p: "ab"

Output:

[0, 1, 2]

Explanation:

The substring with start index = 0 is "ab", which is an anagram of "ab".

The substring with start index = 1 is "ba", which is an anagram of "ab".

The substring with start index = 2 is "ab", which is an anagram of "ab".

Method 1:

Time complexity: O(s.length() \* p.length())

class Solution {

final static int MAX\_CHAR = 256;

public List<Integer> findAnagrams(String s, String p) {

List<Integer> result = new ArrayList<>();

int[] map = new int[MAX\_CHAR];

int len = p.length();

for (int i = 0; i < len; i++){

map[p.charAt(i)]++;

}

for (int i = 0 ; i <= s.length() - len; i++){

int[] temp = new int[MAX\_CHAR];

for (int j = i; j < i + len; j++){

temp[s.charAt(j)]++;

}

if (Arrays.equals(map,temp)){

result.add(i);

}

}

return result;

}

}

Method 2: Best solution

Time complexity: O(n)

Slide window template

https://leetcode.com/problems/find-all-anagrams-in-a-string/discuss/92007?page=1

class Solution {

public List<Integer> findAnagrams(String s, String p) {

List<Integer> result = new ArrayList<>();

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i < p.length(); i++){

map.put(p.charAt(i), map.getOrDefault(p.charAt(i), 0) + 1);

}

int start = 0;

int end = 0;

int count = map.size();

while (end < s.length()){

char endCh = s.charAt(end);

if (map.containsKey(endCh)){

map.put(endCh, map.get(endCh) - 1);

if (map.get(endCh) == 0){

count--;

}

}

end++;

while (count == 0){

if (end - start == p.length()){

result.add(start);

}

char startCh = s.charAt(start);

if (map.containsKey(startCh)){

if (map.get(startCh) == 0){

count++;

}

map.put(startCh, map.get(startCh) + 1);

}

start++;

}

}

return result;

}

}

Method 3:

Time complexity: O(n)

class Solution {

final static int MAX\_CHAR = 256;

public List<Integer> findAnagrams(String s, String p) {

List<Integer> result = new ArrayList<>();

if (s.length() < p.length()){

return result;

}

int[] map = new int[MAX\_CHAR];

int len = p.length();

for (int i = 0; i < len; i++){

map[p.charAt(i)]++;

}

int left = 0, right = 0;

int count = len;

while(right < s.length()){

if(map[s.charAt(right)] > 0){

count--;

}

map[s.charAt(right)]--;

right++;

if(count == 0){

result.add(left);

}

if(right - left == len){

if (map[s.charAt(left)] >= 0){

count++;

}

map[s.charAt(left)]++;

left++;

}

}

return result;

}

}

# Excel

## 168\_ColumnExcel.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

For example:

1 -> A

2 -> B

3 -> C

...

26 -> Z

27 -> AA

28 -> AB

Method 1:

class Solution {

public String convertToTitle(int n) {

StringBuilder sb = new StringBuilder();

int weight = 26;

while (n > 0){

sb.append((char)((n - 1) % weight + 'A'));

n = (n - 1) / weight;

}

return sb.reverse().toString();

}

}

Method 2:

class Solution {

public String convertToTitle(int n) {

StringBuilder sb = new StringBuilder();

int weight = 26;

while (n > 0){

n--;

sb.insert(0, (char)(n % weight + 'A'));

n = n / weight;

}

return sb.toString();

}

}

## 171\_ExcelNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Related to question Excel Sheet Column Title

Given a column title as appear in an Excel sheet, return its corresponding column number.

For example:

A -> 1

B -> 2

C -> 3

...

Z -> 26

AA -> 27

AB -> 28

Method 1:

class Solution {

public int titleToNumber(String s) {

int times = 1;

int weight = 26;

int ans = 0;

for (int i = s.length() - 1; i >= 0; i--){

int num = s.charAt(i) - 'A' + 1;

ans += num \* times;

times \*= weight;

}

return ans;

}

}

Method 2:

class Solution {

public int titleToNumber(String s) {

int ans = 0;

int weight = 26;

for (int i = 0; i < s.length(); i++){

ans = ans \* weight + s.charAt(i) - 'A' + 1;

}

return ans;

}

}

# 169\_MajorityElement.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of size n, find the majority element. The majority element is the element that appears more than ⌊ n/2 ⌋ times.

You may assume that the array is non-empty and the majority element always exist in the array.

Method 1:

class Solution {

public int majorityElement(int[] nums) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++){

map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);

}

for (Integer num : map.keySet()){

if (map.get(num) > nums.length / 2){

return num;

}

}

return Integer.MIN\_VALUE;

}

}

class Solution {

public int majorityElement(int[] nums) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++){

map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);

if (map.get(nums[i]) > nums.length / 2){

return nums[i];

}

}

return Integer.MIN\_VALUE;

}

}

Method 2:

class Solution {

public int majorityElement(int[] nums) {

Arrays.sort(nums);

return nums[nums.length/2];

}

}

Method 3: Best solution:

class Solution {

public int majorityElement(int[] nums) {

int count = 0;

int res = 0;

for (int num : nums){

if (num == res){

count++;

}else if (count == 0){

res = num;

count++;

}else{

count--;

}

}

return res;

}

}

class Solution {

public int majorityElement(int[] nums) {

int res = nums[0];

int count = 1;

for (int i = 1; i < nums.length; i++){

if (count == 0){

res = nums[i];

count = 1;

}else if (nums[i] == res){

count++;

}else{

count--;

}

}

return res;

}

}

Method 4: Similar as Single Number II

bit manipulation:

class Solution {

public int majorityElement(int[] nums) {

int res = 0;

for (int i = 0; i < 32; i++){

int ones = 0;

int zeros = 0;

for (int j = 0; j < nums.length; j++){

if ((nums[j] & (1 << i)) != 0){

ones++;

}else{

zeros++;

}

}

if (ones > zeros){

res |= (1 << i);

}

}

return res;

}

}

# 172\_FactorialTrailingZeroes.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer n, return the number of trailing zeroes in n!.

Note: Your solution should be in logarithmic time complexity.

class Solution {

public int trailingZeroes(int n) {

int ans = 0;

while (n > 0){

ans += n / 5;

n /= 5;

}

return ans;

}

}

class Solution {

public int trailingZeroes(int n) {

return n == 0 ? 0 : n/5 + trailingZeroes(n/5);

}

}

# 174\_DungeonGame.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The demons had captured the princess (P) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms

laid out in a 2D grid. Our valiant knight (K) was initially positioned in the top-left room and must fight his way through the dungeon

to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below,

he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms

are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.

For example, given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path

RIGHT-> RIGHT -> DOWN -> DOWN.

Method 1: f[i][j] denotes the minimal hp needed to exit from dungeon[i][j]

class Solution {

public int calculateMinimumHP(int[][] dungeon) {

int m = dungeon.length;

int n = dungeon[0].length;

int[][] f = new int[m][n];

f[m-1][n-1] = 1;

for (int i = m-2; i>=0; i--){

f[i][n-1] = f[i+1][n-1] - dungeon[i+1][n-1];

if (f[i][n-1] <= 0){

f[i][n-1] = 1;

}

}

for (int i = n-2; i>=0; i--){

f[m-1][i] = f[m-1][i+1] - dungeon[m-1][i+1];

if (f[m-1][i] <= 0){

f[m-1][i] = 1;

}

}

for (int i = m-2; i>=0; i--){

for (int j = n-2; j>=0; j--){

f[i][j] = Math.min(f[i+1][j] - dungeon[i+1][j], f[i][j+1] - dungeon[i][j+1]);

if (f[i][j] <= 0){

f[i][j] = 1;

}

}

}

return Math.max(f[0][0] - dungeon[0][0], 1);

}

}

Method 2: Better solution: health[i][j] denotes the minimal hp needed to enter into dungeon[i][j]

public int calculateMinimumHP(int[][] dungeon) {

if (dungeon == null || dungeon.length == 0 || dungeon[0].length == 0) return 0;

int m = dungeon.length;

int n = dungeon[0].length;

int[][] health = new int[m][n];

health[m - 1][n - 1] = Math.max(1 - dungeon[m - 1][n - 1], 1);

for (int i = m - 2; i >= 0; i--) {

health[i][n - 1] = Math.max(health[i + 1][n - 1] - dungeon[i][n - 1], 1);

}

for (int j = n - 2; j >= 0; j--) {

health[m - 1][j] = Math.max(health[m - 1][j + 1] - dungeon[m - 1][j], 1);

}

for (int i = m - 2; i >= 0; i--) {

for (int j = n - 2; j >= 0; j--) {

int down = Math.max(health[i + 1][j] - dungeon[i][j], 1);

int right = Math.max(health[i][j + 1] - dungeon[i][j], 1);

health[i][j] = Math.min(right, down);

}

}

return health[0][0];

}

class Solution {

public int calculateMinimumHP(int[][] dungeon) {

if (dungeon == null || dungeon.length == 0){

return 0;

}

int m = dungeon.length;

int n = dungeon[0].length;

int[][] dp = new int[m][n];

dp[m-1][n-1] = Math.max(1 - dungeon[m-1][n-1], 1);

for (int i = m - 2; i >= 0; i--){

dp[i][n-1] = Math.max(dp[i+1][n-1] - dungeon[i][n-1], 1);

}

for (int j = n - 2; j >= 0; j--){

dp[m-1][j] = Math.max(dp[m-1][j+1] - dungeon[m-1][j], 1);

}

for (int i = m - 2; i >= 0; i--){

for (int j = n- 2; j >= 0; j--){

int down = Math.max(dp[i+1][j] - dungeon[i][j], 1);

int right = Math.max(dp[i][j+1] - dungeon[i][j], 1);

dp[i][j] = Math.min(down, right);

}

}

return dp[0][0];

}

}

# 179\_LargestNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of non negative integers, arrange them such that they form the largest number.

For example, given [3, 30, 34, 5, 9], the largest formed number is 9534330.

Note: The result may be very large, so you need to return a string instead of an integer.

class Solution {

public String largestNumber(int[] nums) {

if (nums == null || nums.length == 0){

return "";

}

String[] strs = new String[nums.length];

for (int i = 0; i < nums.length; i++){

strs[i] = Integer.toString(nums[i]);

}

Arrays.sort(strs, new Comparator<String>(){

public int compare(String a, String b){

String ab = a + b;

String ba = b + a;

return ba.compareTo(ab);

}

});

if (strs[0].equals("0")){

return "0";

}

StringBuilder ans = new StringBuilder();

for (int i = 0; i < strs.length; i++){

ans.append(strs[i]);

}

return ans.toString();

}

}

# 187\_Repeated.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for

example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example,

Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

Return:

["AAAAACCCCC", "CCCCCAAAAA"].

class Solution {

public List<String> findRepeatedDnaSequences(String s) {

Set<String> set = new HashSet<>();

Set<String> result = new HashSet<>();

for (int i = 0; i + 9 < s.length(); i++){

String str = s.substring(i, i+10);

if (!set.add(str)){

result.add(str);

}

}

return new ArrayList<>(result);

}

}

# 189\_RotateArray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Rotate an array of n elements to the right by k steps.

For example, with n = 7 and k = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4].

Note:

Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

[show hint]

Hint:

Could you do it in-place with O(1) extra space?

Related problem: Reverse Words in a String II

Method 1:

Space complexity: O(n)

class Solution {

public void rotate(int[] nums, int k) {

if (nums == null || nums.length == 0){

return;

}

int n = nums.length;

k = k % n;

int[] temp = new int[2\*n];

for (int i = 0; i < n; i++){

temp[i] = nums[i];

temp[i + n] = nums[i];

}

int i = 0;

int index = n - k;

while (i < n){

nums[i] = temp[index];

i++;

index++;

}

}

}

Method 2: best solution

class Solution {

public void rotate(int[] nums, int k) {

if (nums == null || nums.length == 0){

return;

}

int n = nums.length;

k = k % n;

reverse(nums, 0, n - k - 1);

reverse(nums, n - k, n - 1);

reverse(nums, 0, n - 1);

}

private void reverse(int[] nums, int start, int end){

while (start < end){

int temp = nums[start];

nums[start] = nums[end];

nums[end] = temp;

start++;

end--;

}

}

}

# 190\_ReverseUnsignedInteger.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Reverse bits of a given 32 bits unsigned integer.

Example:

Input: 43261596

Output: 964176192

Explanation: 43261596 represented in binary as 00000010100101000001111010011100,

return 964176192 represented in binary as 00111001011110000010100101000000.

Follow up:

If this function is called many times, how would you optimize it?

public class Solution {

// you need treat n as an unsigned value

public int reverseBits(int n) {

int result = 0;

for (int i = 0; i < 32; i++){

result = (result << 1) + (n & 1);

n = (n >> 1);

}

return result;

}

}

>>> zeros filled right shift

>> right shift

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise\_Operators

# 191\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

Example :

Input: 11

Output: 3

Explanation: the 32-bit integer 11 has binary representation 00000000000000000000000000001011 .

public class Solution {

// you need to treat n as an unsigned value

public int hammingWeight(int n) {

int count = 0;

while (n != 0){

n = n & (n-1);

count++;

}

return count;

}

}

public class Solution {

// you need to treat n as an unsigned value

public int hammingWeight(int n) {

int count = 0;

for (int i = 0; i < 32; i++){

if (((n >> i) & 1) == 1){

count++;

}

}

return count;

}

}

# 198\_HouseRobber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you

from robbing each of them is that adjacent houses have security system connected and it will automatically contact

the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount

of money you can rob tonight without alerting the police.

class Solution {

public int rob(int[] nums) {

int a = 0;

int b = 0;

for (int i = 0; i < nums.length; i++){

if (i%2 == 0){

a = Math.max(nums[i] + a, b);

}else{

b = Math.max(a, nums[i] + b);

}

}

return Math.max(a, b);

}

}

Method 2: dynamic programming (序列型,接龙型）

class Solution {

public int rob(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int n = nums.length;

int[] f = new int[n+1];

f[0] = 0;

f[1] = nums[0];

for (int i = 2; i <= n; i++){

f[i] = Math.max(f[i-1], f[i-2] + nums[i-1]);

}

return f[n];

}

}

Method 3: dynamic programming (滚动数组，滚动指针，滚动矩阵)

class Solution {

public int rob(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int n = nums.length;

int[] f = new int[2];

f[0] = 0;

f[1] = nums[0];

for (int i = 2; i <= n; i++){

f[i%2] = Math.max(f[(i-1)%2], f[(i-2)%2] + nums[i-1]);

}

return f[n%2];

}

}

## 213\_HouseRobberII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Note: This is an extension of House Robber.

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention.

This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the

last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum

amount of money you can rob tonight without alerting the police.

Since every house is either robbed or not robbed and at least half of the houses are not robbed,

the solution is simply the larger of two cases with consecutive houses, i.e. house i not robbed,

break the circle, solve it, or house i + 1 not robbed. Hence, the following solution.

I chose i = n and i + 1 = 0 for simpler coding. But, you can choose whichever two consecutive ones.

class Solution {

public int rob(int[] nums) {

if (nums.length == 1){

return nums[0];

}

return Math.max(rob(nums, 0, nums.length - 2), rob(nums, 1, nums.length - 1));

}

private int rob(int[] nums, int start, int end){

int odd = 0;

int even = 0;

for (int i = start; i <= end; i++){

if (i%2 == 0){

even = Math.max(even + nums[i], odd);

}else{

odd = Math.max(odd + nums[i], even);

}

}

return Math.max(odd, even);

}

}

Method 2: 循环数组处理方法一： 分裂

class Solution {

public int rob(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int n = nums.length;

if (n == 1){

return nums[0];

}

int[] A = new int[n - 1];

for (int i = 0; i < n - 1; i++){

A[i] = nums[i];

}

int[] B = new int[n - 1];

for (int i = 1; i < n; i++){

B[i-1] = nums[i];

}

return Math.max(robI(A), robI(B));

}

private int robI(int[] nums){

int n = nums.length;

int[] f = new int[n+1];

f[0] = 0;

f[1] = nums[0];

for (int i = 2; i <= n; i++){

f[i] = Math.max(f[i-1], f[i-2] + nums[i-1]);

}

return f[n];

}

}

## 337\_HouseRobberIII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root."

Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in

this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on

the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

Example 1:

3

/ \

2 3

\ \

3 1

Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.

Example 2:

3

/ \

4 5

/ \ \

1 3 1

Maximum amount of money the thief can rob = 4 + 5 = 9.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int rob(TreeNode root) {

if (root == null){

return 0;

}

int robRoot = root.val + (root.left == null ? 0 : rob(root.left.left) + rob(root.left.right)) + (root.right == null ? 0 : rob(root.right.left) + rob(root.right.right));

int notRobRoot = rob(root.left) + rob(root.right);

return Math.max(robRoot, notRobRoot);

}

}

Better:

class Solution {

public int rob(TreeNode root) {

if (root == null){

return 0;

}

int maxLeft = rob(root.left);

int maxRight = rob(root.right);

int max = root.val;

if (root.left != null){

max += rob(root.left.left) + rob(root.left.right);

}

if (root.right != null){

max += rob(root.right.left) + rob(root.right.right);

}

return Math.max(max, maxLeft + maxRight);

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int rob(TreeNode root) {

if (root == null){

return 0;

}

int max = robIncludeRoot(root);

int left = rob(root.left);

int right = rob(root.right);

return Math.max(max, left + right);

}

private int robIncludeRoot(TreeNode root){

if (root == null){

return 0;

}

int max = root.val;

if (root.left != null){

max += rob(root.left.left) + rob(root.left.right);

}

if (root.right != null){

max += rob(root.right.left) + rob(root.right.right);

}

return max;

}

}

# 200\_NumberOfIslands.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by

connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

Input:

11110

11010

11000

00000

Output: 1

Example 2:

Input:

11000

11000

00100

00011

Output: 3

Method 1: BFS

class Solution {

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

int count = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == '1'){

bfs(grid, i, j);

count++;

}

}

}

return count;

}

private void bfs(char[][] grid, int x, int y){

int m = grid.length;

int n = grid[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

queue.offer(new int[]{x, y});

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx < m && nx >= 0 && ny < n && ny >= 0 && grid[nx][ny] == '1'){

queue.offer(new int[]{nx, ny});

grid[nx][ny] = '0';

}

}

}

}

}

Better BFS version:

class Solution {

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

boolean[][] visited = new boolean[m][n];

int count = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (!visited[i][j] && grid[i][j] == '1'){

bfs(grid, visited, i, j);

count++;

}

}

}

return count;

}

private void bfs(char[][] grid, boolean[][] visited, int x, int y){

int m = grid.length;

int n = grid[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

queue.offer(new int[]{x, y});

visited[x][y] = true;

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny] && grid[nx][ny] == '1'){

visited[nx][ny] = true;

queue.offer(new int[]{nx, ny});

}

}

}

}

}

Method 2: Union Find faster than BFS

Time complexity: O(m\*n)

class Solution {

class UF {

int[] parent;

public UF (int N){

parent = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

}

}

public int find(int x){

if (x == parent[x]){

return x;

}

return parent[x] = find(parent[x]);

}

public void union (int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

}

}

}

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

UF uf = new UF(m\*n);

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

int count = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == '1'){

count++;

int num1 = i \* n + j;

for (int[] dir : dirs){

int x = i + dir[0];

int y = j + dir[1];

if (x >= 0 && x < m && y >= 0 && y < n && grid[x][y] == '1'){

int num2 = x \* n + y;

if (uf.find(num1) != uf.find(num2)){

uf.union(num1, num2);

count--;

}

}

}

}

}

}

return count;

}

}

# 201\_Bitwise.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

Example 1:

Input: [5,7]

Output: 4

Example 2:

Input: [0,1]

Output: 0

The key is to find common parts from left to right between m and n.

Note that the last bit of any odd number and any even number will be 0

O(1)

class Solution {

public int rangeBitwiseAnd(int m, int n) {

int i = 0;

while (m != n){

m >>= 1;

n >>= 1;

i++;

}

return m << i;

}

}

https://leetcode.com/problems/bitwise-and-of-numbers-range/discuss/56729/Bit-operation-solution(JAVA)

TLE

O(n)

class Solution {

public int rangeBitwiseAnd(int m, int n) {

int res = 0;

for (int i = 0; i < 32; i++){

int digit = 1;

for (int j= m; j <= n; j++){

digit &= (j >> i);

}

res |= (digit << i);

}

return res;

}

}

# 202\_HappyNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write an algorithm to determine if a number is "happy".

A happy number is a number defined by the following process: Starting with any positive integer,

replace the number by the sum of the squares of its digits, and repeat the process until the number

equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. T

hose numbers for which this process ends in 1 are happy numbers.

Example: 19 is a happy number

12 + 92 = 82

82 + 22 = 68

62 + 82 = 100

12 + 02 + 02 = 1

class Solution {

public boolean isHappy(int n) {

Set<Integer> set = new HashSet<>();

while (n != 1){

n = getDigit(n);

if (n == 1){

return true;

}

if (set.contains(n)){

return false;

}else{

set.add(n);

}

}

return true;

}

private int getDigit(int n){

int ans = 0;

while (n != 0){

int temp = n % 10;

ans += temp \* temp;

n /= 10;

}

return ans;

/\* List<Integer> digits = new ArrayList<>();

while (n != 0){

digits.add(n%10);

n /= 10;

}

int ans = 0;

for (Integer i : digits){

ans += i \* i;

}

return ans;\*/

}

}

class Solution {

public boolean isHappy(int n) {

Set<Integer> set = new HashSet<>();

while (true){

int sum = getSum(n);

if (sum == 1){

return true;

}

if (set.contains(sum)){

return false;

}

set.add(sum);

n = sum;

}

}

private int getSum(int n){

int sum = 0;

while (n > 0){

int digit = n % 10;

sum += digit \* digit;

n /= 10;

}

return sum;

}

}

# 204\_CountPrimes.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Description:

Count the number of prime numbers less than a non-negative number, n.

class Solution {

public int countPrimes(int n) {

int ans = 0;

boolean[] notPrime = new boolean[n];

for (int i = 2; i < n; i++){

if (!notPrime[i]){

ans++;

for (int j = 2; i \* j < n; j++){

notPrime[i\*j] = true;

}

}

}

return ans;

}

}

# 205\_IsomorphicStrings.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings s and t, determine if they are isomorphic.

Two strings are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters.

No two characters may map to the same character but a character may map to itself.

For example,

Given "egg", "add", return true.

Given "foo", "bar", return false.

Given "paper", "title", return true.

Note:

You may assume both s and t have the same length.

class Solution {

public boolean isIsomorphic(String s, String t) {

if (s == null || t == null){

return false;

}

if (s.length() == 0 && t.length() == 0){

return true;

}

if (s.length() == 0 || t.length() == 0){

return false;

}

Map<Character, Character> map1 = new HashMap<>();

Map<Character, Character> map2 = new HashMap<>();

for (int i = 0; i < s.length(); i++){

if (!map1.containsKey(s.charAt(i))){

map1.put(s.charAt(i), t.charAt(i));

}else{

if (t.charAt(i) != map1.get(s.charAt(i))){

return false;

}

}

}

for (int i = 0; i < t.length(); i++){

if (!map2.containsKey(t.charAt(i))){

map2.put(t.charAt(i), s.charAt(i));

}else{

if (s.charAt(i) != map2.get(t.charAt(i))){

return false;

}

}

}

return true;

}

}

# 207\_CourseSchedule.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is

expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

For example:

2, [[1,0]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

2, [[1,0],[0,1]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course

0 you should also have finished course 1. So it is impossible.

Note:

The input prerequisites is a graph represented by a list of edges, not adjacency matrices. Read more about how a

graph is represented.

You may assume that there are no duplicate edges in the input prerequisites.

click to show more hints.

Hints:

This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological

ordering exists and therefore it will be impossible to take all courses.

Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.

Topological sort could also be done via BFS.

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

int N = prerequisites.length;

if (numCourses <= 0){

return false;

}

//calculate indegree

int[] inDegree = new int[numCourses];

Queue<Integer> queue = new LinkedList<>();

for (int i = 0; i < N; i++){

inDegree[prerequisites[i][0]]++;

}

for (int i = 0; i < inDegree.length; i++){

if (inDegree[i] == 0){

queue.offer(i);

}

}

while (!queue.isEmpty()){

int cur = queue.poll();

for (int i = 0; i < N; i++){

if (prerequisites[i][1] == cur){

inDegree[prerequisites[i][0]]--;

if (inDegree[prerequisites[i][0]] == 0){

queue.offer(prerequisites[i][0]);

}

}

}

}

for (int i = 0; i < inDegree.length; i++){

if (inDegree[i] != 0){

return false;

}

}

return true;

}

}

Better:

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

int[] indegree = new int[numCourses];

for (int[] p : prerequisites){

indegree[p[0]]++;

}

Queue<Integer> queue = new LinkedList<>();

for (int i = 0; i < numCourses; i++){

if (indegree[i] == 0){

queue.offer(i);

}

}

while (!queue.isEmpty()){

int course = queue.poll();

for (int[] p : prerequisites){

if (p[1] == course){

indegree[p[0]]--;

if (indegree[p[0]] == 0){

queue.offer(p[0]);

}

}

}

}

for (int i = 0; i < numCourses; i++){

if (indegree[i] != 0){

return false;

}

}

return true;

}

}

Best from topological sort template: need two maps

indegree map: Map<Integer, Integer>

priority map: Map<Integer, Set<Integer>>

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

int[] indegree = new int[numCourses];

Map<Integer, Set<Integer>> map = new HashMap<>();

for (int[] p : prerequisites){

indegree[p[0]]++;

if (!map.containsKey(p[1])){

map.put(p[1], new HashSet<>());

}

map.get(p[1]).add(p[0]);

}

Queue<Integer> queue = new LinkedList<>();

for (int i = 0; i < numCourses; i++){

if (indegree[i] == 0){

queue.offer(i);

}

}

List<Integer> res = new ArrayList<>();

while (!queue.isEmpty()){

int course = queue.poll();

res.add(course);

if (!map.containsKey(course)){

continue;

}

Set<Integer> set = map.get(course);

for(int next : set){

indegree[next]--;

if (indegree[next] == 0){

queue.offer(next);

}

}

}

return res.size() == numCourses;

}

}

## 210\_CourseScheduleII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are a total of n courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1,

which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you

should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish

all courses, return an empty array.

For example:

2, [[1,0]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct

course order is [0,1]

4, [[1,0],[2,0],[3,1],[3,2]]

There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2.

Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is [0,1,2,3].

Another correct ordering is[0,2,1,3].

Note:

The input prerequisites is a graph represented by a list of edges, not adjacency matrices. Read more about

how a graph is represented.

You may assume that there are no duplicate edges in the input prerequisites.

click to show more hints.

class Solution {

public int[] findOrder(int numCourses, int[][] prerequisites) {

if (numCourses <= 0){

return new int[0];

}

int[] result = new int[numCourses];

int index = 0;

int[] inDegree = new int[numCourses];

Queue<Integer> queue = new LinkedList<>();

for (int i = 0; i < prerequisites.length; i++){

inDegree[prerequisites[i][0]]++;

}

for (int i = 0 ; i < inDegree.length; i++){

if (inDegree[i] == 0){

queue.offer(i);

result[index] = i;

index++;

}

}

while (!queue.isEmpty()){

int cur = queue.poll();

for (int i = 0; i < prerequisites.length; i++){

if (prerequisites[i][1] == cur){

inDegree[prerequisites[i][0]]--;

if (inDegree[prerequisites[i][0]] == 0){

queue.offer(prerequisites[i][0]);

result[index] = prerequisites[i][0];

index++;

}

}

}

}

if (index == numCourses){

return result;

}

return new int[0];

}

}

Better version: the same as course schedule I

class Solution {

public int[] findOrder(int numCourses, int[][] prerequisites) {

int[] res = new int[numCourses];

int index = 0;

int[] indegree = new int[numCourses];

for (int[] p : prerequisites){

indegree[p[0]]++;

}

Queue<Integer> queue = new LinkedList<>();

for (int i = 0; i < numCourses; i++){

if (indegree[i] == 0){

queue.offer(i);

}

}

while (!queue.isEmpty()){

int curr = queue.poll();

res[index++] = curr;

for (int[] p : prerequisites){

if (p[1] == curr){

indegree[p[0]]--;

if (indegree[p[0]] == 0){

queue.offer(p[0]);

}

}

}

}

for (int i = 0; i < numCourses; i++){

if (indegree[i] != 0){

return new int[0];

}

}

return res;

}

}

Best solution:

class Solution {

public int[] findOrder(int numCourses, int[][] prerequisites) {

int[] indegree = new int[numCourses];

Map<Integer, Set<Integer>> map = new HashMap<>();

for (int[] pre : prerequisites){

indegree[pre[0]]++;

if (!map.containsKey(pre[1])){

map.put(pre[1], new HashSet<>());

}

map.get(pre[1]).add(pre[0]);

}

Queue<Integer> queue = new LinkedList<>();

for (int i = 0; i < indegree.length; i++){

if (indegree[i] == 0){

queue.offer(i);

}

}

List<Integer> list = new ArrayList<>();

while (!queue.isEmpty()){

int curr = queue.poll();

list.add(curr);

if (map.containsKey(curr)){

Set<Integer> set = map.get(curr);

for (int next : set){

indegree[next]--;

if (indegree[next] == 0){

queue.offer(next);

}

}

}

}

if (list.size() < numCourses){

return new int[0];

}

int index = 0;

int[] res = new int[numCourses];

for (int i = 0; i < numCourses; i++){

res[index++] = list.get(i);

}

return res;

}

}

# 214\_Shortest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it.

Find and return the shortest palindrome you can find by performing this transformation.

For example:

Given "aacecaaa", return "aaacecaaa".

Given "abcd", return "dcbabcd"

https://leetcode.com/articles/shortest-palindrome/

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public String shortestPalindrome(String s) {

StringBuilder sb = new StringBuilder();

int len = s.length();

for (int i = len - 1; i >= 0; i--){

sb.append(s.charAt(i));

}

String rev = sb.toString();

StringBuilder ans = new StringBuilder();

for (int i = 0; i < len; i++){

if (s.substring(0, len - i).equals(rev.substring(i))){

ans.append(rev.substring(0, i));

ans.append(s);

break;

}

}

return ans.toString();

}

}

# 215\_Kth.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct

element.

Example 1:

Input: [3,2,1,5,6,4] and k = 2

Output: 5

Example 2:

Input: [3,2,3,1,2,4,5,5,6] and k = 4

Output: 4

Note:

You may assume k is always valid, 1 ≤ k ≤ array's length.

Quick Select

class Solution {

public int findKthLargest(int[] nums, int k) {

int start = 0;

int end = nums.length - 1;

while (start <= end){

int index = partition(nums, start, end);

if (index == k-1){

return nums[index];

}else if (index > k-1){

end = index-1;

}else{

start = index+1;

}

}

return nums[start];

}

private int partition(int[] nums, int start, int end){

Random random = new Random();

int index = start + random.nextInt(end - start + 1);

int pivot = nums[index];

swap(nums, index, start);

int head = start;

start++;

while (start <= end){

while (start <= end && nums[start] >= pivot){

start++;

}

while (start <= end && nums[end] < pivot){

end--;

}

if (start < end){

swap(nums, start, end);

start++;

end--;

}

}

swap(nums, head, end);

return end;

}

private void swap(int[] nums, int i, int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

# 216\_Combination.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find all possible combinations of k numbers that add up to a number n, given that only numbers

from 1 to 9 can be used and each combination should be a unique set of numbers.

Example 1:

Input: k = 3, n = 7

Output:

[[1,2,4]]

Example 2:

Input: k = 3, n = 9

Output:

[[1,2,6], [1,3,5], [2,3,4]]

class Solution {

public List<List<Integer>> combinationSum3(int k, int n) {

List<List<Integer>> result = new ArrayList<>();

int[] nums = new int[9];

for (int i = 0; i < 9; i++){

nums[i] = i+1;

}

dfs(result, new ArrayList<Integer>(), nums, k, n, 0);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int[] nums, int k, int target, int start){

if (target == 0 && k == 0){

result.add(new ArrayList<>(item));

return;

}

for (int i = start; i < nums.length; i++){

if (nums[i] > target){

break;

}

item.add(nums[i]);

dfs(result, item, nums, k - 1, target - nums[i], i+1);

item.remove(item.size() - 1);

}

}

}

# 218\_Skyline.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

http://www.lintcode.com/en/problem/building-outline/

https://leetcode.com/problems/the-skyline-problem/description/

Sweep line + TreeSet

Leetcode:

class Solution {

class Point{

int height;

int position;

int flag;

int start;

public Point(int height, int position, int flag, int start){

this.height = height;

this.position = position;

this.flag = flag;

this.start = start;

}

}

public List<int[]> getSkyline(int[][] buildings) {

List<int[]> result = new ArrayList<>();

List<Point> list = new ArrayList<>();

for(int[] b : buildings){

list.add(new Point(b[2], b[0], 1, b[0]));

list.add(new Point(b[2], b[1], 0, b[0]));

}

Collections.sort(list, new Comparator<Point>(){

public int compare(Point a, Point b){

if (a.position == b.position){

return b.flag - a.flag;

}

return a.position - b.position;

}

});

TreeSet<Point> pointTree = new TreeSet<>(new Comparator<Point>(){

public int compare(Point a, Point b){

if (a.height == b.height){

return a.position - b.position;

}

return a.height - b.height;

}

});

int prevHeight = 0;

int prevPos = -1;

int currHeight = 0;

for (Point p : list){

if (p.flag == 1){

pointTree.add(p);

currHeight = pointTree.last().height;

}else{

pointTree.remove(new Point(p.height, p.start, 1, p.start));

currHeight = pointTree.isEmpty() ? 0 : pointTree.last().height;

}

if (currHeight != prevHeight){

if (prevHeight != 0 && prevPos != -1 && prevPos != p.position){

int[] item = new int[2];

item[0] = prevPos;

item[1] = prevHeight;

result.add(item);

}

prevHeight = currHeight;

prevPos = p.position;

if (currHeight == 0){

int[] ground = new int[2];

ground[0] = p.position;

ground[1] = currHeight;

result.add(ground);

}

}

}

return result;

}

}

Lintcode:

public class Solution {

/\*\*

\* @param buildings: A list of lists of integers

\* @return: Find the outline of those buildings

\*/

class Point{

int height;

int position;

int flag;

int start;

public Point(int height, int position, int flag, int start){

this.height = height;

this.position = position;

this.flag = flag;

this.start = start;

}

}

public List<List<Integer>> buildingOutline(int[][] buildings) {

List<List<Integer>> result = new ArrayList<>();

List<Point> list = new ArrayList<>();

for(int[] b : buildings){

list.add(new Point(b[2], b[0], 1, b[0]));

list.add(new Point(b[2], b[1], 0, b[0]));

}

Collections.sort(list, new Comparator<Point>(){

public int compare(Point a, Point b){

if (a.position == b.position){

return b.flag - a.flag;

}

return a.position - b.position;

}

});

TreeSet<Point> pointTree = new TreeSet<>(new Comparator<Point>(){

public int compare(Point a, Point b){

if (a.height == b.height){

return a.position - b.position;

}

return a.height - b.height;

}

});

int prevHeight = 0;

int prevPos = 0;

int currHeight = 0;

for (Point p : list){

if (p.flag == 1){

pointTree.add(p);

currHeight = pointTree.last().height;

}else{

pointTree.remove(new Point(p.height, p.start, 1, p.start));

currHeight = pointTree.isEmpty() ? 0 : pointTree.last().height;

}

if (currHeight != prevHeight){

if (prevHeight != 0 && prevPos != 0 && prevPos != p.position){

List<Integer> item = new ArrayList<>();

item.add(prevPos);

item.add(p.position);

item.add(prevHeight);

result.add(item);

}

prevHeight = currHeight;

prevPos = p.position;

}

}

return result;

}

}

https://leetcode.com/problems/the-skyline-problem/discuss/61193/Short-Java-solution

Best solution:

class Solution {

public List<int[]> getSkyline(int[][] buildings) {

List<int[]> res = new ArrayList<>();

List<int[]> height = new ArrayList<>();

for(int[] b : buildings){

height.add(new int[]{b[0], b[2]});

height.add(new int[]{b[1], -b[2]});

}

Collections.sort(height, new Comparator<int[]>(){

public int compare (int[] a, int[] b){

if (a[0] == b[0]){

return b[1] - a[1];//start must go first

}

return a[0] - b[0];

}

});

Queue<Integer> pq = new PriorityQueue<>(new Comparator<Integer>(){

public int compare(Integer i1, Integer i2){

return (int)(i2 - i1);

}

});

pq.offer(0);

int prev = 0;

for (int[] b : height){

if (b[1] > 0){//start

pq.offer(b[1]);

}else{//end

pq.remove(-b[1]);

}

int curr = pq.peek();

if (prev != curr){

res.add(new int[]{b[0], curr});

prev = curr;

}

}

return res;

}

}

# 219\_Contains.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers and an integer k, find out whether there are two distinct indices i and j

in the array such that nums[i] = nums[j] and the absolute difference between i and j is at most k.

class Solution {

public boolean containsNearbyDuplicate(int[] nums, int k) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++){

if (map.containsKey(nums[i])){

if ((i - map.get(nums[i])) <= k){

return true;

}

}

map.put(nums[i], i);

}

return false;

}

}

# 22\_FlattenList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list, each element in the list can be a list or integer. flatten it into a simply list with integers.

Notice

If the element in the given list is a list, it can contain list too.

Have you met this question in a real interview?

Example

Given [1,2,[1,2]], return [1,2,1,2].

Given [4,[3,[2,[1]]]], return [4,3,2,1].

Challenge

Do it in non-recursive.

Method 1: recursion, DFS

/\*\*

\* // This is the interface that allows for creating nested lists.

\* // You should not implement it, or speculate about its implementation

\* public interface NestedInteger {

\*

\* // @return true if this NestedInteger holds a single integer,

\* // rather than a nested list.

\* public boolean isInteger();

\*

\* // @return the single integer that this NestedInteger holds,

\* // if it holds a single integer

\* // Return null if this NestedInteger holds a nested list

\* public Integer getInteger();

\*

\* // @return the nested list that this NestedInteger holds,

\* // if it holds a nested list

\* // Return null if this NestedInteger holds a single integer

\* public List<NestedInteger> getList();

\* }

\*/

public class Solution {

// @param nestedList a list of NestedInteger

// @return a list of integer

public List<Integer> flatten(List<NestedInteger> nestedList) {

List<Integer> result = new ArrayList<>();

for (NestedInteger item : nestedList){

if (item.isInteger()){

result.add(item.getInteger());

}else{

result.addAll(flatten(item.getList()));

}

}

return result;

}

}

Method 2:

http://www.jiuzhang.com/solutions/flatten-list/

# 22\_GenerateParentheses.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

[

"((()))",

"(()())",

"(())()",

"()(())",

"()()()"

]

https://leetcode.com/problems/generate-parentheses/solution/

class Solution {

public List<String> generateParenthesis(int n) {

List<String> result = new ArrayList<>();

char[] combination = new char[2\*n];

dfs(result, combination, 0);

return result;

}

private void dfs(List<String> result, char[] combination, int start){

if (start == combination.length){

if (isValid(combination)){

result.add(new String(combination));

}

}else{

combination[start] = '(';

dfs(result, combination, start+1);

combination[start] = ')';

dfs(result, combination, start+1);

}

}

private boolean isValid(char[] combination){

int balance = 0;

for (int i = 0; i < combination.length; i++){

if (combination[i] == '('){

balance++;

}else{

balance--;

}

if (balance < 0){

return false;

}

}

return (balance == 0);

}

}

Best solution:

class Solution {

public List<String> generateParenthesis(int n) {

List<String> res = new ArrayList<>();

helper(res, "", n, n);

return res;

}

private void helper(List<String> res, String str, int leftRemain, int rightRemain){

if (leftRemain == 0 && rightRemain == 0){

res.add(str);

return;

}

if (leftRemain > 0){

helper(res, str + "(", leftRemain - 1, rightRemain);

}

if (rightRemain > leftRemain){ //note that it is not rightRemain > 0 in order to eliminate invalid cases

helper(res, str + ")", leftRemain, rightRemain - 1);

}

}

}

# 220\_Contains.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers, find out whether there are two distinct indices i and j in the array such that

the absolute difference between nums[i] and nums[j] is at most t and the absolute difference between

i and j is at most k.

This problem requires to maintain a window of size k of the previous values that can be queried for value ranges.

The best data structure to do that is Binary Search Tree. As a result maintaining the tree of size k will result

in time complexity O(N lg K). In order to check if there exists any value of range abs(nums[i] - nums[j]) to simple

queries can be executed both of time complexity O(lg K)

To avoid overflow, cast to long

TreeMap function: floorKey(), ceilingKey(), firstKey(), lastKey(), higherKey(), lowerKey(), pollFirstEntry(), pollLastEntry()

TreeSet function: floor(), ceiling(), first(), last(), higher(), lower(), pollFirst(), pollLast()

Method 1: TreeSet

Time complexity: O(Nlogk)

Space complexity: O(k)

class Solution {

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {

TreeSet<Long> set = new TreeSet<>();

for (int i = 0; i < nums.length; i++){

if (i > k){

set.remove((long)nums[i-k-1]);

}

Long max = set.floor((long) nums[i]+t);

Long min = set.ceiling((long)nums[i]-t);

if (max != null && nums[i] <= max || min != null && nums[i] >= min){

return true;

}

set.add((long) nums[i]);

}

return false;

}

}

Method 2

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

class Pair {

int index;

long value;

public Pair (int index, long value){

this.index = index;

this.value = value;

}

}

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {

Pair[] pairs = new Pair[nums.length];

for (int i = 0; i< nums.length; i++){

pairs[i] = new Pair(i, (long) nums[i]);

}

Arrays.sort(pairs, new Comparator<Pair>(){

public int compare (Pair a, Pair b){

if (a.value != b.value){

return (int) (a.value - b.value);

}

return a.index - b.index;

}

});

for (int i = 0; i < nums.length; i++){

for (int j = i + 1; j < nums.length && pairs[j].value - pairs[i].value <= t; j++){

if (Math.abs(pairs[j].index - pairs[i].index) <= k){

return true;

}

}

}

return false;

}

}

# 221\_MaximalSquare.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

iven a 2D binary matrix filled with 0's and 1's, find the largest square containing only 1's and return its area.

For example, given the following matrix:

1 0 1 0 0

1 0 1 1 1

1 1 1 1 1

1 0 0 1 0

Return 4.

Method 1: 2D dynamic programming

class Solution {

public int maximalSquare(char[][] matrix) {

if (matrix == null || matrix.length == 0){

return 0;

}

int m = matrix.length;

int n = matrix[0].length;

int[][] dp = new int[m][n];//denote the max square side ending at i, j

int max = 0;

for (int i = 0; i < m; i++){

dp[i][0] = (int)(matrix[i][0] - '0');

max = Math.max(max, dp[i][0]);

}

for (int j = 0; j < n; j++){

dp[0][j] = (int)(matrix[0][j] - '0');

max = Math.max(max, dp[0][j]);

}

for (int i = 1; i < m; i++){

for (int j = 1; j < n; j++){

if (matrix[i][j] == '1'){

dp[i][j] = Math.min(dp[i-1][j-1], Math.min(dp[i-1][j], dp[i][j-1])) + 1;

max = Math.max(max, dp[i][j]);

}else{

dp[i][j] = 0;

}

}

}

return max \* max;

}

}

Why pick Math.min

https://www.youtube.com/watch?v=aYnEO53H4lw&t=247s

# 222\_count.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a complete binary tree, count the number of nodes.

Definition of a complete binary tree from Wikipedia:

In a complete binary tree every level, except possibly the last, is completely filled,

and all nodes in the last level are as far left as possible. It can have between 1 and 2h

nodes inclusive at the last level h.

https://leetcode.com/problems/count-complete-tree-nodes/discuss/61948/Accepted-Easy-Understand-Java-Solution/122365?page=1

<< and >> are bit manipulate operators, it means to move the 32 bit integer to left or to right, which essentially

does what pow(2,n) does. When the exponential base is 2, use bit moving operations are fast.

1 << leftDepth == 2 ^ leftDepth, leftDepth << 1 == leftDepth \* 2

https://leetcode.com/problems/count-complete-tree-nodes/discuss/61967/A-very-clear-recursive-solution-isn't-it

Method 1:

Time complexity: O(logN \* logN or depth \* depth)

Halve the tree in every recursive step, and there are O(logN) or height steps. So it O((logN)^2)

class Solution {

public int countNodes(TreeNode root) {

if (root == null){

return 0;

}

int leftDepth = getLeftDepth(root.left);

int rightDepth = getLeftDepth(root.right);

if (leftDepth == rightDepth){//left subtree is full, right subtree may be or may not be

return (1 << leftDepth) - 1 + countNodes(root.right) + 1;

}else{//right subtree is full, left subtree is not full

return (1 << rightDepth) - 1 + countNodes(root.left) + 1;

}

}

private int getLeftDepth(TreeNode root){

if (root == null){

return 0;

}

return 1 + getLeftDepth(root.left);

}

}

Method 2: TLE

Time complexity: O(NlogN)

Halve the tree in every step but there are O(N) steps. So it is O(NlogN)

class Solution {

public int countNodes(TreeNode root) {

if (root == null){

return 0;

}

return 1 + countNodes(root.left) + countNodes(root.right);

}

}

# 223\_RectangleArea.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the total area covered by two rectilinear rectangles in a 2D plane.

Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.

Rectangle Area

Assume that the total area is never beyond the maximum possible value of int.

class Solution {

public int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {

int innerL = Math.max(A,E);

int innerR = Math.max(innerL, Math.min(C,G));

int innerB = Math.max(B,F);

int innerT = Math.max(innerB, Math.min(D,H));

return (C-A)\*(D-B) + (G-E)\*(H-F) - (innerR - innerL)\*(innerT - innerB);

}

}

# 224\_Basic.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open ( and closing parentheses ), the plus + or minus sign -,

non-negative integers and empty spaces .

You may assume that the given expression is always valid.

Some examples:

"1 + 1" = 2

" 2-1 + 2 " = 3

"(1+(4+5+2)-3)+(6+8)" = 23

https://leetcode.com/problems/basic-calculator/discuss/62361/Iterative-Java-solution-with-stack

class Solution {

public int calculate(String s) {

int ans = 0;

int number = 0; //denote the current number

int sign = 1; //denote the previous sign

Stack<Integer> stack = new Stack<>(); // store the sign before "(" and the sum before "("

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (Character.isDigit(c)){

number = 10 \* number + (int) (c - '0');

}else if (c == '+'){

ans += sign \* number;

number = 0;

sign = 1;

}else if (c == '-'){

ans += sign \* number;

number = 0;

sign = -1;

}else if (c == '('){

stack.push(ans);

stack.push(sign);

ans = 0;

number = 0;

sign = 1;

}else if (c == ')'){

ans += sign \* number;

number = 0;

sign = 1;

ans \*= stack.pop();

ans += stack.pop();

}

}

ans += sign \* number;

return ans;

}

}

# 227\_Basic.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, +, -, \*, / operators and empty spaces .

The integer division should truncate toward zero.

You may assume that the given expression is always valid.

Some examples:

"3+2\*2" = 7

" 3/2 " = 1

" 3+5 / 2 " = 5

https://leetcode.com/problems/basic-calculator-ii/discuss/63003/Share-my-java-solution

class Solution {

public int calculate(String s) {

Stack<Integer> stack = new Stack<>();

int ans = 0;

int number = 0; //denote the current number

char sign = '+'; //denote the previous sign

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (Character.isDigit(c)){

number = 10 \* number + (int) (c - '0');

}

if (!Character.isDigit(c) && c != ' ' || i == s.length() - 1){

if (sign == '+'){

stack.push(number);

}else if (sign == '-'){

stack.push(-number);

}else if (sign == '\*'){

stack.push(stack.pop() \* number);

}else if (sign == '/'){

stack.push(stack.pop() / number);

}

sign = c;

number = 0;

}

}

while (!stack.isEmpty()){

ans += stack.pop();

}

return ans;

}

}

class Solution {

public int calculate(String s) {

Stack<Integer> stack = new Stack<>();

int ans = 0;

int number = 0; //denote the current number

char sign = '+'; //denote the previous sign

stack.push(number);

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (Character.isDigit(c)){

number = 10 \* number + (int) (c - '0');

}

if (!Character.isDigit(c) && c != ' ' || i == s.length() - 1){

if (sign == '+'){

ans += stack.peek();

stack.push(number);

}else if (sign == '-'){

ans += stack.peek();

stack.push(-number);

}else if (sign == '\*'){

stack.push(stack.pop() \* number);

}else if (sign == '/'){

stack.push(stack.pop() / number);

}

sign = c;

number = 0;

}

}

ans += stack.peek();

return ans;

}

}

Method 2: No stack

class Solution {

public int calculate(String s) {

int ans = 0;//store the result before the prevNum;

int number = 0; //denote the current number

int preNum = 0; //denote the previous number

char preSign = '+'; //denote the previous sign

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (Character.isDigit(c)){

number = 10 \* number + (int) (c - '0');

}

if (!Character.isDigit(c) && c != ' ' || i == s.length() - 1){

if (preSign == '+'){

ans += preNum;

preNum = number;

}else if (preSign == '-'){

ans += preNum;

preNum = -number;

}else if (preSign == '\*'){

preNum = preNum \* number;

}else if (preSign == '/'){

preNum = preNum / number;

}

preSign = c;

number = 0;

}

}

ans += preNum;

return ans;

}

}

# 228\_Summary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted integer array without duplicates, return the summary of its ranges.

Example 1:

Input: [0,1,2,4,5,7]

Output: ["0->2","4->5","7"]

Example 2:

Input: [0,2,3,4,6,8,9]

Output: ["0","2->4","6","8->9"]

Best solution:

class Solution {

public List<String> summaryRanges(int[] nums) {

List<String> res = new ArrayList<>();

int i = 0;

while (i < nums.length){

int j = i;

while (j + 1 < nums.length && nums[j] + 1 == nums[j+1]){

j++;

}

if (j == i){

res.add(String.valueOf(nums[i]));

}else{

res.add(nums[i] + "->" + nums[j]);

}

i = j + 1;

}

return res;

}

}

Method 1:

class Solution {

public List<String> summaryRanges(int[] nums) {

List<String> result = new ArrayList<>();

for (int i = 0, j = 0; j < nums.length; j++){

while (j+1 < nums.length && nums[j] + 1 == nums[j+1]){

j++;

}

if (i == j){

result.add(nums[i] + "");

}else{

result.add(nums[i] + "->" + nums[j]);

}

i = j + 1;

}

return result;

}

}

Method 2:

class Solution {

public List<String> summaryRanges(int[] nums) {

List<String> result = new ArrayList<>();

if (nums == null || nums.length == 0){

return result;

}

int len = nums.length;

if (len == 1){

result.add(String.valueOf(nums[0]));

return result;

}

int start = 0;

int end = 0;

while (end < len){

StringBuilder sb = new StringBuilder();

while (end + 1 < len){

if (nums[end] + 1 != nums[end+1]){

if (end == start){

sb.append(nums[start]);

}else{

sb.append(nums[start] + "->" + nums[end]);

}

break;

}

end++;

}

if (end + 1 == len){

if (nums[end] != nums[end-1] + 1){

sb.append(nums[end]);

}else{

sb.append(nums[start] + "->" + nums[end]);

}

}

result.add(sb.toString());

end++;

start = end;

}

return result;

}

}

# 229\_MajorityElementII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.

The algorithm should run in linear time and in O(1) space.

class Solution {

public List<Integer> majorityElement(int[] nums) {

List<Integer> res = new ArrayList<>();

int major1 = 0;

int major2 = 0;

int count1 = 0;

int count2 = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] == major1){

//must check nums[i] == major1 or major2 first before checking count1 == 0 or count2 == 0 to avoid major1 and major2 are the same

count1++;

}else if (nums[i] == major2){

count2++;

}else if (count1 == 0){

major1 = nums[i];

count1++;

}else if (count2 == 0){

major2 = nums[i];

count2++;

}else{

count1--;

count2--;

}

}

count1 = 0;

count2 = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] == major1){

count1++;

}else if (nums[i] == major2){

count2++;

}

}

if (count1 > nums.length / 3){

res.add(major1);

}

if (count2 > nums.length / 3){

res.add(major2);

}

return res;

}

}

# 230\_KthSmallestBST.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.

Note:

You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Follow up:

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently?

How would you optimize the kthSmallest routine?

Method 1:

Time complexity: O(n)

class Solution {

int ans = 0;

public int kthSmallest(TreeNode root, int k) {

int[] count = new int[1];

count[0] = k;

inorder(root, count);

return ans;

}

private void inorder(TreeNode root, int[] count){

if (root == null){

return;

}

inorder(root.left, count);

count[0]--;

if (count[0] == 0){

ans = root.val;

return;

}

inorder(root.right, count);

}

}

Best solution:

Method 2:

Time complexity: O(n)

class Solution {

public int kthSmallest(TreeNode root, int k) {

int[] result = new int[2];

result[0] = k;

inorder(root, result);

return result[1];

}

private void inorder(TreeNode root, int[] result){

if (root == null){

return;

}

inorder(root.left, result);

result[0]--;

if (result[0] == 0){

result[1] = root.val;

return;

}

inorder(root.right, result);

}

}

Method 3:

Time complexity: O(nlogn)

class Solution {

public int kthSmallest(TreeNode root, int k) {

int count = countNode(root.left);

if (k <= count){

return kthSmallest(root.left, k);

}else if (k > count + 1){

return kthSmallest(root.right, k - count - 1);

}

return root.val;

}

private int countNode(TreeNode root){

if (root == null){

return 0;

}

return 1 + countNode(root.left) + countNode(root.right);

}

}

Method 4:

https://leetcode.com/problems/validate-binary-search-tree/discuss/32112/Learn-one-iterative-inorder-traversal-apply-it-to-multiple-tree-questions-(Java-Solution)

Iteration:

O(n)

class Solution {

public int kthSmallest(TreeNode root, int k) {

Stack<TreeNode> stack = new Stack<>();

TreeNode node = root;

while (node != null || !stack.isEmpty()){

while (node != null){

stack.push(node);

node = node.left;

}

TreeNode curr = stack.pop();

k--;

if (k == 0){

return curr.val;

}

node = curr.right;

}

return -1;

}

}

# 231\_PowerTwo.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer, write a function to determine if it is a power of two.

Method 1:

time complexity: O(logn)

class Solution {

public boolean isPowerOfTwo(int n) {

if (n <= 0){

return false;

}

while (n > 1){

if (n % 2 != 0){

return false;

}

n /= 2;

}

return true;

}

}

https://leetcode.com/problems/power-of-two/discuss/63966/4-different-ways-to-solve-Iterative-Recursive-Bit-operation-Math

Method 2:

time complexity: O(1)

class Solution {

public boolean isPowerOfTwo(int n) {

if (n <= 0){

return false;

}

return (n & (n-1)) == 0;

}

}

Trick:

use (n&(n-1)) to check how many 1 in binary format

int count = 0;

while (n != 0){

n = n & (n - 1);

count++;

}

return count;

# 238\_ProductofArrayExceptSelf.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of n integers where n > 1, nums, return an array output such that output[i] is equal to

the product of all the elements of nums except nums[i].

Solve it without division and in O(n).

For example, given [1,2,3,4], return [24,12,8,6].

Follow up:

Could you solve it with constant space complexity? (Note: The output array does not count

as extra space for the purpose of space complexity analysis.)

Method:

the first scan: ans keeps the product of all numbers ahead of the current index;

the second scan: right keeps the product of all numbers behind the current index;

class Solution {

public int[] productExceptSelf(int[] nums) {

int[] ans = new int[nums.length];

ans[0] = 1;

for (int i = 1; i < nums.length; i++){

ans[i] = ans[i-1] \* nums[i-1];

}

int right = 1;

for (int i = nums.length - 1; i >=0; i--){

ans[i] \*= right;

right \*= nums[i];

}

return ans;

}

}

class Solution {

public int[] productExceptSelf(int[] nums) {

int n = nums.length;

int[] res = new int[n];

int prod = 1;

res[0] = 1;

for (int i = 0; i < n - 1; i++){

prod \*= nums[i];

res[i+1] = prod;

}

prod = 1;

for (int i = n - 1; i > 0; i--){

prod \*= nums[i];

res[i-1] \*= prod;

}

return res;

}

}

# 239\_SlidingWindowMaximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right.

You can only see the k numbers in the window. Each time the sliding window moves right by one position.

For example,

Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.

Window position Max

--------------- -----

[1 3 -1] -3 5 3 6 7 3

1 [3 -1 -3] 5 3 6 7 3

1 3 [-1 -3 5] 3 6 7 5

1 3 -1 [-3 5 3] 6 7 5

1 3 -1 -3 [5 3 6] 7 6

1 3 -1 -3 5 [3 6 7] 7

Therefore, return the max sliding window as [3,3,5,5,6,7].

Note:

You may assume k is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.

Follow up:

Could you solve it in linear time?

Method 1:

Time complexity; O(n^2)

class Solution {

public int[] maxSlidingWindow(int[] nums, int k) {

if (nums == null || nums.length == 0){

return nums;

}

int len = nums.length;

int[] result = new int[len - k + 1];

for (int i = 0; i < len-k+1; i++){

int max = Integer.MIN\_VALUE;

for (int j = i; j < i + k; j++){

max = Math.max(max, nums[j]);

}

result[i] = max;

}

return result;

}

}

Method 2: use deque O(n)

class Solution {

public int[] maxSlidingWindow(int[] nums, int k) {

if (nums == null || nums.length == 0){

return nums;

}

int len = nums.length;

int[] result = new int[len - k + 1];

LinkedList<Integer> doubleQ = new LinkedList<>(); //save index not number

for (int i = 0; i < len; i++){

if (!doubleQ.isEmpty() && doubleQ.peek() < i - k + 1){

doubleQ.poll();

}

while (!doubleQ.isEmpty() && nums[i] >= nums[doubleQ.peekLast()]){

//remove thoese elements that definitely not the result and push all candidates to peek()

doubleQ.pollLast();

}

doubleQ.offer(i);

if (i >= k - 1){

result[i-k+1] = nums[doubleQ.peek()];

}

}

return result;

}

}

Method 3: O(nlogk) use priority queue

class Solution {

public int[] maxSlidingWindow(int[] nums, int k) {

if (nums == null || nums.length == 0 || k == 0){

return new int[0];

}

int n = nums.length;

int[] res = new int[n-k+1];

Queue<Integer> pq = new PriorityQueue<Integer>(new Comparator<Integer>(){

public int compare (Integer i1, Integer i2){

return i2 - i1;

}

});

for (int i = 0; i < k; i++){

pq.offer(nums[i]);

}

res[0] = pq.peek();

for (int i = k; i < n; i++){

pq.remove(nums[i-k]); // take O(k)

pq.offer(nums[i]);

res[i-k+1] = pq.peek();

}

return res;

}

}

Best solution:

https://leetcode.com/problems/sliding-window-maximum/discuss/65881/O(n)-solution-in-Java-with-two-simple-pass-in-the-array?page=3

class Solution {

public int[] maxSlidingWindow(int[] nums, int k) {

if (nums == null || nums.length == 0 || k == 0){

return new int[0];

}

int n = nums.length;

int[] maxLeft = new int[n];

int[] maxRight = new int[n];

maxLeft[0] = nums[0];

maxRight[n-1] = nums[n-1];

for (int i = 1; i < nums.length; i++){

maxLeft[i] = i % k == 0 ? nums[i] : Math.max(maxLeft[i-1], nums[i]);

int j = n - 1 - i;

maxRight[j] = j % k == 0 ? nums[j] : Math.max(maxRight[j+1], nums[j]);

}

int[] res = new int[n-k+1];

for (int i = 0; i < n-k+1; i++){

res[i] = Math.max(maxRight[i], maxLeft[i+k-1]);

}

return res;

}

}

# 240\_SearchII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

Integers in each row are sorted in ascending from left to right.

Integers in each column are sorted in ascending from top to bottom.

For example,

Consider the following matrix:

[

[1, 4, 7, 11, 15],

[2, 5, 8, 12, 19],

[3, 6, 9, 16, 22],

[10, 13, 14, 17, 24],

[18, 21, 23, 26, 30]

]

Given target = 5, return true.

Given target = 20, return false.

Method:

Time complexity: m + n

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return false;

}

int r = matrix.length - 1;

int c = 0;

while (r >= 0 && c < matrix[0].length){

if (matrix[r][c] == target){

return true;

}else if (matrix[r][c] > target){

r--;

}else{

c++;

}

}

return false;

}

}

# 241\_Different.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string of numbers and operators, return all possible results from computing all the different possible

ways to group numbers and operators. The valid operators are +, - and \*.

Example 1

Input: "2-1-1".

((2-1)-1) = 0

(2-(1-1)) = 2

Output: [0, 2]

Example 2

Input: "2\*3-4\*5"

(2\*(3-(4\*5))) = -34

((2\*3)-(4\*5)) = -14

((2\*(3-4))\*5) = -10

(2\*((3-4)\*5)) = -10

(((2\*3)-4)\*5) = 10

Output: [-34, -14, -10, -10, 10]

Method 1: Divide and Conquer

class Solution {

public List<Integer> diffWaysToCompute(String input) {

List<Integer> result = new ArrayList<>();

for (int i = 0; i < input.length(); i++){

if (!Character.isDigit(input.charAt(i))){

String part1 = input.substring(0, i);

String part2 = input.substring(i+1);

List<Integer> res1 = diffWaysToCompute(part1);

List<Integer> res2 = diffWaysToCompute(part2);

for (int i1 : res1){

for(int i2 : res2){

char c = input.charAt(i);

switch (c){

case '+' :

result.add(i1 + i2);

break;

case '-':

result.add(i1 - i2);

break;

case '\*':

result.add(i1 \* i2);

break;

}

}

}

}

}

if (result.isEmpty()){

result.add(Integer.parseInt(input));

}

return result;

}

}

Method 2: Divide and Conquer + Memorization

class Solution {

Map<String, List<Integer>> map = new HashMap<>();

public List<Integer> diffWaysToCompute(String input) {

if (map.containsKey(input)){

return map.get(input);

}

List<Integer> result = new ArrayList<>();

for (int i = 0; i < input.length(); i++){

if (!Character.isDigit(input.charAt(i))){

String part1 = input.substring(0, i);

String part2 = input.substring(i+1);

List<Integer> res1 = diffWaysToCompute(part1);

List<Integer> res2 = diffWaysToCompute(part2);

for (int i1 : res1){

for(int i2 : res2){

char c = input.charAt(i);

switch (c){

case '+' :

result.add(i1 + i2);

break;

case '-':

result.add(i1 - i2);

break;

case '\*':

result.add(i1 \* i2);

break;

}

}

}

}

}

if (result.isEmpty()){

result.add(Integer.parseInt(input));

}

map.put(input, result);

return result;

}

}

# 243\_ShortestI.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

For example,

Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “coding”, word2 = “practice”, return 3.

Given word1 = "makes", word2 = "coding", return 1.

Note:

You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

class Solution {

public int shortestDistance(String[] words, String word1, String word2) {

int ans = Integer.MAX\_VALUE;

int word1Ind = -1;

int word2Ind = -1;

for (int i = 0; i < words.length; i++){

if (words[i].equals(word1)){

word1Ind = i;

}else if (words[i].equals(word2)){

word2Ind = i;

}

if (word1Ind != -1 && word2Ind != -1){

ans = Math.min(ans, Math.abs(word1Ind - word2Ind));

}

}

return ans;

}

}

## 244\_ShortestII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is a follow up of Shortest Word Distance. The only difference is now you are given the list of words

and your method will be called repeatedly many times with different parameters. How would you optimize it?

Design a class which receives a list of words in the constructor, and implements a method that takes two

words word1 and word2 and return the shortest distance between these two words in the list.

For example,

Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “coding”, word2 = “practice”, return 3.

Given word1 = "makes", word2 = "coding", return 1.

Note:

You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

In shortest( ) function, since list1 (size n) and list2 (size m) are sorted already,

we can use the idea of merge sort and perform the comparison in O(n + m) time, rather than O(n \* m).

class WordDistance {

private Map<String, List<Integer>> map;

public WordDistance(String[] words) {

map = new HashMap<String, List<Integer>>();

for (int i = 0; i < words.length; i++){

if (!map.containsKey(words[i])){

map.put(words[i], new ArrayList<Integer>());

}

map.get(words[i]).add(i);

}

}

public int shortest(String word1, String word2) {

List<Integer> l1 = map.get(word1);

List<Integer> l2 = map.get(word2);

int i = 0;

int j = 0;

int ans = Integer.MAX\_VALUE;

while (i < l1.size() && j < l2.size()){

ans = Math.min(ans, Math.abs(l1.get(i)-l2.get(j)));

if (l1.get(i) < l2.get(j)){

i++;

}else{

j++;

}

}

return ans;

}

}

/\*\*

\* Your WordDistance object will be instantiated and called as such:

\* WordDistance obj = new WordDistance(words);

\* int param\_1 = obj.shortest(word1,word2);

\*/

## 245\_ShortestIII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is a follow up of Shortest Word Distance. The only difference is now word1 could be the same as word2.

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

word1 and word2 may be the same and they represent two individual words in the list.

For example,

Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “makes”, word2 = “coding”, return 1.

Given word1 = "makes", word2 = "makes", return 3.

Note:

You may assume word1 and word2 are both in the list.

class Solution {

public int shortestWordDistance(String[] words, String word1, String word2) {

int word1Ind = -1;

int word2Ind = -1;

int ans = Integer.MAX\_VALUE;

boolean equal = false;

if (word1.equals(word2)){

equal = true;

}

for (int i = 0; i < words.length; i++){

if (words[i].equals(word1)){

if (equal && word1Ind != -1){

ans = Math.min(ans, i - word1Ind);

}

word1Ind = i;

}else if (words[i].equals(word2)){

if (equal && word2Ind != -1){

ans = Math.min(ans, i - word2Ind);

}

word2Ind = i;

}

if (word1Ind != -1 && word2Ind != -1){

ans = Math.min(ans, Math.abs(word1Ind - word2Ind));

}

}

return ans;

}

}

# 246\_Strobogrammatic.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to determine if a number is strobogrammatic. The number is represented as a string.

For example, the numbers "69", "88", and "818" are all strobogrammatic.

class Solution {

public boolean isStrobogrammatic(String num) {

int i = 0, j = num.length()-1;

String s = num;

while(i<=j){

if((s.charAt(i)=='1' && s.charAt(j)=='1')|| (s.charAt(i)=='8' && s.charAt(j)=='8')|| (s.charAt(i)=='0' && s.charAt(j)=='0')){

i++;

j--;

}

else if((s.charAt(i)=='6' && s.charAt(j)=='9')|| (s.charAt(i)=='9' && s.charAt(j)=='6')){

i++;

j--;

}

else{

return false;

}

}

return true;

}

}

## 247\_Strobogrammatic.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Find all strobogrammatic numbers that are of length = n.

For example,

Given n = 2, return ["11","69","88","96"].

Method: Recusion

Best solution:

class Solution {

public List<String> findStrobogrammatic(int n) {

return helper(n, n);

}

private List<String> helper(int len, int target){

if (step == 0){

return Arrays.asList("");

}

if (step == 1){

return Arrays.asList("1", "8", "0");

}

List<String> list = helper(len-2, target);

List<String> result = new ArrayList<>();

for (String str : list){

if (len != target){

result.add("0" + str + "0");

}

result.add("1" + str + "1");

result.add("6" + str + "9");

result.add("8" + str + "8");

result.add("9" + str + "6");

}

return result;

}

}

Method2 : iteration

public class Solution {

public List<String> findStrobogrammatic(int n) {

List<String> one = Arrays.asList("0", "1", "8"), two = Arrays.asList(""), r = two;

if(n%2 == 1)

r = one;

for(int i=(n%2)+2; i<=n; i+=2){

List<String> newList = new ArrayList<>();

for(String str : r){

if(i != n)

newList.add("0" + str + "0");

newList.add("1" + str + "1");

newList.add("6" + str + "9");

newList.add("8" + str + "8");

newList.add("9" + str + "6");

}

r = newList;

}

return r;

}

}

# 251\_Flatten2DVector.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement an iterator to flatten a 2d vector.

For example,

Given 2d vector =

[

[1,2],

[3],

[4,5,6]

]

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,2,3,4,5,6].

Follow up:

As an added challenge, try to code it using only iterators in C++ or iterators in Java.

http://www.jiuzhang.com/solutions/flatten-2d-vector/

http://massivealgorithms.blogspot.com/2015/11/buttercola-airbnb-2d-iterator-with.html

Airbnb phone interview: With remove()

public class Solution{

private List<List<Integer>> matrix;

private int rowId;

private int colId;

private int numRows;

public Solution(List<List<Integer>> matrix){

this.matrix = matrix;

rowId = 0;

colId = 0;

numRows = matrix.size();

}

public boolean hasNext(){

if (matrix == null || matrix.size() == 0){

return false;

}

while (rowId < numRows && (matrix.get(rowId) == null || matrix.get(rowId).isEmpty())){

rowId++;

}

return rowId < numRows;

}

public void next(){

int val = matrix.get(rowId).get(colId);

colId++;

if (colId == matrix.get(rowId).size()){

rowId++;

colId = 0;

}

return val;

}

public void remove(){

if (colId == 0){ // Case 1: if the element to remove is the last element of the row

List<Integer> list = matrix.get(rowId - 1);

list.remove(list.size() - 1);

//update rowId: this is just a possible option

if (list.size() == 0){

matrix.remove(list);

rowId--;

}

}else{ // Case 2: the element to remove is not the last element

List<Integer> list = matrix.get(rowId);

list.remove(colId - 1);

//update colId

colId--;

}

}

}

Best solution:

Method 2:

public class Vector2D implements Iterator<Integer> {

private Iterator<List<Integer>> i;

private Iterator<Integer> j;

public Vector2D(List<List<Integer>> vec2d) {

i = vec2d.iterator();

}

@Override

public Integer next() {

return j.next();

}

@Override

public boolean hasNext() {

while (i.hasNext() && (j == null || !j.hasNext())){

j = i.next().iterator();

}

if (j == null){

return false;

}

return j.hasNext();

}

}

/\*\*

\* Your Vector2D object will be instantiated and called as such:

\* Vector2D i = new Vector2D(vec2d);

\* while (i.hasNext()) v[f()] = i.next();

\*/

public class Vector2D implements Iterator<Integer> {

Stack<Integer> stack;

public Vector2D(List<List<Integer>> vec2d) {

stack = new Stack<Integer>();

pushListToStack(vec2d);

}

@Override

public Integer next() {

if (!hasNext()){

return null;

}

return stack.pop();

}

@Override

public boolean hasNext() {

return !stack.isEmpty();

}

private void pushListToStack(List<List<Integer>> list){

Stack<Integer> temp = new Stack<>();

for (List<Integer> l : list){

for (Integer n : l){

temp.push(n);

}

}

while (!temp.isEmpty()){

stack.push(temp.pop());

}

}

}

/\*\*

\* Your Vector2D object will be instantiated and called as such:

\* Vector2D i = new Vector2D(vec2d);

\* while (i.hasNext()) v[f()] = i.next();

\*/

# 254\_Factor.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Numbers can be regarded as product of its factors. For example,

8 = 2 x 2 x 2;

= 2 x 4.

Write a function that takes an integer n and return all possible combinations of its factors.

Note:

You may assume that n is always positive.

Factors should be greater than 1 and less than n.

Examples:

input: 1

output:

[]

input: 37

output:

[]

input: 12

output:

[

[2, 6],

[2, 2, 3],

[3, 4]

]

input: 32

output:

[

[2, 16],

[2, 2, 8],

[2, 2, 2, 4],

[2, 2, 2, 2, 2],

[2, 4, 4],

[4, 8]

]

Similar as combination sum

class Solution {

public List<List<Integer>> getFactors(int n) {

List<List<Integer>> result = new ArrayList<>();

int[] nums = new int[n/2];

for (int i = 0; i < nums.length; i++){

nums[i] = i + 2;

}

dfs(result, new ArrayList<Integer>(), n, 0, nums);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int target, int start, int[] nums){

if (target == 1){

if (item.size() > 1){

result.add(new ArrayList<Integer>(item));

}

return;

}

for (int i = start; i < nums.length; i++){

if (target % nums[i] == 0){

item.add(nums[i]);

dfs(result, item, target/nums[i], i, nums);

item.remove(item.size() - 1);

}

}

}

}

Method 2: best solution

public List<List<Integer>> getFactors(int n) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

helper(result, new ArrayList<Integer>(), n, 2);

return result;

}

public void helper(List<List<Integer>> result, List<Integer> item, int n, int start){

if (n <= 1) {

if (item.size() > 1) {

result.add(new ArrayList<Integer>(item));

}

return;

}

for (int i = start; i <= n; ++i) {

if (n % i == 0) {

item.add(i);

helper(result, item, n/i, i);

item.remove(item.size()-1);

}

}

}

# 255\_Verify.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.

You may assume each number in the sequence is unique.

Follow up:

Could you do it using only constant space complexity?

https://leetcode.com/problems/verify-preorder-sequence-in-binary-search-tree/discuss/68142/Java-O(n)-and-O(1)-extra-space

https://www.geeksforgeeks.org/check-if-a-given-array-can-represent-preorder-traversal-of-binary-search-tree/

Similar as Next Greater Element :

find next greater element and after finding next greater, if we find a smaller element, then return false.

Method 1: monotonic stack

time complexity: O(n)

space complexity: O(n)

class Solution {

public boolean verifyPreorder(int[] preorder) {

Stack<Integer> monotonicStack = new Stack<>();

int root = Integer.MIN\_VALUE;

for (int p: preorder){

if (p < root){

return false;

}

while (!monotonicStack.isEmpty() && p > monotonicStack.peek()){

root = monotonicStack.pop(); // note that the order of root travesal is the in order BST traversal

}

monotonicStack.push(p);

}

return true;

}

}

Method 2: use array as monotonic stack

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public boolean verifyPreorder(int[] preorder) {

int i = -1;

int low = Integer.MIN\_VALUE;

for (int p: preorder){

if (p < low){

return false;

}

while (i >= 0 && p > preorder[i]){

low = preorder[i--];

}

preorder[++i] = p;

}

return true;

}

}

Method 3:

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public boolean verifyPreorder(int[] preorder) {

return verify(preorder, 0, preorder.length - 1);

}

private boolean verify(int[] nums, int start, int end){

if (start >= end){

return true;

}

int root = nums[start];

int indexOfFirstGreater = -1;

for (int i = start + 1; i <= end; i++){

if (indexOfFirstGreater == -1 && nums[i] > root){

indexOfFirstGreater = i;

}

if (indexOfFirstGreater != -1 && nums[i] < root){

return false;

}

}

if (indexOfFirstGreater == -1){

return verify(nums, start+1, end);

}

return verify(nums, start+1, indexOfFirstGreater-1) && verify(nums, indexOfFirstGreater, end);

}

}

# 256\_paint.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\*

\* There are a row of n houses, each house can be painted with one of the three colors: red, blue or green.

The cost of painting each house with a certain color is different. You have to paint all the houses such that no

two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a n x 3 cost matrix. For example,

costs[0][0] is the cost of painting house 0 with color red; costs[1][2] is the cost of painting house 1 with color

green, and so on... Find the minimum cost to paint all houses.

Note:

All costs are positive integers.

\*/

Method: top-down dp

public int minCost(int[][] costs) {

int n = cost.length;

int r = 0;

int g = 0;

int b = 0;

int preR = 0;

int preG = 0;

int preB = 0;

for (int i = 0; i < n; i++) {

r = costs[i][0] + Math.min(preG, preB);

g = costs[i][1] + Math.min(preR, preB);

b = costs[i][2] + Math.min(preR, preG);

preR = r;

preG = g;

preB = b;

}

return Math.min(r, Math.min(g,b));

}

Method 2: generalization

public int minCostII(int[][] costs) {

if (costs == null || costs.length == 0){

return 0;

}

int n = costs.length;

int k = costs[0].length;

int[][] dp = new int[n+1][k];

for (int i = 1; i <= n; i++){

for (int j = 0; j < k; j++){

int min = Integer.MAX\_VALUE;

for (int m = 0; m < k; m++){

if (j != m){

min = Math.min(min, dp[i-1][m]);

}

}

if (min != Integer.MAX\_VALUE){

dp[i][j] = min + costs[i-1][j];

}else{

dp[i][j] = costs[i-1][j];

}

}

}

int ans = Integer.MAX\_VALUE;

for (int i = 0; i < k; i++){

ans = Math.min(ans, dp[n][i]);

}

return ans;

}

Method 2: generalization + rolling array

public int minCostII(int[][] costs) {

if (costs == null || costs.length == 0){

return 0;

}

int n = costs.length;

int k = costs[0].length;

int[][] dp = new int[2][k];

for (int i = 1; i <= n; i++){

for (int j = 0; j < k; j++){

int min = Integer.MAX\_VALUE;

for (int m = 0; m < k; m++){

if (j != m){

min = Math.min(min, dp[(i-1)%2][m]);

}

}

if (min != Integer.MAX\_VALUE){

dp[i%2][j] = min + costs[i-1][j];

}else{

dp[i%2][j] = costs[i-1][j];

}

}

}

int ans = Integer.MAX\_VALUE;

for (int i = 0; i < k; i++){

ans = Math.min(ans, dp[n%2][i]);

}

return ans;

}

}

# 258\_Add.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-negative integer num, repeatedly add all its digits until the result has only one digit.

For example:

Given num = 38, the process is like: 3 + 8 = 11, 1 + 1 = 2. Since 2 has only one digit, return it.

Follow up:

Could you do it without any loop/recursion in O(1) runtime?

Method 1:

class Solution {

public int addDigits(int num) {

while (num > 9){

int sum = 0;

while (num > 0){

sum += num % 10;

num /= 10;

}

num = sum;

}

return num;

}

}

Method 2:

class Solution {

public int addDigits(int num) {

return num == 0 ? 0 : (num % 9 == 0 ? 9 : num % 9);

}

}

abc%9 = a \* (100 % 9) + b \* (10 %9) + c % 9 = a + b + c

https://leetcode.com/problems/add-digits/discuss/68588/1-line-Java-Solution

# 259\_3SumSmaller.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of n integers nums and a target, find the number of index triplets i, j, k with 0 <= i < j < k < n

that satisfy the condition nums[i] + nums[j] + nums[k] < target.

For example, given nums = [-2, 0, 1, 3], and target = 2.

Return 2. Because there are two triplets which sums are less than 2:

[-2, 0, 1]

[-2, 0, 3]

Follow up:

Could you solve it in O(n2) runtime?

class Solution {

public int threeSumSmaller(int[] nums, int target) {

int ans = 0;

Arrays.sort(nums);

for (int i = nums.length - 1; i >= 2; i--){

int left = 0;

int right = i - 1;

while (left < right){

if (nums[left] + nums[right] < target - nums[i]){

ans += right - left;

left++;

}else{

right--;

}

}

}

return ans;

}

}

class Solution {

public int threeSumSmaller(int[] nums, int target) {

int ans = 0;

Arrays.sort(nums);

for (int i = 0; i < nums.length - 2; i++){

int left = i+1;

int right = nums.length - 1;

while (left < right){

if (nums[left] + nums[right] < target - nums[i]){

ans += right - left;

left++;

}else{

right--;

}

}

}

return ans;

}

}

# 26\_RemoveDuplicatesfromSortedArray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted array, remove the duplicates in-place such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this by modifying the input array in-place with O(1) extra memory.

Example:

Given nums = [1,1,2],

Your function should return length = 2, with the first two elements of nums being 1 and 2 respectively.

It doesn't matter what you leave beyond the new length.

class Solution {

public int removeDuplicates(int[] nums) {

int index = 0;

for (int i = 0 ; i < nums.length; i++){

if (nums[i] != nums[index]){

nums[++index] = nums[i];

}

}

return index + 1;

}

}

Method 2: inspired by

https://github.com/optimisea/Leetcode/blob/master/Java/80\_RemoveDuplicatesII.java

class Solution {

public int removeDuplicates(int[] nums) {

int index = 0;

for (int i = 0; i < nums.length; i++){

if (index < 1 || nums[index-1] != nums[i]){

nums[index++] = nums[i];

}

}

return index;

}

}

# 260\_Single.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example:

Given nums = [1, 2, 1, 3, 2, 5], return [3, 5].

Note:

The order of the result is not important. So in the above example, [5, 3] is also correct.

Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

https://leetcode.com/problems/single-number-iii/discuss/68900/Accepted-C++Java-O(n)-time-O(1)-space-Easy-Solution-with-Detail-Explanations

class Solution {

public int[] singleNumber(int[] nums) {

int diff = 0;

for (int i = 0; i < nums.length; i++){

diff ^= nums[i];

}

// diff &= -diff;

diff = Integer.lowestOneBit(diff);

int[] result = {0, 0};

for (int i = 0; i < nums.length; i++){

if ((nums[i] & diff) == 0){

result[0] ^= nums[i];

}else{

result[1] ^= nums[i];

}

}

return result;

}

}

class Solution {

public int[] singleNumber(int[] nums) {

int diff = 0;

for (int i = 0; i < nums.length; i++){

diff ^= nums[i];

}

// diff &= -diff;

// diff = Integer.lowestOneBit(diff);

int bit = 0;

while (diff != 0){ // find the lowest bit 1

if ((diff & 1) == 1){

diff = (1 << bit);

break;

}

diff = (diff >> 1);

bit++;

}

int[] result = {0, 0};

for (int i = 0; i < nums.length; i++){

if ((nums[i] & diff) == 0){

result[0] ^= nums[i];

}else{

result[1] ^= nums[i];

}

}

return result;

}

}

Better version:

class Solution {

public int[] singleNumber(int[] nums) {

int diff = 0;

for (int num : nums){

diff ^= num;

}

int bit = 0;

while (bit < 32){

if (((diff >> bit) & 1) == 1){

diff = (1 << bit);

break;

}

bit++;

}

int[] res = new int[2];

for (int num : nums){

if ((diff & num) == 0){

res[0] ^= num;

}else{

res[1] ^= num;

}

}

return res;

}

}

class Solution {

public int[] singleNumber(int[] nums) {

int diff = 0;

for (int num : nums){

diff ^= num;

}

for (int i = 0; i < 32; i++){

if (((diff >> i) & 1) == 1){

diff = (1 << i);

break;

}

}

int first = 0;

int second = 0;

for (int num : nums){

if ((num & diff) == 0){

first ^= num;

}else{

second ^= num;

}

}

return new int[]{first, second};

}

}

# 263\_UglyNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a program to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5.

For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7.

Note that 1 is typically treated as an ugly number.

class Solution {

public boolean isUgly(int num) {

if (num <= 0){

return false;

}

for (int i = 2; i <= 5 ; i++){

if (i == 4){

continue;

}

while (num % i == 0){

num /= i;

}

}

return num == 1;

}

}

## 264\_UglyNumberII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a program to find the n-th ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5.

For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.

Note that 1 is typically treated as an ugly number, and n does not exceed 1690.

class Solution {

public int nthUglyNumber(int n) {

Queue<Long> pq = new PriorityQueue<Long>();

Set<Long> set = new HashSet<Long>();

long[] primes = new long[3];

primes[0] = Long.valueOf(2);

primes[1] = Long.valueOf(3);

primes[2] = Long.valueOf(5);

set.add(Long.valueOf(1));

pq.offer(Long.valueOf(1));

Long v = Long.valueOf(1);

for (int i = 1 ; i <= n; i++){

v = pq.poll();

for (int j = 0; j < 3; j++){

if (!set.contains(v \* primes[j])){

set.add(v\*primes[j]);

pq.offer(v\*primes[j]);

}

}

}

return v.intValue();

}

}

Dynamic programming: better solution

https://leetcode.com/problems/ugly-number-ii/discuss/69364/My-16ms-C++-DP-solution-with-short-explanation

the concept of merge sort

the same as super ugly number

https://github.com/optimisea/Leetcode/blob/master/Java/313\_Super.java

class Solution {

public int nthUglyNumber(int n) {

int[] res = new int[n];

int[] indices = new int[3];//hold the index of res[i] for each prime, i is increased by 1 whenever that prime is used

int[] primes = new int[3];

primes[0] = 2;

primes[1] = 3;

primes[2] = 5;

res[0] = 1;

for (int i = 1; i < n; i++){

res[i] = Integer.MAX\_VALUE;

for (int j = 0; j < 3; j++){

res[i] = Math.min(res[i], primes[j] \* res[indices[j]]);

}

for (int j = 0; j < 3; j++){

if (res[i] == primes[j] \* res[indices[j]]){

indices[j]++;

}

}

}

return res[n-1];

}

}

# 265\_Paint.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are a row of n houses, each house can be painted with one of the k colors. The cost of painting each house with a

certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a n x k cost matrix. For example, costs[0][0]

is the cost of painting house 0 with color 0; costs[1][2] is the cost of painting house 1 with color 2, and so on...

Find the minimum cost to paint all houses.

Note:

All costs are positive integers.

Example:

Input: [[1,5,3],[2,9,4]]

Output: 5

Explanation: Paint house 0 into color 0, paint house 1 into color 2. Minimum cost: 1 + 4 = 5;

Or paint house 0 into color 2, paint house 1 into color 0. Minimum cost: 3 + 2 = 5.

Follow up:

Could you solve it in O(nk) runtime?

Method 1: Best solution Very Good Analysis

Time complexity: O(nk)

Space complexity: O(1)

Explanation: dp[i][j] represents the min paint cost from house 0 to house i when house i use color j;

The formula will be dp[i][j] = Math.min(any k!= j| dp[i-1][k]) + costs[i][j].

Take a closer look at the formula, we don't need an array to represent dp[i][j], we only need to know the min cost to

the previous house of any color and if the color j is used on previous house to get prev min cost, use the second min

cost that are not using color j on the previous house. So I have three variable to record: prevMin, prevMinColor, prevSecondMin.

and the above formula will be translated into:

dp[currentHouse][currentColor] = (currentColor == prevMinColor? prevSecondMin: prevMin) + costs[currentHouse][currentColor].

class Solution {

public int minCostII(int[][] costs) {

if (costs == null || costs.length == 0){

return 0;

}

int n = costs.length;

int k = costs[0].length;

int prevMin = 0;

int prevSecMin = 0;

int prevMinInd = -1;

for (int i = 0; i < n; i++){

int currMin = Integer.MAX\_VALUE;

int currSecMin = Integer.MAX\_VALUE;

int currMinInd = -1;

for (int j = 0; j < k; j++){

int val = costs[i][j] + (j == prevMinInd ? prevSecMin : prevMin);

if (val < currMin){

currSecMin = currMin;

currMin = val;

currMinInd = j;

}else if (val < currSecMin){

currSecMin = val;

}

}

prevMin = currMin;

prevSecMin = currSecMin;

prevMinInd = currMinInd;

}

return prevMin;

}

}

Method 2:

Time complexity: O(n\*k\*k)

Space complexity: O(n\*k)

public int minCostII(int[][] costs) {

if (costs == null || costs.length == 0){

return 0;

}

int n = costs.length;

int k = costs[0].length;

int[][] dp = new int[n+1][k];

for (int i = 1; i <= n; i++){

for (int j = 0; j < k; j++){

int min = Integer.MAX\_VALUE;

for (int m = 0; m < k; m++){

if (j != m){

min = Math.min(min, dp[i-1][m]);

}

}

if (min != Integer.MAX\_VALUE){

dp[i][j] = min + costs[i-1][j];

}else{

dp[i][j] = costs[i-1][j];

}

}

}

int ans = Integer.MAX\_VALUE;

for (int i = 0; i < k; i++){

ans = Math.min(ans, dp[n][i]);

}

return ans;

}

Method 3: rolling array

Time complexity: O(n\*k\*k)

Space complexity: O(k)

public int minCostII(int[][] costs) {

if (costs == null || costs.length == 0){

return 0;

}

int n = costs.length;

int k = costs[0].length;

int[][] dp = new int[2][k];

for (int i = 1; i <= n; i++){

for (int j = 0; j < k; j++){

int min = Integer.MAX\_VALUE;

for (int m = 0; m < k; m++){

if (j != m){

min = Math.min(min, dp[(i-1)%2][m]);

}

}

if (min != Integer.MAX\_VALUE){

dp[i%2][j] = min + costs[i-1][j];

}else{

dp[i%2][j] = costs[i-1][j];

}

}

}

int ans = Integer.MAX\_VALUE;

for (int i = 0; i < k; i++){

ans = Math.min(ans, dp[n%2][i]);

}

return ans;

}

}

# 268\_Missing.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.

Example 1:

Input: [3,0,1]

Output: 2

Example 2:

Input: [9,6,4,2,3,5,7,0,1]

Output: 8

Note:

Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

Method 1:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int missingNumber(int[] nums) {

Set<Integer> set = new HashSet<>();

for (int i : nums){

set.add(i);

}

int n = nums.length + 1;

for (int i = 0; i < n; i++){

if (!set.contains(i)){

return i;

}

}

return -1;

}

}

Method 2:

Time complexity: O(nlogn)

Space complexity: O(1)

class Solution {

public int missingNumber(int[] nums) {

Arrays.sort(nums);

if (nums[nums.length-1] != nums.length){

return nums.length;

}else if (nums[0] != 0){

return 0;

}

for (int i = 0; i < nums.length; i++){

if (nums[i] != i){

return i;

}

}

return -1;

}

}

class Solution {

public int missingNumber(int[] nums) {

Arrays.sort(nums);

for (int i = 0; i < nums.length; i++){

if (nums[i] != i){

return i;

}

}

return nums.length;

}

}

Method 3:

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public int missingNumber(int[] nums) {

int ans = nums.length;

for (int i = 0; i < nums.length; i++){

ans ^= i;

ans ^= nums[i];

}

return ans;

}

}

Because we know that nums contains nnn numbers and that it is missing exactly one number on the range [0..n−1][0..n-1][0..n−1],

we know that nnn definitely replaces the missing number in nums. Therefore, if we initialize an integer to nnn and XOR it with every

index and value, we will be left with the missing number. Consider the following example (the values have been sorted for intuitive

convenience, but need not be):

Index 0 1 2 3

Value 0 1 3 4

x ^ x = 0;

x ^ 0 = x;

Method 4: Gauss' Formula, sum from 1 to n = (n+1) \* n / 2;

class Solution {

public int missingNumber(int[] nums) {

int n = nums.length;

int expectedSum = (n+1) \* n / 2;

int sum = 0;

for (int i : nums){

sum += i;

}

return expectedSum - sum;

}

}

Method 3: Swap

class Solution {

public int missingNumber(int[] nums) {

int n = nums.length;

int i = 0;

int res = -1;

while(i < n){

if (nums[i] == n || nums[i] == i){

if (nums[i] == n){

res = i;

}

i++;

continue;

}

swap(nums, i, nums[i]);

}

return res == -1 ? n : res;

}

private void swap(int[] A, int i, int j){

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

# 269\_AlienDictionary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a new alien language which uses the latin alphabet. However, the order among

letters are unknown to you. You receive a list of non-empty words from the dictionary,

where words are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.

Example 1:

Given the following words in dictionary,

[

"wrt",

"wrf",

"er",

"ett",

"rftt"

]

The correct order is: "wertf".

Example 2:

Given the following words in dictionary,

[

"z",

"x"

]

The correct order is: "zx".

Example 3:

Given the following words in dictionary,

[

"z",

"x",

"z"

]

The order is invalid, so return "".

Note:

You may assume all letters are in lowercase.

You may assume that if a is a prefix of b, then a must appear before b in the given dictionary.

If the order is invalid, return an empty string.

There may be multiple valid order of letters, return any one of them is fine.

Method: Topological sort

First, build a degree map for each character in all the words:

w:0

r:0

t:0

f:0

e:0

Then build the hashmap by comparing the adjacent words, the first character that is different between two adjacent words

reflect the lexicographical order. For example:

"wrt",

"wrf",

first different character is 3rd letter, so t comes before f

"wrf",

"er",

first different character is 1rd letter, so w comes before e

The characters in set come after the key. x->y means letter x comes before letter y. x -> set: y,z,t,w means x comes

before all the letters in the set. The final HashMap “map” looks like.

t -> set: f

w -> set: e

r -> set: t

e -> set: r

and final HashMap “degree” looks like, the number means “how many letters come before the key”:

w:0

r:1

t:1

f:1

e:1

Then use Kahn’s aglorithm to do topological sort. This is essentially BFS.

https://en.wikipedia.org/wiki/Topological\_sorting

indegree: Map<Integer, Integer> indegree, which stores the count of numbers that have higher level of the key

priority map: Map<Integer, Set<Integer>> map, which stores the numbers that have lower level of the key

class Solution {

public String alienOrder(String[] words) {

if (words == null || words.length == 0){

return "";

}

Map<Character, Set<Character>> map = new HashMap<>(); //priority map

Map<Character, Integer> indegree = new HashMap<>(); //indegree map

//1. build indgree map and priority map

for (String word : words){

for (char c : word.toCharArray()){

indegree.put(c, 0);

}

}

for (int i = 0; i < words.length - 1; i++){

String curr = words[i];

String next = words[i+1];

int len = Math.min(curr.length(), next.length());

for (int j = 0; j < len; j++){

if (curr.charAt(j) != next.charAt(j)){

char c1 = curr.charAt(j);

char c2 = next.charAt(j);

if (!map.containsKey(c1)){

map.put(c1, new HashSet<Character>());

}

if (!map.get(c1).contains(c2)){

map.get(c1).add(c2);

indegree.put(c2, indegree.get(c2) + 1);

}

break;

}

}

}

//2. topological sort (BFS)

StringBuilder sb = new StringBuilder();

Queue<Character> queue = new LinkedList<>();

for (Character c : indegree.keySet()){

if (indegree.get(c) == 0){

queue.offer(c);

}

}

while (!queue.isEmpty()){

char ch = queue.poll();

sb.append(ch);

if (map.containsKey(ch)){

for (Character c : map.get(ch)){

indegree.put(c, indegree.get(c) - 1);

if (indegree.get(c) == 0){

queue.offer(c);

}

}

}

}

if (sb.length() != indegree.size()){

return "";

}

return sb.toString();

}

}

# 27\_RemoveElement.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array and a value, remove all instances of that value in-place and return the new length.

Do not allocate extra space for another array, you must do this by modifying the input array in-place with O(1) extra memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

Example:

Given nums = [3,2,2,3], val = 3,

Your function should return length = 2, with the first two elements of nums being 2.

class Solution {

public int removeElement(int[] nums, int val) {

int index = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] != val){

nums[index++] = nums[i];

}

}

return index;

}

}

# 270\_ClosestBinarySearchTreeValue.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note:

Given target value is a floating point.

You are guaranteed to have only one unique value in the BST that is closest to the target.

Method 1:

class Solution {

public int closestValue(TreeNode root, double target) {

if (root.val > target){

if (root.left == null){

return root.val;

}else{

int leftclosestValue = closestValue(root.left, target);

if (Math.abs(leftclosestValue - target) < Math.abs(root.val - target)){

return leftclosestValue;

}else{

return root.val;

}

}

}else if (root.val < target){

if (root.right == null){

return root.val;

}else{

int rightclosestValue = closestValue(root.right, target);

if (Math.abs(root.val - target) > Math.abs(rightclosestValue - target)){

return rightclosestValue;

}else{

return root.val;

}

}

}

return root.val;

}

}

Method 2:

class Solution {

public int closestValue(TreeNode root, double target) {

TreeNode next = target > root.val ? root.right : root.left;

if (next == null){

return root.val;

}

int nextClosesetValue = closestValue(next, target);

return Math.abs(nextClosesetValue - target) < Math.abs(root.val - target) ? nextClosesetValue : root.val;

}

}

Best solution

Method 3:

class Solution {

public int closestValue(TreeNode root, double target) {

int closest = root.val;

while (root != null){

if (Math.abs(root.val - target) < Math.abs(closest - target)){

closest = root.val;

}

root = target < root.val ? root.left : root.right;

}

return closest;

}

}

Method 4: BST inorder iteration;

public int closestValue(TreeNode root, double target){

Stack<TreeNode> stack = new Stack<>();

TreeNode node = root;

int res = Integer.MAX\_VALUE;

double diff = Double.MAX\_VALUE;

while (node != null || !stack.isEmpty()){

while (node != null){

stack.push(node);

node = node.left;

}

TreeNode curr = stack.pop();

if (Math.abs(curr.val - target) < diff){

diff = Math.abs(curr.val - target);

res = curr.val;

}

node = curr.right;

}

return res;

}

# 273\_Integer.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Convert a non-negative integer to its english words representation. Given input is guaranteed to be less than 231 - 1.

Example 1:

Input: 123

Output: "One Hundred Twenty Three"

Example 2:

Input: 12345

Output: "Twelve Thousand Three Hundred Forty Five"

Example 3:

Input: 1234567

Output: "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"

Example 4:

Input: 1234567891

Output: "One Billion Two Hundred Thirty Four Million Five Hundred Sixty Seven Thousand Eight Hundred Ninety One"

class Solution {

private final String[] lessTwenty = new String[] {"", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};

private final String[] tens = new String[] {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy","Eighty", "Ninety"};

public String numberToWords(int num) {

if (num == 0){

return "Zero";

}

return helper(num);

}

private String helper(int num){

String result = "";

if (num < 20){

result = lessTwenty[num];

}else if (num < 100){

result = tens[num/10] + " " + helper(num%10);

}else if (num < 1000){

result = helper(num/100) + " Hundred " + helper(num%100);

}else if (num < 1000000){

result = helper(num/1000) + " Thousand " + helper(num%1000);

}else if (num < 1000000000){

result = helper(num/1000000) + " Million " + helper(num%1000000);

}else{

result = helper(num/1000000000) + " Billion " + helper(num%1000000000);

}

return result.trim();

}

}

# 274\_H-Index.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of citations (each citation is a non-negative integer) of a researcher,

write a function to compute the researcher's h-index.

According to the definition of h-index on Wikipedia: "A scientist has index h if h of

his/her N papers have at least h citations each, and the other N − h papers have no more than h citations each."

For example, given citations = [3, 0, 6, 1, 5], which means the researcher has 5 papers

in total and each of them had received 3, 0, 6, 1, 5 citations respectively. Since the researcher

has 3 papers with at least 3 citations each and the remaining two with no more than 3 citations each, his h-index is 3.

Note: If there are several possible values for h, the maximum one is taken as the h-index.

class Solution {

public int hIndex(int[] citations) {

if (citations == null || citations.length == 0){

return 0;

}

int n = citations.length;

int[] candidate = new int[n+1];

for (int i = 0; i < n; i++){

if (citations[i] >= n){

candidate[n]++;

}else{

candidate[citations[i]]++;

}

}

int ans = 0;

int sum = 0;

for (int i = n; i >= 0; i--){

sum += candidate[i];

if (sum >= i){

return i;

}

}

return ans;

}

}

# 275\_Hindex.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Follow up for H-Index: What if the citations array is sorted in ascending order? Could you optimize your algorithm?

The idea is to search for the first index from the sorted array so that :

citations[index] >= length(citations) - index.

And return (length - index) as the result.

class Solution {

public int hIndex(int[] citations) {

if (citations == null || citations.length == 0){

return 0;

}

int start = 0;

int end = citations.length - 1;

int len = citations.length;

int target = 0;

while (start <= end){

int mid = start + (end - start)/2;

target = len - mid;

if (citations[mid] >= target){

end = mid - 1;

}else{

start = mid + 1;

}

}

return len - start;

}

}

# 276\_Paint.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence.

Note:

n and k are non-negative integers.

Example:

Input: n = 3, k = 2

Output: 6

Explanation: Take c1 as color 1, c2 as color 2. All possible ways are:

post1 post2 post3

----- ----- ----- -----

1 c1 c1 c2

2 c1 c2 c1

3 c1 c2 c2

4 c2 c1 c1

5 c2 c1 c2

6 c2 c2 c1

We divided it into two cases.

the last two posts have the same color, the number of ways to paint in this case is sameColorCounts.

the last two posts have different colors, and the number of ways in this case is diffColorCounts.

The reason why we have these two cases is that we can easily compute both of them, and that is all I do.

When adding a new post, we can use the same color as the last one (if allowed) or different color. If we use different color,

there're k-1 options, and the outcomes shoule belong to the diffColorCounts category.

If we use same color, there's only one option, and we can only do this when the last two have different colors

(which is the diffColorCounts). There we have our induction step.

Here is an example, let's say we have 3 posts and 3 colors. The first two posts we have 9 ways to do them,

(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3). Now we know that

diffColorCounts = 6;

And

sameColorCounts = 3;

Now for the third post, we can compute these two variables like this:

If we use different colors than the last one (the second one), these ways can be added into diffColorCounts,

so if the last one is 3, we can use 1 or 2, if it's 1, we can use 2 or 3, etc. Apparently there are

(diffColorCounts + sameColorCounts) \* (k-1) possible ways.

If we use the same color as the last one, we would trigger a violation in these three cases (1,1,1), (2,2,2) and (3,3,3).

This is because they already used the same color for the last two posts. So is there a count that rules out these kind of cases?

YES, the diffColorCounts. So in cases within diffColorCounts, we can use the same color as the last one without worrying about

triggering the violation. And now as we append a same-color post to them,

the former diffColorCounts becomes the current sameColorCounts.

Then we can keep going until we reach the n. And finally just sum up these two variables as result.

class Solution {

public int numWays(int n, int k) {

if (n == 0){

return 0;

}

if (n == 1){

return k;

}

int diffColorCount = k\*(k-1);

int sameColorCount = k;

for (int i = 2; i < n; i++){

int temp = diffColorCount;

diffColorCount = (diffColorCount + sameColorCount) \* (k-1);

sameColorCount = temp;

}

return diffColorCount + sameColorCount;

}

}

# 279\_PerfectSquares.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n.

For example, given n = 12, return 3 because 12 = 4 + 4 + 4; given n = 13, return 2 because 13 = 4 + 9.

Method: dynamic programming

class Solution {

public int numSquares(int n) {

int[] dp = new int[n+1];

Arrays.fill(dp, Integer.MAX\_VALUE);

dp[0] = 0;

for (int i = 1; i <= n; i++){

for (int j = 0; j \* j <= i; j++){

if (dp[i - j \* j] != Integer.MAX\_VALUE){

dp[i] = Math.min(dp[i], dp[i - j \* j] + 1);

}

}

}

return dp[n];

}

}

class Solution {

public int numSquares(int n) {

int[] dp = new int[n+1];

Arrays.fill(dp, Integer.MAX\_VALUE);

dp[0] = 0;

for (int i = 1; i <= n; i++){

for (int j = 1; j \* j <= i; j++){

if (dp[i-j\*j] != Integer.MAX\_VALUE){

dp[i] = Math.min(dp[i], dp[i-j\*j] + 1);

}

}

}

return dp[n];

}

}

# 28\_ImplementstrStr.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement strStr().

Return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Example 1:

Input: haystack = "hello", needle = "ll"

Output: 2

Example 2:

Input: haystack = "aaaaa", needle = "bba"

Output: -1

class Solution {

public int strStr(String haystack, String needle) {

int j;

for (int i = 0; i < haystack.length() - needle.length() + 1; i++){

for (j = 0; j < needle.length(); j++){

if (haystack.charAt(i+j) != needle.charAt(j)){

break;

}

}

if (j == needle.length()){

return i;

}

}

return -1;

}

}

# 280\_WiggleSort.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted array nums, reorder it in-place such that nums[0] <= nums[1] >= nums[2] <= nums[3]....

For example, given nums = [3, 5, 2, 1, 6, 4], one possible answer is [1, 6, 2, 5, 3, 4].

Method 1:

Time complexity: O(nlogn)

Space complexity : O(1)O(1)O(1). Space depends on the sorting implementation which, usually,

costs O(1)O(1)O(1) auxiliary space if heapsort is used.

class Solution {

public void wiggleSort(int[] nums) {

if (nums == null || nums.length == 0){

return;

}

Arrays.sort(nums);

for (int i = 1; i < nums.length - 1; i += 2){

swap(nums, i, i+1);

}

}

private void swap(int[] nums, int i, int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

Method 2:

As we iterate through the array, we compare the current element to its next element

and if the order is incorrect, we swap them.

Time complexity : O(n)O(n)O(n). In the worst case we swap at most n/2 times. An example input is [2,1,3,1,4,1].

Space complexity : O(1)

class Solution {

public void wiggleSort(int[] nums) {

if (nums == null || nums.length == 0){

return;

}

boolean less = true;

for (int i = 0; i < nums.length - 1; i++){

if (less){

if (nums[i] > nums[i+1]){

swap(nums, i, i+1);

}

}else{

if (nums[i] < nums[i+1]){

swap(nums, i, i+1);

}

}

less = !less;

}

}

private void swap(int[] nums, int i, int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

## 324\_WiggleSortII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted array nums, reorder it such that nums[0] < nums[1] > nums[2] < nums[3]....

Example:

(1) Given nums = [1, 5, 1, 1, 6, 4], one possible answer is [1, 4, 1, 5, 1, 6].

(2) Given nums = [1, 3, 2, 2, 3, 1], one possible answer is [2, 3, 1, 3, 1, 2].

Note:

You may assume all input has valid answer.

Follow Up:

Can you do it in O(n) time and/or in-place with O(1) extra space?

Method 1:

O(nlogn) with extra space

class Solution {

public void wiggleSort(int[] nums) {

if (nums == null || nums.length == 0){

return;

}

Arrays.sort(nums);

int n = nums.length;

int m = (n + 1) / 2;

int[] temp = new int[n];

for (int i = 0; i < nums.length; i++){

if (i % 2 == 0){

temp[i] = nums[--m];

}else{

temp[i] = nums[--n];

}

}

for (int i = 0; i < nums.length; i++){

nums[i] = temp[i];

}

}

}

# 281\_Zigzag.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two 1d vectors, implement an iterator to return their elements alternately.

Example:

Input:

v1 = [1,2]

v2 = [3,4,5,6]

Output: [1,3,2,4,5,6]

Explanation: By calling next repeatedly until hasNext returns false,

the order of elements returned by next should be: [1,3,2,4,5,6].

Follow up: What if you are given k 1d vectors? How well can your code be extended to such cases?

Clarification for the follow up question:

The "Zigzag" order is not clearly defined and is ambiguous for k > 2 cases. If "Zigzag" does not look right to you,

replace "Zigzag" with "Cyclic". For example:

Input:

[1,2,3]

[4,5,6,7]

[8,9]

Output: [1,4,8,2,5,9,3,6,7].

Method 1: can be extented to k vector

public class ZigzagIterator {

Queue<Iterator> queue;

public ZigzagIterator(List<Integer> v1, List<Integer> v2) {

queue = new LinkedList<Iterator>();

if (!v1.isEmpty()){

queue.offer(v1.iterator());

}

if (!v2.isEmpty()){

queue.offer(v2.iterator());

}

}

public int next() {

Iterator ite = queue.poll();

int next = (int) ite.next();

if (ite.hasNext()){

queue.offer(ite);

}

return next;

}

public boolean hasNext() {

return !queue.isEmpty();

}

}

https://www.geeksforgeeks.org/how-to-use-iterator-in-java/

Iterator iterator = list.iterator();

while (iterator.hasNext())

System.out.print(iterator.next()+ " ");

/\*\*

\* Your ZigzagIterator object will be instantiated and called as such:

\* ZigzagIterator i = new ZigzagIterator(v1, v2);

\* while (i.hasNext()) v[f()] = i.next();

\*/

Method 2:

public class ZigzagIterator {

private int next;

private int p1;

private int p2;

private boolean p1Curr;

private List<Integer> l1;

private List<Integer> l2;

public ZigzagIterator(List<Integer> v1, List<Integer> v2) {

l1 = new ArrayList<>();

l2 = new ArrayList<>();

p1 = 0;

p2 = 0;

next = 0;

p1Curr = true;

if (v1.isEmpty()){

p1Curr = false;

}

for (int i = 0; i < v1.size(); i++){

l1.add(v1.get(i));

}

for (int i = 0; i < v2.size(); i++){

l2.add(v2.get(i));

}

}

public int next() {

return next;

}

public boolean hasNext() {

if (p1 + p2 < l1.size() + l2.size()){

if (p1 < l1.size() && p1Curr){

next = l1.get(p1);

p1++;

if (p2 < l2.size()){

p1Curr = false;

}

}else if (p2 < l2.size() && !p1Curr){

next = l2.get(p2);

p2++;

if (p1 < l1.size()){

p1Curr = true;

}

}

return true;

}

return false;

}

}

/\*\*

\* Your ZigzagIterator object will be instantiated and called as such:

\* ZigzagIterator i = new ZigzagIterator(v1, v2);

\* while (i.hasNext()) v[f()] = i.next();

\*/

# 282\_Expression.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string that contains only digits 0-9 and a target value, return all possibilities to

add binary operators (not unary) +, -, or \* between the digits so they evaluate to the target value.

Examples:

"123", 6 -> ["1+2+3", "1\*2\*3"]

"232", 8 -> ["2\*3+2", "2+3\*2"]

"105", 5 -> ["1\*0+5","10-5"]

"00", 0 -> ["0+0", "0-0", "0\*0"]

"3456237490", 9191 -> []

Best solution:

class Solution {

public List<String> addOperators(String num, int target) {

List<String> res = new ArrayList<>();

dfs(res, num, target, "", 0, 0, 0);

return res;

}

private void dfs(List<String> res, String num, int target, String item, int start, long eval, long prev){

if(start == num.length()){

if (eval + prev == target){

res.add(item);

}

return;

}

for (int i = start; i < num.length(); i++){

String sub = num.substring(start, i+1);

if (sub.length() > 1 && sub.charAt(0) == '0'){

break;

}

long curr = Long.parseLong(sub);

if (item.length() == 0){

dfs(res, num, target, sub, i + 1, eval, curr);

}else{

dfs(res, num, target, item + "+" + sub, i+1, eval + prev, curr);

dfs(res, num, target, item + "-" + sub, i+1, eval + prev, -curr);

dfs(res, num, target, item + "\*" + sub, i + 1, eval, prev \* curr);

}

}

}

}

class Solution {

public List<String> addOperators(String num, int target) {

List<String> result = new ArrayList<>();

if (num == null || num.length() == 0){

return result;

}

dfs(result, "", num, target, 0, 0, 0);

return result;

}

private void dfs(List<String> result, String item, String num, int target, int start, long eval, long prev){

if (start == num.length()){

if (eval == target){

result.add(item);

}

return;

}

for (int i = start; i < num.length(); i++){

if (i != start && num.charAt(start) == '0'){ //case: starts with 0. e.g. '012'

break;

}

Long curr = Long.parseLong(num.substring(start, i+1));

if (start == 0){

dfs(result, item + curr, num, target, i + 1, curr, curr);

}else{

dfs(result, item + "+" + curr, num, target, i + 1, eval + curr, curr);

dfs(result, item + "-" + curr, num, target, i + 1, eval - curr, -curr);

dfs(result, item + "\*" + curr, num, target, i + 1, eval - prev + prev \* curr, prev \* curr);

}

}

}

}

Backtracking using stringbuilder

https://leetcode.com/problems/expression-add-operators/discuss/71895/Java-Standard-Backtrace-AC-Solutoin-short-and-clear

use setLength to do backtracking

public List<String> addOperators(String num, int target) {

List<String> res = new ArrayList<>();

StringBuilder sb = new StringBuilder();

dfs(res, sb, num, 0, target, 0, 0);

return res;

}

public void dfs(List<String> res, StringBuilder sb, String num, int pos, int target, long prev, long multi) {

if(pos == num.length()) {

if(target == prev) res.add(sb.toString());

return;

}

for(int i = pos; i < num.length(); i++) {

if(num.charAt(pos) == '0' && i != pos) break;

long curr = Long.parseLong(num.substring(pos, i + 1));

int len = sb.length();

if(pos == 0) {

dfs(res, sb.append(curr), num, i + 1, target, curr, curr);

sb.setLength(len);

} else {

dfs(res, sb.append("+").append(curr), num, i + 1, target, prev + curr, curr);

sb.setLength(len);

dfs(res, sb.append("-").append(curr), num, i + 1, target, prev - curr, -curr);

sb.setLength(len);

dfs(res, sb.append("\*").append(curr), num, i + 1, target, prev - multi + multi \* curr, multi \* curr);

sb.setLength(len);

}

}

}

Best solution:

class Solution {

public List<String> addOperators(String num, int target) {

List<String> res = new ArrayList<>();

dfs(res, num, target, "", 0, 0, 0);

return res;

}

private void dfs(List<String> res, String num, int target, String item, int start, long eval, long prev){

if (start == num.length()){

if (eval + prev == target){

res.add(item);

}

return;

}

for (int i = start; i < num.length(); i++){

String curr = num.substring(start, i+1);

if (curr.length() > 1 && curr.charAt(0) == '0'){

break;

}

if (start == 0){

dfs(res, num, target, item + curr, i+1, 0, Long.parseLong(curr));

}else{

dfs(res, num, target, item + "+" + curr, i+1, eval + prev, Long.parseLong(curr));

dfs(res, num, target, item + "-" + curr, i+1, eval + prev, -Long.parseLong(curr));

dfs(res, num, target, item + "\*" + curr, i+1, eval, prev \* Long.parseLong(curr));

}

}

}

}

# 283\_MoveZeroes.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array nums, write a function to move all 0's to the end of it while maintaining the

relative order of the non-zero elements.

For example, given nums = [0, 1, 0, 3, 12], after calling your function, nums should be [1, 3, 12, 0, 0].

Note:

You must do this in-place without making a copy of the array.

Minimize the total number of operations.

class Solution {

public void moveZeroes(int[] nums) {

int index = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] != 0){

int temp = nums[i];

nums[i] = nums[index];

nums[index] = temp;

index++;

}

}

}

}

class Solution {

public void moveZeroes(int[] nums) {

int index = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] != 0){

nums[index++] = nums[i];

}

}

for (int i = index; i < nums.length; i++){

nums[i] = 0;

}

}

}

# 284\_Peeking.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an Iterator class interface with methods: next() and hasNext(), design and implement a PeekingIterator that support the peek() operation -- it essentially peek() at the element that will be returned by the next call to next().

Here is an example. Assume that the iterator is initialized to the beginning of the list: [1, 2, 3].

Call next() gets you 1, the first element in the list.

Now you call peek() and it returns 2, the next element. Calling next() after that still return 2.

You call next() the final time and it returns 3, the last element. Calling hasNext() after that should return false.

Follow up: How would you extend your design to be generic and work with all types, not just integer?

// Java Iterator interface reference:

// https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

class PeekingIterator implements Iterator<Integer> {

private Integer next;

private Iterator<Integer> iter;

public PeekingIterator(Iterator<Integer> iterator) {

iter = iterator;

if (iter.hasNext()){

next = iter.next();

}

}

// Returns the next element in the iteration without advancing the iterator.

public Integer peek() {

return next;

}

// hasNext() and next() should behave the same as in the Iterator interface.

// Override them if needed.

@Override

public Integer next() {

Integer res = next;

if (iter.hasNext()){

next = iter.next();

}else{

next = null;

}

return res;

}

@Override

public boolean hasNext() {

return next != null;

}

}

# 286\_Walls.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a m x n 2D grid initialized with these three possible values.

-1 - A wall or an obstacle.

0 - A gate.

INF - Infinity means an empty room. We use the value 231 - 1 = 2147483647 to represent INF as you may assume that the distance

to a gate is less than 2147483647.

Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF.

Example:

Given the 2D grid:

INF -1 0 INF

INF INF INF -1

INF -1 INF -1

0 -1 INF INF

After running your function, the 2D grid should be:

3 -1 0 1

2 2 1 -1

1 -1 2 -1

0 -1 3 4

Method: BFS

Time complexity: O(mn)

class Solution {

public void wallsAndGates(int[][] rooms) {

if (rooms == null || rooms.length == 0){

return;

}

Queue<int[]> queue = new LinkedList<>();

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

int m = rooms.length;

int n = rooms[0].length;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (rooms[i][j] == 0){

queue.offer(new int[]{i, j});

}

}

}

int level = 0;

while (!queue.isEmpty()){

level++;

int size = queue.size();

for (int i = 0; i < size; i++){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx < m && nx >= 0 && ny < n && ny >= 0 && rooms[nx][ny] != -1 && rooms[nx][ny] != 0){

if (level < rooms[nx][ny]){

rooms[nx][ny] = level;

queue.offer(new int[]{nx, ny});

}

}

}

}

}

}

}

# 287\_FindDuplicateNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array nums containing n + 1 integers where each integer is between 1 and n (inclusive),

prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Note:

You must not modify the array (assume the array is read only).

You must use only constant, O(1) extra space.

Your runtime complexity should be less than O(n2).

There is only one duplicate number in the array, but it could be repeated more than once.

Method 1:

Time complexity: O(nlogn)

class Solution {

public int findDuplicate(int[] nums) {

Arrays.sort(nums);

for (int i = 1; i < nums.length; i++){

if (nums[i-1] == nums[i]){

return nums[i];

}

}

return 0;

}

}

Method 2:

Time complexity: O(nlogn)

Find the biggest number that check\_smaller\_number <= itself

https://leetcode.com/problems/find-the-duplicate-number/discuss/72844/Two-Solutions-(with-explanation):-O(nlog(n))-and-O(n)-time-O(1)-space-without-changing-the-input-array

class Solution {

public int findDuplicate(int[] nums) {

int start = 1;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (check\_smaller\_number(mid, nums) <= mid){

start = mid;

}else{

end = mid;

}

}

if (check\_smaller\_number(start, nums) <= start){

return end;

}

return start;

}

private int check\_smaller\_number(int mid, int[] nums){

int count = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] <= mid){

count++;

}

}

return count;

}

}

class Solution {

public int findDuplicate(int[] nums) {

int start = 1;

int end = nums.length - 1;

while (start <= end){

int mid = start + (end - start) / 2;

if (check\_smaller\_number(mid, nums) <= mid){

start = mid + 1;

}else{

end = mid - 1;

}

}

return start;

}

private int check\_smaller\_number(int mid, int[] nums){

int count = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] <= mid){

count++;

}

}

return count;

}

}

Method 3:

https://leetcode.com/problems/find-the-duplicate-number/discuss/72846/My-easy-understood-solution-with-O(n)-time-and-O(1)-space-without-modifying-the-array.-With-clear-explanation.

class Solution {

public int findDuplicate(int[] nums) {

int slow = nums[0];

int fast = nums[nums[0]];

while (fast != slow){

slow = nums[slow];

fast = nums[nums[fast]];

}

int entry = 0;//can't use nums[0]

while (entry != slow){

entry = nums[entry];

slow = nums[slow];

}

return slow;

}

}

# 288\_Unique.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

An abbreviation of a word follows the form <first letter><number><last letter>. Below are some examples of word abbreviations:

a) it --> it (no abbreviation)

1

↓

b) d|o|g --> d1g

1 1 1

1---5----0----5--8

↓ ↓ ↓ ↓ ↓

c) i|nternationalizatio|n --> i18n

1

1---5----0

↓ ↓ ↓

d) l|ocalizatio|n --> l10n

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation

is unique if no other word from the dictionary has the same abbreviation.

Example:

Given dictionary = [ "deer", "door", "cake", "card" ]

isUnique("dear") -> false

isUnique("cart") -> true

isUnique("cane") -> false

isUnique("make") -> true

Method:

class ValidWordAbbr {

Map<String, String> map;

public ValidWordAbbr(String[] dictionary) {

map = new HashMap<>();

for (String s : dictionary){

String key = getAbbr(s);

if (map.containsKey(key)){

if (!map.get(key).equals(s)){

map.put(key, "");

}

}else{

map.put(key, s);

}

}

}

public boolean isUnique(String word) {

return !map.containsKey(getAbbr(word)) || map.get(getAbbr(word)).equals(word);

}

private String getAbbr(String word){

int n = word.length();

if (n <= 2){

return word;

}

return word.charAt(0) + String.valueOf(n - 2) + word.charAt(n - 1);

}

}

/\*\*

\* Your ValidWordAbbr object will be instantiated and called as such:

\* ValidWordAbbr obj = new ValidWordAbbr(dictionary);

\* boolean param\_1 = obj.isUnique(word);

\*/

# 289\_GameofLife.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

According to the Wikipedia's article: "The Game of Life, also known simply as Life, i

s a cellular automaton devised by the British mathematician John Horton Conway in 1970."

Given a board with m by n cells, each cell has an initial state live (1) or dead (0).

Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following

four rules (taken from the above Wikipedia article):

Any live cell with fewer than two live neighbors dies, as if caused by under-population.

Any live cell with two or three live neighbors lives on to the next generation.

Any live cell with more than three live neighbors dies, as if by over-population..

Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

Follow up:

Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot

update some cells first and then use their updated values to update other cells.

In this question, we represent the board using a 2D array. In principle, the board is infinite,

which would cause problems when the active area encroaches the border of the array. How would you address these problems?

Method 1:

class Solution {

public void gameOfLife(int[][] board) {

if (board == null || board.length == 0 || board[0].length == 0){

return;

}

int m = board.length;

int n = board[0].length;

int[][] temp = new int[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

int count = countLive(board, i, j);

if (board[i][j] == 1){

if (count < 2 || count > 3){

temp[i][j] = 0;

}else if (count == 2 || count == 3){

temp[i][j] = 1;

}

}else{

if (count == 3){

temp[i][j] = 1;

}

}

}

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

board[i][j] = temp[i][j];

}

}

}

private int countLive(int[][] board, int i, int j){

int m = board.length;

int n = board[0].length;

int[] dx = {1, 0, -1, -1, -1, 0, 1, 1};

int[] dy = {1, 1, 1, 0, -1, -1, -1, 0};

int count = 0;

for (int k = 0; k < dx.length; k++){

int cx = i + dx[k];

int cy = j + dy[k];

if (cx >= 0 && cx < m && cy >= 0 && cy < n && board[cx][cy] == 1){

count++;

}

}

return count;

}

}

Method 2:

https://leetcode.com/problems/game-of-life/discuss/73223/Easiest-JAVA-solution-with-explanation

[2nd bit, 1st bit] = [next state, current state]

- 00 dead (next) <- dead (current)

- 01 dead (next) <- live (current)

- 10 live (next) <- dead (current)

- 11 live (next) <- live (current)

To get the current state, simply do

board[i][j] & 1

To get the next state, simply do

board[i][j] >> 1

public void gameOfLife(int[][] board) {

if (board == null || board.length == 0) return;

int m = board.length, n = board[0].length;

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

int lives = liveNeighbors(board, m, n, i, j);

// In the beginning, every 2nd bit is 0;

// So we only need to care about when will the 2nd bit become 1.

if (board[i][j] == 1 && lives >= 2 && lives <= 3) {

board[i][j] = 3; // Make the 2nd bit 1: 01 ---> 11

}

if (board[i][j] == 0 && lives == 3) {

board[i][j] = 2; // Make the 2nd bit 1: 00 ---> 10

}

}

}

for (int i = 0; i < m; i++) {

for (int j = 0; j < n; j++) {

board[i][j] >>= 1; // Get the 2nd state.

}

}

}

public int liveNeighbors(int[][] board, int m, int n, int i, int j) {

int lives = 0;

for (int x = Math.max(i - 1, 0); x <= Math.min(i + 1, m - 1); x++) {

for (int y = Math.max(j - 1, 0); y <= Math.min(j + 1, n - 1); y++) {

lives += board[x][y] & 1;

}

}

lives -= board[i][j] & 1;

return lives;

}

# 29\_Divide.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two integers dividend and divisor, divide two integers without using multiplication, division and mod operator.

Return the quotient after dividing dividend by divisor.

The integer division should truncate toward zero.

Example 1:

Input: dividend = 10, divisor = 3

Output: 3

Example 2:

Input: dividend = 7, divisor = -3

Output: -2

Note:

Both dividend and divisor will be 32-bit signed integers.

The divisor will never be 0.

Assume we are dealing with an environment which could only store integers within the 32-bit signed integer range: [−231, 231 − 1]. For the purpose of this problem, assume that your function returns 231 − 1 when the division result overflows.

Method 1: O(N) TLE

class Solution {

public int divide(int dividend, int divisor) {

int sign = 1;

if (dividend < 0 && divisor > 0 || dividend > 0 && divisor < 0){

sign = -1;

}

long dLong = Math.abs((long)dividend);

long sLong = Math.abs((long)divisor);

long res = div(dLong, sLong);

if (res > Integer.MAX\_VALUE){

if (sign == 1){

return Integer.MAX\_VALUE;

}else{

return Integer.MIN\_VALUE;

}

}

if (sign == 1){

return (int)res;

}

return -(int)res;

}

private long div(long dividend, long divisor){

int multi = 0;

int sum = 0;

while (sum <= dividend){

sum += divisor;

multi++;

}

if (multi == 0){

return 0;

}

return multi-1;

}

}

Method 2: O(log(N)

class Solution {

public int divide(int dividend, int divisor) {

int sign = 1;

if (dividend < 0 && divisor > 0 || dividend > 0 && divisor < 0){

sign = -1;

}

long dLong = Math.abs((long)dividend);

long sLong = Math.abs((long)divisor);

long res = div(dLong, sLong);

if (res > Integer.MAX\_VALUE){

if (sign == 1){

return Integer.MAX\_VALUE;

}else{

return Integer.MIN\_VALUE;

}

}

if (sign == 1){

return (int)res;

}

return -(int)res;

}

private long div(long dividend, long divisor){

if (dividend < divisor){

return 0;

}

long multi = 1;

long sum = divisor;

while (sum + sum <= dividend){

sum += sum;

multi += multi;

}

return multi + div(dividend - sum, divisor);

}

}

# 290\_WordPattern.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a pattern and a string str, find if str follows the same pattern.

Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty word in str.

Examples:

pattern = "abba", str = "dog cat cat dog" should return true.

pattern = "abba", str = "dog cat cat fish" should return false.

pattern = "aaaa", str = "dog cat cat dog" should return false.

pattern = "abba", str = "dog dog dog dog" should return false.

Notes:

You may assume pattern contains only lowercase letters, and str contains lowercase letters separated by a single space.

class Solution {

public boolean wordPattern(String pattern, String str) {

Map<Character, String> map = new HashMap<>();

String[] strArray = str.split(" ");

//String[] strs = str.split("\\s+");

if (pattern.length() != strArray.length){

return false;

}

for (int i = 0; i < pattern.length(); i++){

char c = pattern.charAt(i);

if (!map.containsKey(c)){

map.put(c, strArray[i]);

}else{

if (!map.get(c).equals(strArray[i])){

return false;

}

}

}

Map<String, Character> map1 = new HashMap<>();

for (int i = 0; i < strArray.length; i++){

char c = pattern.charAt(i);

if (!map1.containsKey(strArray[i])){

map1.put(strArray[i], c);

}else{

if (map1.get(strArray[i]) != c){

return false;

}

}

}

return true;

}

}

https://leetcode.com/problems/word-pattern/discuss/73402/8-lines-simple-Java

previousValue =map.put(key, value) return the stored previous value

class Solution {

public boolean wordPattern(String pattern, String str) {

Map map = new HashMap();

String[] strArray = str.split(" ");

if (pattern.length() != strArray.length){

return false;

}

for (Integer i = 0; i < pattern.length(); i++){ //notice here we must use Integer instead of int

// i being an Integer object, which allows to compare with just != because

// there’s no autoboxing-same-value-to-different-objects-problem anymore.

char c = pattern.charAt(i);

if (map.put(c, i) != map.put(strArray[i], i)){

return false;

}

}

return true;

}

}

# 291\_Word.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a pattern and a string str, find if str follows the same pattern.

Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty substring in str.

Example 1:

Input: pattern = "abab", str = "redblueredblue"

Output: true

Example 2:

Input: pattern = pattern = "aaaa", str = "asdasdasdasd"

Output: true

Example 3:

Input: pattern = "aabb", str = "xyzabcxzyabc"

Output: false

Notes:

You may assume both pattern and str contains only lowercase letters.

Backtracking

class Solution {

public boolean wordPatternMatch(String pattern, String str) {

Map<Character, String> map = new HashMap<>();

Set<String> set = new HashSet<>();

return isMatch(pattern, 0, str, 0, map, set);

}

private boolean isMatch(String pattern, int pStart, String source, int sStart, Map<Character, String> map, Set<String> set){

if (pStart == pattern.length() && sStart == source.length()){

return true;

}

if (pStart == pattern.length() || sStart == source.length()){

return false;

}

char c = pattern.charAt(pStart);

// if the pattern character exists

if (map.containsKey(c)){

String s = map.get(c);

// then check if we can use it to match str[i...i+s.length()]

if (!source.startsWith(s, sStart)){

return false;

}

return isMatch(pattern, pStart+1, source, sStart+s.length(), map, set);

}

// pattern character does not exist in the map

for (int i = sStart; i < source.length(); i++){

String str = source.substring(sStart, i + 1);

//use a hash set to avoid duplicate matches

if (set.contains(str)){

continue;

}

//backtracking

map.put(c, str);

set.add(str);

if (isMatch(pattern, pStart+1, source, sStart+str.length(), map, set)){

return true;

}

map.remove(c);

set.remove(str);

}

//try every possbile case, still not match

return false;

}

}

Reference:

We can solve this problem using backtracking, we just have to keep trying to use a character in the pattern to match different

length of substrings in the input string, keep trying till we go through the input string and the pattern.

For example, input string is "redblueredblue", and the pattern is "abab", first let's use 'a' to match "r", 'b' to match "e",

then we find that 'a' does not match "d", so we do backtracking, use 'b' to match "ed", so on and so forth ...

When we do the recursion, if the pattern character exists in the hash map already, we just have to see if we can use it to

match the same length of the string. For example, let's say we have the following map:

'a': "red"

'b': "blue"

now when we see 'a' again, we know that it should match "red", the length is 3, then let's see if str[i ... i+3] matches 'a',

where i is the current index of the input string. Thanks to StefanPochmann's suggestion, in Java we can elegantly

use str.startsWith(s, i) to do the check.

Also thanks to i-tikhonov's suggestion, we can use a hash set to avoid duplicate matches, if character a matches string "red",

then character b cannot be used to match "red". In my opinion though, we can say apple (pattern 'a') is "fruit", orange (pattern 'o')

is "fruit", so they can match the same string, anyhow, I guess it really depends on how the problem states.

The following code should pass OJ now, if we don't need to worry about the duplicate matches, just remove the code that associates

with the hash set.

public class Solution {

public boolean wordPatternMatch(String pattern, String str) {

Map<Character, String> map = new HashMap<>();

Set<String> set = new HashSet<>();

return isMatch(str, 0, pattern, 0, map, set);

}

boolean isMatch(String str, int i, String pat, int j, Map<Character, String> map, Set<String> set) {

// base case

if (i == str.length() && j == pat.length()) return true;

if (i == str.length() || j == pat.length()) return false;

// get current pattern character

char c = pat.charAt(j);

// if the pattern character exists

if (map.containsKey(c)) {

String s = map.get(c);

// then check if we can use it to match str[i...i+s.length()]

if (!str.startsWith(s, i)) {

return false;

}

// if it can match, great, continue to match the rest

return isMatch(str, i + s.length(), pat, j + 1, map, set);

}

// pattern character does not exist in the map

for (int k = i; k < str.length(); k++) {

String p = str.substring(i, k + 1);

if (set.contains(p)) {

continue;

}

// create or update it

map.put(c, p);

set.add(p);

// continue to match the rest

if (isMatch(str, k + 1, pat, j + 1, map, set)) {

return true;

}

// backtracking

map.remove(c);

set.remove(p);

}

// we've tried our best but still no luck

return false;

}

}

# 292\_Nim.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of

you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the

first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you

can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you

remove, the last stone will always be removed by your friend.

class Solution {

public boolean canWinNim(int n) {

return (n%4 != 0);

}

}

# 293\_Flip.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -,

you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move

and therefore the other person will be the winner.

Write a function to compute all possible states of the string after one valid move.

Example:

Input: s = "++++"

Output:

[

"--++",

"+--+",

"++--"

]

class Solution {

public List<String> generatePossibleNextMoves(String s) {

List<String> result = new ArrayList<>();

for (int i = 0; i < s.length() - 1; i++){

StringBuilder sb = new StringBuilder();

if (s.charAt(i) == '+' && s.charAt(i+1) == '+'){

sb.append(s.substring(0, i));

sb.append("--");

sb.append(s.substring(i+2));

result.add(sb.toString());

}

}

return result;

}

}

# 294\_Flip.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -,

you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move

and therefore the other person will be the winner.

Write a function to determine if the starting player can guarantee a win.

Example:

Input: s = "++++"

Output: true

Explanation: The starting player can guarantee a win by flipping the middle "++" to become "+--+".

Follow up:

Derive your algorithm's runtime complexity.

Method 1: Inspired by Flip Game I

Time complexity: O(N^N)

T(N) = (N-2) \* T(N-2) = (N-2) \* (N-4) \* T(N-4) ... = (N-2) \* (N-4) \* (N-6) \* ... ~ O(N!!)

At first glance, backtracking seems to be the only feasible solution to this problem. We can basically try every possible

move for the first player (Let's call him 1P from now on), and recursively check if the second player 2P has any chance to win.

If 2P is guaranteed to lose, then we know the current move 1P takes must be the winning move. The naive implementation is actually

very simple:

Time: 349 ms

class Solution {

public boolean canWin(String s) {

List<String> result = new ArrayList<>();

// generate all possible sequence after every attempt, inspired by Flip game I

for (int i = 0; i < s.length() - 1; i++){

StringBuilder sb = new StringBuilder();

if (s.charAt(i) == '+' && s.charAt(i+1) == '+'){

sb.append(s.substring(0, i));

sb.append("--");

sb.append(s.substring(i+2));

result.add(sb.toString());

}

}

for (String str : result){

if (!canWin(str)){ // if there is any one way the next player can't win, take it and you'll win

return true;

}

}

return false;

}

}

Method 2: with memorization

time: 34ms

class Solution {

Map<String, Boolean> map = new HashMap<>();

public boolean canWin(String s) {

List<String> result = new ArrayList<>();

// generate all possible sequence after every attempt, inspired by Flip game I

for (int i = 0; i < s.length() - 1; i++){

StringBuilder sb = new StringBuilder();

if (s.charAt(i) == '+' && s.charAt(i+1) == '+'){

sb.append(s.substring(0, i));

sb.append("--");

sb.append(s.substring(i+2));

String temp = sb.toString();

result.add(temp);

if (map.containsKey(temp)){

return true;

}

}

}

for (String str : result){

if (!canWin(str)){ // if there is any one way the next player can't win, take it and you'll win

map.put(str, true);

return true;

}

}

return false;

}

}

# 295\_FindMedianDataStream.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:

[2,3,4] , the median is 3

[2,3], the median is (2 + 3) / 2 = 2.5

Design a data structure that supports the following two operations:

void addNum(int num) - Add a integer number from the data stream to the data structure.

double findMedian() - Return the median of all elements so far.

For example:

addNum(1)

addNum(2)

findMedian() -> 1.5

addNum(3)

findMedian() -> 2

Similar to Leetcode 480

https://github.com/optimisea/Leetcode/blob/master/Java/480\_Sliding.java

class MedianFinder {

private Queue<Integer> minHeap; // keep the larger half

private Queue<Integer> maxHeap; // keep the smaller half

/\*\* initialize your data structure here. \*/

public MedianFinder() {

minHeap = new PriorityQueue<Integer>();

maxHeap = new PriorityQueue<Integer>(new Comparator<Integer>(){

public int compare(Integer a, Integer b){

return b.compareTo(a);

}

});

}

public void addNum(int num) {

if (minHeap.size() == 0 && maxHeap.size() == 0){

minHeap.add(num);

}else if (minHeap.size() > maxHeap.size()){

if (num > minHeap.peek()){

maxHeap.offer(minHeap.poll());

minHeap.offer(num);

}else{

maxHeap.offer(num);

}

}else if (minHeap.size() < maxHeap.size()){

if (num < maxHeap.peek()){

minHeap.offer(maxHeap.poll());

maxHeap.offer(num);

}else{

minHeap.offer(num);

}

}else{

if (num > minHeap.peek()){

minHeap.offer(num);

}else{

maxHeap.offer(num);

}

}

}

public double findMedian() {

if (minHeap.size() == 0 && maxHeap.size() == 0){

return 0.0;

}

if (minHeap.size() > maxHeap.size()){

return (double) minHeap.peek();

}

if (maxHeap.size() > minHeap.size()){

return (double) maxHeap.peek();

}

return (minHeap.peek() + maxHeap.peek()) / 2.0;

}

}

/\*\*

\* Your MedianFinder object will be instantiated and called as such:

\* MedianFinder obj = new MedianFinder();

\* obj.addNum(num);

\* double param\_2 = obj.findMedian();

\*/

Better version:

class MedianFinder {

Queue<Integer> maxHeap;

Queue<Integer> minHeap;

/\*\* initialize your data structure here. \*/

public MedianFinder() {

minHeap = new PriorityQueue<>();

maxHeap = new PriorityQueue<>(new Comparator<Integer>(){

public int compare (Integer i1, Integer i2){

return (int)(i2 - i1);

}

});

}

public void addNum(int num) {

if (minHeap.isEmpty()){

minHeap.offer(num);

}else{

if (num < minHeap.peek()){

maxHeap.offer(num);

if (maxHeap.size() > minHeap.size()){

minHeap.offer(maxHeap.poll());

}

}else{

minHeap.offer(num);

if (minHeap.size() > maxHeap.size() + 1){

maxHeap.offer(minHeap.poll());

}

}

}

}

public double findMedian() {

if (minHeap.size() == maxHeap.size()){

return (minHeap.peek() + maxHeap.peek()) / 2.0;

}

return (double)minHeap.peek();

}

}

Better version:

class MedianFinder {

Queue<Integer> maxHeap;

Queue<Integer> minHeap;

/\*\* initialize your data structure here. \*/

public MedianFinder() {

minHeap = new PriorityQueue<>();

maxHeap = new PriorityQueue<>(new Comparator<Integer>(){

public int compare (Integer i1, Integer i2){

return (int)(i2 - i1);

}

});

}

public void addNum(int num) {

if (minHeap.isEmpty()){

minHeap.offer(num);

return;

}

if (num < minHeap.peek()){

maxHeap.offer(num);

}else{

minHeap.offer(num);

}

if (maxHeap.size() > minHeap.size()){

minHeap.offer(maxHeap.poll());

}else if (minHeap.size() > maxHeap.size() + 1){

maxHeap.offer(minHeap.poll());

}

}

public double findMedian() {

if (minHeap.size() == maxHeap.size()){

return (minHeap.peek() + maxHeap.peek()) / 2.0;

}

return (double)minHeap.peek();

}

}

/\*\*

\* Your MedianFinder object will be instantiated and called as such:

\* MedianFinder obj = new MedianFinder();

\* obj.addNum(num);

\* double param\_2 = obj.findMedian();

\*/

# 296\_Best.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A group of two or more people wants to meet and minimize the total travel distance.

You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group.

The distance is calculated using Manhattan Distance, where distance(p1, p2) = |p2.x - p1.x| + |p2.y - p1.y|.

Example:

Input:

1 - 0 - 0 - 0 - 1

| | | | |

0 - 0 - 0 - 0 - 0

| | | | |

0 - 0 - 1 - 0 - 0

Output: 6

Explanation: Given three people living at (0,0), (0,4), and (2,2):

The point (0,2) is an ideal meeting point, as the total travel distance

of 2+2+2=6 is minimal. So return 6.

Method 1:

Time complexity: O(mnk) k is the number of 1 in grid

Space complexity: O(k)

class Solution {

public int minTotalDistance(int[][] grid) {

List<int[]> list = new ArrayList<>();

for (int i = 0; i < grid.length; i++){

for (int j = 0; j < grid[0].length; j++){

if (grid[i][j] == 1){

int[] temp = new int[2];

temp[0] = i;

temp[1] = j;

list.add(temp);

}

}

}

int min = Integer.MAX\_VALUE;

for (int i = 0; i < grid.length; i++){

for (int j = 0; j < grid[0].length; j++){

int d = dist(i, j, list);

min = Math.min(min, d);

}

}

return min;

}

private int dist(int i, int j, List<int[]> list){

int sum = 0;

for (int[] point : list){

int x = point[0];

int y = point[1];

sum += Math.abs(x - i) + Math.abs(y - j);

}

return sum;

}

}

Method 2: Sort

Note that the best meeting point must locate somewhere between the left-most and right-most point.

he median must be the optimal meeting point, not mean

Case #4: 1-1-0-0-1

To see why this is so, let us look at case #4 above and choose the median x=1x = 1x=1 as our initial meeting point.

Assume that the total distance traveled is d. Note that we have equal number of points distributed to its left and to its right.

Now let us move one step to its right where x=2x = 2x=2 and notice how the distance changes accordingly.

Since there are two points to the left of x=2x = 2x=2, we add 2∗(+1)2 \* (+1)2∗(+1) to d. And d is offset by –1 since

there is one point to the right. This means the distance had overall increased by 1.

Time complexity: O(mnlog(mn))

Space complexity: O(mn)

class Solution {

public int minTotalDistance(int[][] grid) {

List<Integer> rows = new ArrayList<>();

List<Integer> cols = new ArrayList<>();

for (int i = 0; i < grid.length; i++){

for (int j = 0; j < grid[0].length; j++){

if (grid[i][j] == 1){

rows.add(i);

cols.add(j);

}

}

}

int row = rows.get(rows.size()/2); //no need to sort rows since it is already sorted

Collections.sort(cols);

int col = cols.get(cols.size()/2);

return dist(rows, row) + dist(cols, col);

}

private int dist(List<Integer> list, int origin){

int sum = 0;

for (int p : list){

sum += Math.abs(p - origin);

}

return sum;

}

}

Method 3:replace sorting with directly collect data

Time complexity: O(mn)

Space complexity: O(mn)

class Solution {

public int minTotalDistance(int[][] grid) {

List<Integer> rows = collectRows(grid);

List<Integer> cols = collectCols(grid);

int row = rows.get(rows.size()/2); //no need to sort rows since it is already sorted

int col = cols.get(cols.size()/2);

return dist(rows, row) + dist(cols, col);

}

private int dist(List<Integer> list, int origin){

int sum = 0;

for (int p : list){

sum += Math.abs(p - origin);

}

return sum;

}

private List<Integer> collectRows(int[][] grid){

List<Integer> rows = new ArrayList<>();

for (int i = 0; i < grid.length; i++){

for (int j = 0; j < grid[0].length; j++){

if (grid[i][j] == 1){

rows.add(i);

}

}

}

return rows;

}

private List<Integer> collectCols(int[][] grid){

List<Integer> cols = new ArrayList<>();

for (int j = 0; j < grid[0].length; j++){

for (int i = 0; i < grid.length; i++){

if (grid[i][j] == 1){

cols.add(j);

}

}

}

return cols;

}

}

Method 4: (best solution) calculate distance without knowing the median point

Time complexity: O(mn)

Space compleixty: O(mn)

class Solution {

public int minTotalDistance(int[][] grid) {

List<Integer> rows = collectRows(grid);

List<Integer> cols = collectCols(grid);

return dist(rows) + dist(cols);

}

private int dist(List<Integer> list){

int sum = 0;

int i = 0;

int j = list.size() - 1;

while (i < j){

sum += list.get(j) - list.get(i);

i++;

j--;

}

return sum;

}

private List<Integer> collectRows(int[][] grid){

List<Integer> rows = new ArrayList<>();

for (int i = 0; i < grid.length; i++){

for (int j = 0; j < grid[0].length; j++){

if (grid[i][j] == 1){

rows.add(i);

}

}

}

return rows;

}

private List<Integer> collectCols(int[][] grid){

List<Integer> cols = new ArrayList<>();

for (int j = 0; j < grid[0].length; j++){

for (int i = 0; i < grid.length; i++){

if (grid[i][j] == 1){

cols.add(j);

}

}

}

return cols;

}

}

# 299\_Bulls.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the

number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret

number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong

position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number.

For example:

Secret number: "1807"

Friend's guess: "7810"

Hint: 1 bull and 3 cows. (The bull is 8, the cows are 0, 1 and 7.)

Write a function to return a hint according to the secret number and friend's guess, use A to indicate the bulls and B to indicate the cows. In the above example, your function should return "1A3B".

Please note that both secret number and friend's guess may contain duplicate digits, for example:

Secret number: "1123"

Friend's guess: "0111"

In this case, the 1st 1 in friend's guess is a bull, the 2nd or 3rd 1 is a cow, and your function should return "1A1B".

You may assume that the secret number and your friend's guess only contain digits, and their lengths are always equal.

class Solution {

public String getHint(String secret, String guess) {

int A = 0;

int B = 0;

Map<Character, Integer> map = new HashMap();

for (int i = 0; i < secret.length(); i++){

char gchar = guess.charAt(i);

char schar = secret.charAt(i);

if (gchar == schar){

A++;

}else{

map.put(schar, map.getOrDefault(schar, 0) + 1);

}

}

for (int i = 0; i < guess.length(); i++){

char gchar = guess.charAt(i);

char schar = secret.charAt(i);

if (gchar != schar && map.containsKey(gchar) && map.get(gchar) != 0){

B++;

map.put(gchar, map.get(gchar) - 1);

}

}

return A + "A" + B + "B";

}

}

https://leetcode.com/problems/bulls-and-cows/discuss/74621/One-pass-Java-solution

class Solution {

public String getHint(String secret, String guess) {

int A = 0;

int B = 0;

int[] hash = new int[10];

for (int i = 0; i < secret.length(); i++){

char gchar = guess.charAt(i);

char schar = secret.charAt(i);

if (gchar == schar){

A++;

}else{

if (hash[schar -'0']++ < 0){

B++;

}

if (hash[gchar - '0']-- > 0){

B++;

}

}

}

return A + "A" + B + "B";

}

}

best solution:

class Solution {

public String getHint(String secret, String guess) {

int A = 0;

int B = 0;

int len = secret.length();

boolean[] bull = new boolean[len];

for (int i = 0; i < len; i++){

if (secret.charAt(i) == guess.charAt(i)){

A++;

bull[i] = true;

}

}

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i < len; i++){

if (!bull[i]){

char c = secret.charAt(i);

map.put(c, map.getOrDefault(c, 0) + 1);

}

}

for (int i = 0; i < len; i++){

if (!bull[i]){

char c = guess.charAt(i);

if (map.containsKey(c) && map.get(c) > 0){

map.put(c, map.get(c) - 1);

B++;

}

}

}

return A + "A" + B + "B";

}

}

# Intervals

## 30\_InsertInterval.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-overlapping interval list which is sorted by start point.

Insert a new interval into it, make sure the list is still in order and non-overlapping (merge intervals if necessary).

Have you met this question in a real interview? Yes

Example

Insert [2, 5] into [[1,2], [5,9]], we get [[1,9]].

Insert [3, 4] into [[1,2], [5,9]], we get [[1,2], [3,4], [5,9]].

Method 1: based on 156. Merge Interval

/\*\*

\* Definition of Interval:

\* public classs Interval {

\* int start, end;

\* Interval(int start, int end) {

\* this.start = start;

\* this.end = end;

\* }

\*/

public class Solution {

/\*

\* @param intervals: Sorted interval list.

\* @param newInterval: new interval.

\* @return: A new interval list.

\*/

public List<Interval> insert(List<Interval> intervals, Interval newInterval) {

List<Interval> result = new ArrayList<Interval>();

if (intervals == null ||intervals.size() == 0){

result.add(newInterval);

return result;

}

int idx = 0;

while (idx < intervals.size() && intervals.get(idx).start < newInterval.start){

idx++;

}

intervals.add(idx, newInterval);

//the following code is the same as 156 Merge Interval

Interval last = null;

for (Interval item : intervals){

if (last == null || last.end < item.start){

result.add(item);

last = item;

}else{

last.end = Math.max(last.end, item.end);

}

}

return result;

}

}

Method 2:

/\*\*

\* Definition of Interval:

\* public classs Interval {

\* int start, end;

\* Interval(int start, int end) {

\* this.start = start;

\* this.end = end;

\* }

\*/

public class Solution {

/\*

\* @param intervals: Sorted interval list.

\* @param newInterval: new interval.

\* @return: A new interval list.

\*/

public List<Interval> insert(List<Interval> intervals, Interval newInterval) {

List<Interval> result = new ArrayList<>();

if (intervals == null || intervals.size() == 0){

result.add(newInterval);

return result;

}

int idx = 0;

int size = intervals.size();

while (idx < size && intervals.get(idx).end < newInterval.start){

result.add(intervals.get(idx));

idx++;

}

while (idx < size && intervals.get(idx).start <= newInterval.end){

newInterval.start = Math.min(newInterval.start,

intervals.get(idx).start);

newInterval.end = Math.max(newInterval.end, intervals.get(idx).end);

idx++;

}

result.add(newInterval);

while (idx < size){

result.add(intervals.get(idx));

idx++;

}

return result;

}

}

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public List<Interval> insert(List<Interval> intervals, Interval newInterval) {

List<Interval> res = new ArrayList<>();

intervals.add(newInterval);

Collections.sort(intervals, new Comparator<Interval>(){

public int compare (Interval i1, Interval i2){

return i1.start - i2.start;

}

});

Interval last = null;

for (Interval interval : intervals){

if (last == null || last.end < interval.start){

last = interval;

res.add(interval);

}else{

last.end = Math.max(last.end, interval.end);

}

}

return res;

}

}

Best solution:

class Solution {

public List<Interval> insert(List<Interval> intervals, Interval newInterval) {

List<Interval> res = new ArrayList<>();

if (intervals == null){

res.add(newInterval);

return res;

}

int idx = 0;

while (idx < intervals.size() && intervals.get(idx).start < newInterval.start){

idx++;

}

intervals.add(idx, newInterval);

Interval last = null;

for (int i = 0; i < intervals.size(); i++){

if (last == null || last.end < intervals.get(i).start){

res.add(intervals.get(i));

last = intervals.get(i);

}else{

last.end = Math.max(last.end, intervals.get(i).end);

}

}

return res;

}

}

## 435\_ Non-overlapping Intervals.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a collection of intervals, find the minimum number of intervals you need to remove to make the

rest of the intervals non-overlapping.

Note:

You may assume the interval's end point is always bigger than its start point.

Intervals like [1,2] and [2,3] have borders "touching" but they don't overlap each other.

Example 1:

Input: [ [1,2], [2,3], [3,4], [1,3] ]

Output: 1

Explanation: [1,3] can be removed and the rest of intervals are non-overlapping.

Example 2:

Input: [ [1,2], [1,2], [1,2] ]

Output: 2

Explanation: You need to remove two [1,2] to make the rest of intervals non-overlapping.

Example 3:

Input: [ [1,2], [2,3] ]

Output: 0

Explanation: You don't need to remove any of the intervals since they're already non-overlapping.

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

The similar as merge interval, but the only difference is find min, not max in merge interval

last.end = Math.min(last.end, item.end);

class Solution {

public int eraseOverlapIntervals(Interval[] intervals) {

if (intervals == null || intervals.length == 0){

return 0;

}

Arrays.sort(intervals, new Comparator<Interval>(){

public int compare (Interval a, Interval b){

if (a.start == b.start){

return a.end - b.end;

}

return a.start - b.start;

}

});

Interval last = null;

int count = 0;

for (Interval item : intervals){

if (last == null || last.end <= item.start){

last = item;

}else{

last.end = Math.min(last.end, item.end);

count++;

}

}

return count;

}

}

## 436\_ Find Right Interval.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a set of intervals, for each of the interval i, check if there exists an interval j whose start point is bigger than or

equal to the end point of the interval i, which can be called that j is on the "right" of i.

For any interval i, you need to store the minimum interval j's index, which means that the interval j has the minimum start

point to build the "right" relationship for interval i. If the interval j doesn't exist, store -1 for the interval i.

Finally, you need output the stored value of each interval as an array.

Note:

You may assume the interval's end point is always bigger than its start point.

You may assume none of these intervals have the same start point.

Example 1:

Input: [ [1,2] ]

Output: [-1]

Explanation: There is only one interval in the collection, so it outputs -1.

Example 2:

Input: [ [3,4], [2,3], [1,2] ]

Output: [-1, 0, 1]

Explanation: There is no satisfied "right" interval for [3,4].

For [2,3], the interval [3,4] has minimum-"right" start point;

For [1,2], the interval [2,3] has minimum-"right" start point.

Example 3:

Input: [ [1,4], [2,3], [3,4] ]

Output: [-1, 2, -1]

Explanation: There is no satisfied "right" interval for [1,4] and [3,4].

For [2,3], the interval [3,4] has minimum-"right" start point.

Method 1: HashMap O(nlogn)

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public int[] findRightInterval(Interval[] intervals) {

int[] result = new int [intervals.length];

Map<Integer, Integer> map = new HashMap<>();

int[] startPts = new int[intervals.length];

for (int i = 0; i < intervals.length; i++){

map.put(intervals[i].start, i);

startPts[i] = intervals[i].start;

}

Arrays.sort(startPts);

for (int i = 0; i < intervals.length; i++){

int key = binarySearch(startPts, intervals[i].end); //binary search end point in sorted start points array

if (key != Integer.MAX\_VALUE){

result[i] = map.get(key);

}else{

result[i] = -1;

}

}

return result;

}

private int binarySearch(int[] startPts, int target){

int left = 0;

int right = startPts.length - 1;

while (left + 1 < right){

int mid = left + (right - left) / 2;

int key = startPts[mid];

if (key == target){

return key;

}else if (key > target){

right = mid;

}else{

left = mid;

}

}

if (startPts[left] >= target){

return startPts[left];

}else if (startPts[right] >= target){

return startPts[right];

}

return Integer.MAX\_VALUE; // return start point value not index

}

}

Method 2: TreeMap O(nlogn)

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public int[] findRightInterval(Interval[] intervals) {

int[] result = new int [intervals.length];

TreeMap<Integer, Integer> treeMap = new TreeMap<>();

for (int i = 0; i < intervals.length; i++){

treeMap.put(intervals[i].start, i);

}

for (int i = 0; i < intervals.length; i++){

Map.Entry<Integer, Integer> entry = treeMap.ceilingEntry(intervals[i].end);

if (entry != null){

result[i] = entry.getValue();

}else{

result[i] = -1;

}

}

return result;

}

}

TreeMap comparator is implemented the same as others

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public int[] findRightInterval(Interval[] intervals) {

int[] result = new int [intervals.length];

TreeMap<Integer, Integer> treeMap = new TreeMap<>(new Comparator<Integer>(){

public int compare (Integer a, Integer b){

return a - b;

}

});

for (int i = 0; i < intervals.length; i++){

treeMap.put(intervals[i].start, i);

}

for (int i = 0; i < intervals.length; i++){

Map.Entry<Integer, Integer> entry = treeMap.ceilingEntry(intervals[i].end);

if (entry != null){

result[i] = entry.getValue();

}else{

result[i] = -1;

}

}

return result;

}

}

Best solution:

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public int[] findRightInterval(Interval[] intervals) {

int n = intervals.length;

int[] res = new int[n];

TreeMap<Integer, Integer> treemap = new TreeMap<>();

for (int i = 0; i < n; i++){

treemap.put(intervals[i].start, i);

}

for (int i = 0; i < n; i++){

Integer key = treemap.ceilingKey(intervals[i].end);

if (key == null){

res[i] = -1;

}else{

res[i] = treemap.get(key);

}

}

return res;

}

}

## 252\_MeetingRooms.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of meeting time intervals consisting of start and end times

[[s1,e1],[s2,e2],...] (si < ei), determine if a person could attend all meetings.

For example,

Given [[0, 30],[5, 10],[15, 20]],

return false.

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public boolean canAttendMeetings(Interval[] intervals) {

if (intervals == null || intervals.length == 0){

return true;

}

Arrays.sort(intervals, new Comparator<Interval>(){

public int compare(Interval a, Interval b){

return a.start - b.start;

}

});

for (int i = 1; i < intervals.length; i++){

if (intervals[i].start < intervals[i-1].end){

return false;

}

}

return true;

}

}

Method 2:

Sweep Line O(nlogn), n is the number of points

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

class Point{

int time;

int flag;

public Point(int time, int flag){

this.time = time;

this.flag = flag;

}

}

public boolean canAttendMeetings(Interval[] intervals) {

if (intervals == null || intervals.length == 0){

return true;

}

List<Point> list = new ArrayList<>();

for (Interval i : intervals){

list.add(new Point(i.start, 1));

list.add(new Point(i.end, 0));

}

Collections.sort(list, new Comparator<Point>(){

public int compare(Point a, Point b){

if (a.time == b.time){

return a.flag - b.flag;

}

return a.time - b.time;

}

});

int count = 0;

for (Point p : list){

if (p.flag == 1){

count++;

}else{

count--;

}

if (count > 1){

return false;

}

}

return true;

}

}

Method 3: the same as Meeting rooms II

Time complexity: O(nlogn)

class Solution {

public int minMeetingRooms(Interval[] intervals) {

if (intervals == null || intervals.length == 0){

return 0;

}

TreeMap<Integer, Integer> map = new TreeMap<>();

for(Interval interval : intervals){

map.put(interval.start, map.getOrDefault(interval.start, 0) + 1);

map.put(interval.end, map.getOrDefault(interval.end, 0) - 1);

}

int active = 0;

for (int val : map.values()){

active += val;

if (active > 1){

return false;

}

}

return true;

}

}

## 253\_MeetingRoomsII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei),

find the minimum number of conference rooms required.

For example,

Given [[0, 30],[5, 10],[15, 20]],

return 2.

Method 1:

This is a two pointer problem (greedy solution).

Sort start and end intervals. Take two pointers, one for start time and one for end time.

if the start interval is less than the end interval increment the room counter

since we would need a extra room, else decrement the count since we have freed up the room.

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public int minMeetingRooms(Interval[] intervals) {

if (intervals == null || intervals.length == 0){

return 0;

}

int len = intervals.length;

int[] starts = new int[len];

int[] ends = new int[len];

for (int i = 0; i < len; i++){

starts[i] = intervals[i].start;

ends[i] = intervals[i].end;

}

Arrays.sort(starts);

Arrays.sort(ends);

int startPtr = 0;

int endPtr = 0;

int count = 0;

int max = 0;

while(startPtr < len){

if (starts[startPtr] < ends[endPtr]){

count++;

startPtr++;

}else{

count--;

endPtr++;

}

max = Math.max(max, count);

}

return max;

}

}

Method 2:

Sweep Line: O(nlogn)

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

class Point{

int time;

int flag;

public Point(int time, int flag){

this.time = time;

this.flag = flag;

}

}

public int minMeetingRooms(Interval[] intervals) {

if (intervals == null || intervals.length == 0){

return 0;

}

List<Point> list = new ArrayList<>();

for(Interval i : intervals){

list.add(new Point(i.start, 1));

list.add(new Point(i.end, 0));

}

Collections.sort(list, new Comparator<Point>(){

public int compare(Point a, Point b){

if (a.time == b.time){

return a.flag - b.flag;//end goes ahead

}

return a.time - b.time;

}

});

int count = 0;

int ans = 0;

for(Point p : list){

if (p.flag == 1){

count++;

}else{

count--;

}

ans = Math.max(ans, count);

}

return ans;

}

}

Method 3: Best solution (inspired by My calendar II), the same s My calendar III

Time complexity: O(nlogn)

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public int minMeetingRooms(Interval[] intervals) {

if (intervals == null || intervals.length == 0){

return 0;

}

TreeMap<Integer, Integer> map = new TreeMap<>();

for(Interval interval : intervals){

map.put(interval.start, map.getOrDefault(interval.start, 0) + 1);

map.put(interval.end, map.getOrDefault(interval.end, 0) - 1);

}

int active = 0;

int max = Integer.MIN\_VALUE;

for (int val : map.values()){

active += val;

max = Math.max(max, active);

}

return max;

}

}

# 30\_Substring.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a string, s, and a list of words, words, that are all of the same length.

Find all starting indices of substring(s) in s that is a concatenation of each word in

words exactly once and without any intervening characters.

For example, given:

s: "barfoothefoobarman"

words: ["foo", "bar"]

You should return the indices: [0,9].

(order does not matter).

Time complexity: O(N \* M)

class Solution {

public List<Integer> findSubstring(String s, String[] words) {

List<Integer> res = new ArrayList<>();

if (words == null || words.length == 0){

return res;

}

Map<String, Integer> map = new HashMap<>();

for (String word : words){

map.put(word, map.getOrDefault(word, 0) + 1);

}

int len = words[0].length();

int n = words.length;

int start = 0;

int last = s.length() - n \* len;

while (start <= last){

Map<String, Integer> copy = new HashMap<>(map);

for (int i = 0; i < words.length; i++){

String sub = s.substring(start + i \* len, start + i \* len + len);

if (copy.containsKey(sub)){

copy.put(sub, copy.get(sub) - 1);

if (copy.get(sub) == 0){

copy.remove(sub);

}

}else{

break;

}

if (copy.isEmpty()){

res.add(start);

}

}

start++;

}

return res;

}

}

# 302\_Smallest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

An image is represented by a binary matrix with 0 as a white pixel and 1 as a black pixel. The black pixels are connected, i.e.,

there is only one black region. Pixels are connected horizontally and vertically. Given the location (x, y) of one of the black

pixels, return the area of the smallest (axis-aligned) rectangle that encloses all black pixels.

Example:

Input:

[

"0010",

"0110",

"0100"

]

and x = 0, y = 2

Output: 6

Method 1:

Time complexity: O(mn)

class Solution {

public int minArea(char[][] image, int x, int y) {

int m = image.length;

int n = image[0].length;

int[] rowsum = new int[m];

int[] colsum = new int[n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

rowsum[i] += (int)(image[i][j] - '0');

}

}

for (int j = 0; j < n; j++){

for (int i = 0; i < m; i++){

colsum[j] += (int)(image[i][j] - '0');

}

}

int top = 0;

for (int i = 0; i < m; i++){

if (rowsum[i] > 0){

top = i;

break;

}

}

int bottom = 0;

for (int i = m - 1; i >= 0; i--){

if (rowsum[i] > 0){

bottom = i;

break;

}

}

int left = 0;

for (int j = 0; j < n; j++){

if (colsum[j] > 0){

left = j;

break;

}

}

int right = 0;

for (int j = n - 1; j >= 0; j--){

if (colsum[j] > 0){

right = j;

break;

}

}

return (right - left + 1) \* (bottom - top + 1);

}

}

Best solution:

Method 2: binary search

Time complexity: O(mlogn or nlogm)

class Solution {

public int minArea(char[][] image, int x, int y) {

int m = image.length;

int n = image[0].length;

int left = searchFirst(image, 0, y, true);

int right = searchLast(image, y, n - 1, true);

int top = searchFirst(image, 0, x, false);

int bottom = searchLast(image, x, m - 1, false);

return (right - left + 1) \* (bottom - top + 1);

}

private int searchFirst(char[][] image, int start, int end, boolean checkVertical){

while (start + 1 < end){

int mid = (start + end) / 2;

if (hasBlack(image, mid, checkVertical)){

end = mid;

}else{

start = mid;

}

}

if (hasBlack(image, start, checkVertical)){

return start;

}

return end;

}

private int searchLast(char[][] image, int start, int end, boolean checkVertical){

while (start + 1 < end){

int mid = (start + end) / 2;

if (hasBlack(image, mid, checkVertical)){

start = mid;

}else{

end = mid;

}

}

if (hasBlack(image, end, checkVertical)){

return end;

}

return start;

}

private boolean hasBlack(char[][] image, int ind, boolean checkVertical){

int m = image.length;

int n = image[0].length;

int sum = 0;

if (checkVertical){

for (int i = 0; i < m; i++){

sum += (int)(image[i][ind] - '0');

}

}else{

for (int j = 0; j < n; j++){

sum += (int)(image[ind][j] - '0');

}

}

return sum > 0;

}

}

# 303\_Range.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array nums, find the sum of the elements between indices i and j (i ≤ j), inclusive.

Example:

Given nums = [-2, 0, 3, -5, 2, -1]

sumRange(0, 2) -> 1

sumRange(2, 5) -> -1

sumRange(0, 5) -> -3

Note:

You may assume that the array does not change.

There are many calls to sumRange function.

class NumArray {

int[] preSum;

public NumArray(int[] nums) {

int n = nums.length;

preSum = new int[n+1];

preSum[0] = 0;

for (int i = 1; i <= n; i++){

preSum[i] = preSum[i-1] + nums[i-1];

}

}

public int sumRange(int i, int j) {

return preSum[j+1] - preSum[i];

}

}

/\*\*

\* Your NumArray object will be instantiated and called as such:

\* NumArray obj = new NumArray(nums);

\* int param\_1 = obj.sumRange(i,j);

\*/

# 304\_Range.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

Range Sum Query 2D

The above rectangle (with the red border) is defined by (row1, col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum = 8.

Example:

Given matrix = [

[3, 0, 1, 4, 2],

[5, 6, 3, 2, 1],

[1, 2, 0, 1, 5],

[4, 1, 0, 1, 7],

[1, 0, 3, 0, 5]

]

sumRegion(2, 1, 4, 3) -> 8

sumRegion(1, 1, 2, 2) -> 11

sumRegion(1, 2, 2, 4) -> 12

Method 1: Best solution

dp on consructor

class NumMatrix {

int[][] preSum;

public NumMatrix(int[][] matrix) {

if (matrix == null || matrix.length == 0){

return;

}

int m = matrix.length;

int n = matrix[0].length;

preSum = new int[m+1][n+1];

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

preSum[i][j] = preSum[i-1][j] + preSum[i][j-1] - preSum[i-1][j-1] + matrix[i-1][j-1];

}

}

}

public int sumRegion(int row1, int col1, int row2, int col2) {

return preSum[row2+1][col2+1] - preSum[row1][col2+1] - preSum[row2+1][col1] + preSum[row1][col1];

}

}

/\*\*

\* Your NumMatrix object will be instantiated and called as such:

\* NumMatrix obj = new NumMatrix(matrix);

\* int param\_1 = obj.sumRegion(row1,col1,row2,col2);

\*/

Method 2:

use sum to store the row preSum

class NumMatrix {

int[][] preSum;

public NumMatrix(int[][] matrix) {

if (matrix == null || matrix.length == 0){

return;

}

int m = matrix.length;

int n = matrix[0].length;

preSum = new int[m+1][n+1];

for (int i = 0; i < matrix.length; i++) {

int sum = 0;

for (int j = 0; j < matrix[0].length; j++) {

sum += matrix[i][j];

preSum[i + 1][j + 1] = sum + preSum[i][j + 1];

}

}

}

public int sumRegion(int row1, int col1, int row2, int col2) {

return preSum[row2+1][col2+1] - preSum[row1][col2+1] - preSum[row2+1][col1] + preSum[row1][col1];

}

}

/\*\*

\* Your NumMatrix object will be instantiated and called as such:

\* NumMatrix obj = new NumMatrix(matrix);

\* int param\_1 = obj.sumRegion(row1,col1,row2,col2);

\*/

Method 3:

First pass to get the row preSum

Second pass to get the row+col preSum

class NumMatrix {

int[][] preSum;

public NumMatrix(int[][] matrix) {

if (matrix == null || matrix.length == 0){

return;

}

int m = matrix.length;

int n = matrix[0].length;

preSum = new int[m+1][n+1];

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

preSum[i][j] = preSum[i][j-1] + matrix[i-1][j-1];

}

}

for (int j = 1; j <= n; j++){

for (int i = 1; i <= m; i++){

preSum[i][j] += preSum[i-1][j];

}

}

}

public int sumRegion(int row1, int col1, int row2, int col2) {

return preSum[row2+1][col2+1] - preSum[row1][col2+1] - preSum[row2+1][col1] + preSum[row1][col1];

}

}

/\*\*

\* Your NumMatrix object will be instantiated and called as such:

\* NumMatrix obj = new NumMatrix(matrix);

\* int param\_1 = obj.sumRegion(row1,col1,row2,col2);

\*/

# 31\_NextPermutation.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Solution:

Step1, from right to left, find the first number which not increase in a ascending order. In this case which is 3.

Step2, here we can have two situations:

We cannot find the number, all the numbers increasing in a ascending order. This means this permutation is the

last permutation, we need to rotate back to the first permutation. So we reverse the whole array, for example,

6,5,4,3,2,1 we turn it to 1,2,3,4,5,6.

We can find the number, then the next step, we will start from right most to leftward, try to find the first

number which is larger than 3, in this case it is 4.

Then we swap 3 and 4, the list turn to 2,4,6,5,3,1.

Last, we reverse numbers on the right of 4, we finally get 2,4,1,3,5,6.

Time complexity is: O(3\*n)=O(n).

class Solution {

public void nextPermutation(int[] nums) {

int len = nums.length;

//step 1: find the first number that breaks ascending order from right most

int left = 0;

int i;

boolean lowest = true;

for (i = len - 2; i >= 0; i--){

if (nums[i] < nums[i+1]){

left = i;

lowest = false;

break;

}

}

if (lowest){

reverse(nums, 0, len - 1);

return;

}

//step 2: find the smallest number that is greater than the number in step 1

int right = left;

int min = Integer.MAX\_VALUE;

for (i = left + 1; i < len; i++){

if (nums[i] > nums[left] && nums[i] <= min){ // need use to "=" to ensure the smallest value goes to the right most

right = i;

min = nums[i];

}

}

swap(nums, left, right);

//step 3: reverse nums starting from the index of the number in step 1

reverse(nums, left + 1, len - 1);

}

private void reverse(int[] A, int start, int end){

while (start < end){

swap(A, start, end);

start++;

end--;

}

}

private void swap(int[] A, int i, int j){

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

Better version:

class Solution {

public void nextPermutation(int[] nums) {

//three steps:

//step 1: find the first non-ascending element from right to left

if (nums == null || nums.length == 0){

return;

}

int n = nums.length;

int index = n;

for (int i = n - 1; i > 0; i--){

if (nums[i-1] < nums[i]){

index = i-1;

break;

}

}

if (index == n){//reverse the nums

reverse(nums, 0, n-1);

return;

}

int pivot = nums[index];

//step 2: find the first element that is greater than pivot from right to left

for (int i = n - 1; i > index; i--){

if (nums[i] > pivot){

swap(nums, index, i);

break;

}

}

//step 3: reverse the elements from index + 1 to right

reverse(nums, index+1, n-1);

}

private void reverse(int[] nums, int start, int end){

while (start < end){

int temp = nums[start];

nums[start] = nums[end];

nums[end] = temp;

start++;

end--;

}

}

private void swap(int[] nums, int start, int end){

int temp = nums[start];

nums[start] = nums[end];

nums[end] = temp;

}

}

# 31\_PartitionArray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array nums of integers and an int k, partition the array (i.e move the elements in "nums") such that:

All elements < k are moved to the left

All elements >= k are moved to the right

Return the partitioning index, i.e the first index i nums[i] >= k.

Notice

You should do really partition in array nums instead of just counting the numbers of integers smaller than k.

If all elements in nums are smaller than k, then return nums.length

Have you met this question in a real interview? Yes

Example

If nums = [3,2,2,1] and k=2, a valid answer is 1.

Challenge

Can you partition the array in-place and in O(n)?

public class Solution {

/\*

\* @param nums: The integer array you should partition

\* @param k: An integer

\* @return: The index after partition

\*/

public int partitionArray(int[] nums, int k) {

if (nums == null || nums.length == 0){

return 0;

}

int left = 0;

int right = nums.length - 1;

while (left < right){

while (left < right && nums[left] < k){

left++;

}

while (left < right && nums[right] >= k){

right--;

}

if (left < right){

int temp = nums[left];

nums[left] = nums[right];

nums[right] = temp;

left++;

right--;

}

}

if (nums[left] < k){

return left + 1;

}

return left;

}

}

# 310\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree.

Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a

function to find all the MHTs and return a list of their root labels.

Format

The graph contains n nodes which are labeled from 0 to n - 1. You will be given the number n and a list of undirected edges

(each edge is a pair of labels).

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0]

and thus will not appear together in edges.

Example 1:

Given n = 4, edges = [[1, 0], [1, 2], [1, 3]]

0

|

1

/ \

2 3

return [1]

Example 2:

Given n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]

0 1 2

\ | /

3

|

4

|

5

return [3, 4]

Note:

(1) According to the definition of tree on Wikipedia: “a tree is an undirected graph in which any two vertices are

connected by exactly one path. In other words, any connected graph without simple cycles is a tree.”

(2) The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

Method 1 :

Time complexity: O(n)

similar as topological sorting course

https://leetcode.com/problems/minimum-height-trees/discuss/76055/Share-some-thoughts

class Solution {

public List<Integer> findMinHeightTrees(int n, int[][] edges) {

List<Integer> result = new ArrayList<>();

if (n <= 0 || edges == null || n != edges.length + 1){

return result;

}

if (n == 1){

result.add(0);

return result;

}

Map<Integer, Set<Integer>> graph = initializeGraph(n, edges);

for (int i : graph.keySet()){

if (graph.get(i).size() == 1){

result.add(i);

}

}

while (n > 2){

n -= result.size();

List<Integer> newResult = new ArrayList<>();

for (int i : result){

for (int neighbor: graph.get(i)){

graph.get(neighbor).remove(i);

if (graph.get(neighbor).size() == 1){

newResult.add(neighbor);

}

}

}

result = newResult;

}

return result;

}

private Map<Integer, Set<Integer>> buildGraph(int n, int[][] edges){

Map<Integer, Set<Integer>> graph = new HashMap<>();

for (int i = 0; i < n; i++){

graph.put(i, new HashSet<>());

}

for (int[] edge : edges){

graph.get(edge[0]).add(edge[1]);

graph.get(edge[1]).add(edge[0]);

}

return graph;

}

}

Method 2:

Time complexity: O(n^2) TLE

class Solution {

public List<Integer> findMinHeightTrees(int n, int[][] edges) {

List<Integer> result = new ArrayList<>();

if (n <= 0 || edges == null || n != edges.length + 1){

return result;

}

Map<Integer, Set<Integer>> graph = initializeGraph(n, edges);

int min = Integer.MAX\_VALUE;

for (int i = 0; i < n; i++){

int mht = calculateMHT(graph, i);

if (mht < min){

result = new ArrayList<>();

result.add(i);

min = mht;

}else if (mht == min){

result.add(i);

}

}

return result;

}

private Map<Integer, Set<Integer>> initializeGraph (int n, int[][] edges){

Map<Integer, Set<Integer>> result = new HashMap<>();

for (int i = 0; i < n; i++){

result.put(i, new HashSet<Integer>());

}

for (int i = 0; i < edges.length; i++){

int u = edges[i][0];

int v = edges[i][1];

result.get(u).add(v);

result.get(v).add(u);

}

return result;

}

private int calculateMHT(Map<Integer, Set<Integer>> graph, int root){

Queue<Integer> queue = new LinkedList<>();

Set<Integer> set = new HashSet<>();

queue.offer(root);

set.add(root);

int height = 0;

while (!queue.isEmpty()){

int size = queue.size();

height++;

for (int i = 0; i < size; i++){

int curr = queue.poll();

for (int neighbor : graph.get(curr)){

if (!set.contains(neighbor)){

queue.offer(neighbor);

set.add(neighbor);

}

}

}

}

return height;

}

}

# 312\_BurstBalloons.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums.

You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] \* nums[i] \* nums[right] coins.

Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note:

(1) You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them.

(2) 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

Example:

Given [3, 1, 5, 8]

Return 167

nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []

coins = 3\*1\*5 + 3\*5\*8 + 1\*3\*8 + 1\*8\*1 = 167

Best solution:

https://leetcode.com/problems/burst-balloons/discuss/194608/DP-solution-with-detailed-text-and-video-explanation.

class Solution {

public int maxCoins(int[] nums) {

int n = nums.length;

int[] arr = new int[n+2];

for (int i = 1; i <= n; i++){

arr[i] = nums[i-1];

}

arr[0] = 1;

arr[n+1] = 1;

int[][] dp = new int[n+2][n+2];

for (int len = 0; len <= n; len++){

for (int i = 1; i + len <= n; i++){

int j = i + len;

for (int k = i; k <= j; k++){

dp[i][j] = Math.max(dp[i][j], dp[i][k-1] + dp[k+1][j] + arr[k] \* arr[i-1] \* arr[j+1]);

}

}

}

return dp[1][n];

}

}

Method:

区间DP

O(n^2)

http://www.jiuzhang.com/solutions/burst-ballons/

class Solution {

public int maxCoins(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int n = nums.length;

int[] arr = new int[n+2];

for (int i = 1; i <= n; i++){

arr[i] = nums[i-1];

}

arr[0] = 1;

arr[n+1] = 1;

int[][] dp = new int[n+2][n+2];

int[][] visit = new int[n+2][n+2];

return memorySearch(dp, visit, arr, 1, n);

}

private int memorySearch(int[][] dp, int[][] visit, int[] arr, int l, int r){

if (visit[l][r] == 1){

return dp[l][r];

}

int ans = 0;

for (int k = l; k <= r; k++){

int midValue = arr[l-1] \* arr[k] \* arr[r+1];

int left = memorySearch(dp, visit, arr, l, k - 1);

int right = memorySearch(dp, visit, arr, k + 1, r);

ans = Math.max(ans, left + right + midValue);

}

visit[l][r] = 1;

dp[l][r] = ans;

return ans;

}

}

https://leetcode.com/problems/burst-balloons/discuss/76228/Share-some-analysis-and-explanations

class Solution {

public int maxCoins(int[] nums) {

int n = nums.length;

int[] arr = new int[n+2];

for (int i = 1; i <= n; i++){

arr[i] = nums[i-1];

}

arr[0] = 1;

arr[n+1] = 1;

Map<Integer, Integer> map = new HashMap<>();

return burst(arr, 1, n, map);

}

private int burst(int[] arr, int left, int right, Map<Integer, Integer> map){

if (left > right){

return 0;

}

int N = arr.length;

int key = left \* N + right;

if (map.containsKey(key)){

return map.get(key);

}

int res = 0;

for (int i = left; i <= right; i++){

int lastBurst = arr[left-1] \* arr[i] \* arr[right+1];

int leftBurst = burst(arr, left, i - 1, map);

int rightBurst = burst(arr, i + 1, right, map);

res = Math.max(res, lastBurst + leftBurst + rightBurst);

}

map.put(key, res);

return res;

}

}

# 313\_SuperUglyNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a program to find the nth super ugly number.

Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of size k.

Example:

Input: n = 12, primes = [2,7,13,19]

Output: 32

Explanation: [1,2,4,7,8,13,14,16,19,26,28,32] is the sequence of the first 12

super ugly numbers given primes = [2,7,13,19] of size 4.

Note:

1 is a super ugly number for any given primes.

The given numbers in primes are in ascending order.

0 < k ≤ 100, 0 < n ≤ 106, 0 < primes[i] < 1000.

The nth super ugly number is guaranteed to fit in a 32-bit signed integer.

Method 1: O(nlogn + onlogk) TLE

class Solution {

public int nthSuperUglyNumber(int n, int[] primes) {

Queue<Integer> pq = new PriorityQueue<Integer>();

Set<Integer> seen = new HashSet<>();

pq.offer(1);

seen.add(1);

int v = 1;

for (int i = 1; i <= n; i++){

v = pq.poll();

for (int j = 0; j < primes.length; j++){

int cand = v \* primes[j];

if (!seen.contains(cand)){

pq.offer(cand);

seen.add(cand);

}

}

}

return v;

}

}

Method 2: O(nk) the principle is like merge sort

The same as Ugly Number II

https://github.com/optimisea/Leetcode/blob/master/Java/264\_UglyNumberII.java

https://leetcode.com/problems/ugly-number-ii/discuss/69362/O(n)-Java-solution

https://leetcode.com/problems/super-ugly-number/discuss/76291/Java-three-methods-23ms-36-ms-58ms(with-heap)-performance-explained

class Solution {

public int nthSuperUglyNumber(int n, int[] primes) {

int[] res = new int[n];

res[0] = 1;

int[] indices = new int[primes.length];

for (int i = 1; i < n; i++){

res[i] = Integer.MAX\_VALUE;

for (int j = 0; j < primes.length; j++){

res[i] = Math.min(res[i], primes[j] \* res[indices[j]]);

}

for (int j = 0; j < primes.length; j++){

if (res[i] == primes[j] \* res[indices[j]]){

indices[j]++;

}

}

}

return res[n-1];

}

}

# 315\_Count.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given an integer array nums and you have to return a new counts array. The counts array has the property

where counts[i] is the number of smaller elements to the right of nums[i].

Example:

Given nums = [5, 2, 6, 1]

To the right of 5 there are 2 smaller elements (2 and 1).

To the right of 2 there is only 1 smaller element (1).

To the right of 6 there is 1 smaller element (1).

To the right of 1 there is 0 smaller element.

Return the array [2, 1, 1, 0].

Method 1: Brute force O(n^2) TLE

class Solution {

public List<Integer> countSmaller(int[] nums) {

List<Integer> result = new ArrayList<>();

for (int i = 0; i < nums.length; i++){

int count = 0;

for (int j = i+1; j < nums.length; j++){

if (nums[j] < nums[i]){

count++;

}

}

result.add(count);

}

return result;

}

}

Method 2: Binary tree

O(nLogn)

class Solution {

class TreeNode {

int val;

int count = 1;

TreeNode left;

TreeNode right;

public TreeNode (int val){

this.val = val;

}

}

public List<Integer> countSmaller(int[] nums) {

List<Integer> result = new ArrayList<>();

if (nums == null || nums.length == 0){

return result;

}

int end = nums.length - 1;

TreeNode root = new TreeNode(nums[end]);

result.add(0);

for (int i = end - 1; i >= 0; i--){

int count = insertNode(root, nums[i]);

result.add(0, count);

}

return result;

}

private int insertNode(TreeNode root, int val){

int thisCount = 0;

while(true) {

if(val <= root.val) {

root.count++;

if(root.left == null) {

root.left = new TreeNode(val); break;

} else {

root = root.left;

}

} else {

thisCount += root.count;

if(root.right == null) {

root.right = new TreeNode(val); break;

} else {

root = root.right;

}

}

}

return thisCount;

}

}

Method 3: Merge Sort Idea: Best solution

The smaller numbers on the right of a number are exactly those that jump from its right to its left during a stable sort.

So I do mergesort with added tracking of those right-to-left jumps.

https://leetcode.com/problems/count-of-smaller-numbers-after-self/discuss/76583/11ms-JAVA-solution-using-merge-sort-with-explanation

class Solution {

class Pair {

int val;

int index;

public Pair (int val, int index){

this.val = val;

this.index = index;

}

}

public List<Integer> countSmaller(int[] nums) {

List<Integer> res = new ArrayList<>();

int n = nums.length;

int[] count = new int[n];

Pair[] pair = new Pair[n];

Pair[] temp = new Pair[n];

for (int i = 0; i < n; i++){

pair[i] = new Pair(nums[i], i);

}

mergeSort(count, pair, temp, 0, n - 1);

for (int i = 0; i < n; i++){

res.add(count[i]);

}

return res;

}

private void mergeSort(int[] count, Pair[] pair, Pair[] temp, int start, int end){

if (start >= end){

return;

}

int mid = (start + end) / 2;

mergeSort(count, pair, temp, start, mid);

mergeSort(count, pair, temp, mid + 1, end);

merge(count, pair, temp, start, mid, end);

}

private void merge(int[] count, Pair[] pair, Pair[] temp, int start, int mid, int end){

int left = start;

int right = mid + 1;

int index = start;

int rightCount = 0;

while (left <= mid && right <= end){

if (pair[left].val > pair[right].val){

// temp[index++] = pair[right++];

temp[index] = new Pair(pair[right].val, pair[right].index);

rightCount++;

right++;

}else{

// temp[index++] = pair[left++];

temp[index] = new Pair(pair[left].val, pair[left].index);

count[pair[left].index] += rightCount;

left++;

}

index++;

}

while (left <= mid){

// temp[index++] = pair[left++];

temp[index] = new Pair(pair[left].val, pair[left].index);

count[pair[left].index] += rightCount;

left++;

index++;

}

while (right <= end){

// temp[index++] = pair[right++];

temp[index] = new Pair(pair[right].val, pair[right].index);

right++;

index++;

}

//copy back to pair

for (int i = start; i <= end; i++){

pair[i] = new Pair(temp[i].val, temp[i].index);

}

}

}

# 317\_Shortest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You want to build a house on an empty land which reaches all buildings in the shortest amount of distance. You can only move up,

down, left and right. You are given a 2D grid of values 0, 1 or 2, where:

Each 0 marks an empty land which you can pass by freely.

Each 1 marks a building which you cannot pass through.

Each 2 marks an obstacle which you cannot pass through.

Example:

Input: [[1,0,2,0,1],[0,0,0,0,0],[0,0,1,0,0]]

1 - 0 - 2 - 0 - 1

| | | | |

0 - 0 - 0 - 0 - 0

| | | | |

0 - 0 - 1 - 0 - 0

Output: 7

Explanation: Given three buildings at (0,0), (0,4), (2,2), and an obstacle at (0,2),

the point (1,2) is an ideal empty land to build a house, as the total

travel distance of 3+3+1=7 is minimal. So return 7.

Note:

There will be at least one building. If it is not possible to build such house according to the above rules, return -1.

Time complexity: O((m\*n)^2)

class Solution {

public int shortestDistance(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int res = Integer.MAX\_VALUE;

int count = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1){

count++;

}

}

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 0){

int dist = bfs(grid, i, j, count);

if (dist != -1){

res = Math.min(res, dist);

}

}

}

}

return res == Integer.MAX\_VALUE ? -1 : res;

}

private int bfs(int[][] grid, int x, int y, int count){

int res = 0;

int m = grid.length;

int n = grid[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

Set<Integer> set = new HashSet<>();

set.add(x \* n + y);

queue.offer(new int[]{x, y});

int level = 0;

while (!queue.isEmpty()){

level++;

int size = queue.size();

for (int i = 0; i < size; i++){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] != 2 && !set.contains(nx \* n + ny)){

set.add(nx \* n + ny);

if (grid[nx][ny] == 0){

queue.offer(new int[]{nx, ny});

}else{

res += level;

count--;

}

}

}

}

}

return count == 0 ? res : -1;

}

}

# 318\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string array words, find the maximum value of length(word[i]) \* length(word[j]) where the two words

do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

Example 1:

Given ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]

Return 16

The two words can be "abcw", "xtfn".

Example 2:

Given ["a", "ab", "abc", "d", "cd", "bcd", "abcd"]

Return 4

The two words can be "ab", "cd".

Example 3:

Given ["a", "aa", "aaa", "aaaa"]

Return 0

No such pair of words.

Method 1:

Time complexity: O(n^2 \* m)

Space complexity: O(n)

class Solution {

public int maxProduct(String[] words) {

int ans = 0;

for (int i = 0; i < words.length; i++){

Set<Character> set = new HashSet<>();

for (int k = 0; k < words[i].length(); k++){

set.add(words[i].charAt(k));

}

for (int j = i + 1; j < words.length; j++){

boolean isValid = true;

for (int m = 0; m < words[j].length(); m++){

if (set.contains(words[j].charAt(m))){

isValid = false;

break;

}

}

if (isValid){

ans = Math.max(ans, words[i].length() \* words[j].length());

}

}

}

return ans;

}

}

Method 2: use bit manipulation to hash and compare the two words do not share common letters

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public int maxProduct(String[] words) {

int ans = 0;

int len = words.length;

int[] hash = new int[len];

for (int i = 0; i < len; i++){

for (int j = 0; j < words[i].length(); j++){

hash[i] |= 1 << (words[i].charAt(j) - 'a');

}

}

for (int i = 0; i < len; i++){

for (int j = i + 1; j < len; j++){

if ((hash[i] & hash[j]) == 0){

ans = Math.max(ans, words[i].length() \* words[j].length());

}

}

}

return ans;

}

}

# 320\_GeneralizedAbbreviationsOfWord.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a function to generate the generalized abbreviations of a word.

Note: The order of the output does not matter.

Example:

Input: "word"

Output:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

he idea is: for every character, we can keep it or abbreviate it. To keep it, we add it to the current solution and carry

on backtracking. To abbreviate it, we omit it in the current solution, but increment the count, which indicates how many

characters have we abbreviated. When we reach

the end or need to put a character in the current solution, and count is bigger than zero, we add the number into the solution.

Method 1: Backtracking

class Solution {

public List<String> generateAbbreviations(String word) {

List<String> result = new ArrayList<>();

if (word == null){

return result;

}

if (word.length() == 0){

result.add(word);

return result;

}

backtrack(result, word, new StringBuilder(), 0, 0);

return result;

}

private void backtrack(List<String> result, String word, StringBuilder sb, int start, int count){

if (start == word.length()){

int currLen = sb.length();

if (count > 0){

sb.append(count);

}

result.add(sb.toString());

sb.setLength(currLen);

return;

}

backtrack(result, word, sb, start+1, count+1); // skip current char

//not skip

int currLen = sb.length();

if (count > 0){

sb.append(count);

}

sb.append(word.charAt(start));

backtrack(result, word, sb, start+1, 0);

sb.setLength(currLen);

}

}

Method 2: DFS Best solution

class Solution {

public List<String> generateAbbreviations(String word) {

List<String> result = new ArrayList<>();

if (word == null){

return result;

}

dfs(result, word, "", 0, 0);

return result;

}

private void dfs(List<String> result, String word, String str, int start, int count){

if (start == word.length()){

if (count > 0){

str += String.valueOf(count);

}

result.add(str);

return;

}

dfs(result, word, str, start+1, count+1); // skip current char

dfs(result, word, str + (count > 0 ? count : "") + word.charAt(start), start+1, 0); //not skip

}

}

# 321\_Create.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two arrays of length m and n with digits 0-9 representing two numbers. Create the maximum number of

length k <= m + n from digits of the two. The relative order of the digits from the same array must be preserved.

Return an array of the k digits. You should try to optimize your time and space complexity.

Example 1:

nums1 = [3, 4, 6, 5]

nums2 = [9, 1, 2, 5, 8, 3]

k = 5

return [9, 8, 6, 5, 3]

Example 2:

nums1 = [6, 7]

nums2 = [6, 0, 4]

k = 5

return [6, 7, 6, 0, 4]

Example 3:

nums1 = [3, 9]

nums2 = [8, 9]

k = 3

return [9, 8, 9]

https://leetcode.com/problems/create-maximum-number/discuss/77285/Share-my-greedy-solution

Time complexity: O((m+n)^2)

class Solution {

//step 1: find the max Array within one array

//step 2: merge two max arrays into one array

//step 3: find the max one with different combination of merge

public int[] maxNumber(int[] nums1, int[] nums2, int k) {

int n = nums1.length;

int m = nums2.length;

int[] ans = new int[k];

for (int i = Math.max(0, k - m); i <= k && i <= n; i++){

int[] maxArray1 = maxArray(nums1, i);

int[] maxArray2 = maxArray(nums2, k-i);

int[] candidate = merge(maxArray1, maxArray2, k);

if (greater(candidate, 0, ans, 0)){

ans = candidate;

}

}

return ans;

}

private int[] maxArray(int[] nums, int len){

int[] ans = new int[len];

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < nums.length; i++){

while (stack.size() + nums.length - i > len && !stack.isEmpty() && stack.peek() < nums[i]){

stack.pop();

}

if (stack.size() < len){

stack.push(nums[i]);

}

}

for (int i = len - 1; i >= 0; i--){

ans[i] = stack.pop();

}

return ans;

}

private int[] merge(int[] nums1, int[] nums2, int k){

int[] ans = new int[k];

for (int i = 0, j = 0, r = 0; r < k; r++){

ans[r] = greater(nums1, i, nums2, j) ? nums1[i++] : nums2[j++];

}

return ans;

}

private boolean greater(int[] nums1, int i, int[] nums2, int j){

while (i < nums1.length && j < nums2.length && nums1[i] == nums2[j]){

i++;

j++;

}

return j == nums2.length || (i < nums1.length && nums1[i] > nums2[j]);

}

}

# 322\_CoinChange.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given coins of different denominations and a total amount of money amount.

Write a function to compute the fewest number of coins that you need to make up that amount.

If that amount of money cannot be made up by any combination of the coins, return -1.

Example 1:

coins = [1, 2, 5], amount = 11

return 3 (11 = 5 + 5 + 1)

Example 2:

coins = [2], amount = 3

return -1.

Note:

You may assume that you have an infinite number of each kind of coin.

Method 1:

Recursion + memorization

class Solution {

Map<Integer, Integer> map = new HashMap<>();

public int coinChange(int[] coins, int amount) {

if (amount == 0){

return 0;

}

if (amount < 0){

return -1;

}

if (map.containsKey(amount)){

return map.get(amount);

}

int ans = Integer.MAX\_VALUE;

for (int coin : coins){

int curr = 0;

if (coin <= amount){

int next = coinChange(coins, amount - coin);

if (next >= 0){

curr = next + 1;

}

}

if (curr > 0){

ans = Math.min(ans, curr);

}

}

int count = (ans == Integer.MAX\_VALUE) ? -1 : ans;

map.put(amount, count);

return count;

}

}

class Solution {

Map<Integer, Integer> map = new HashMap<>();

public int coinChange(int[] coins, int amount) {

if (amount == 0){

return 0;

}

if (amount < 0){

return -1;

}

if (map.containsKey(amount)){

return map.get(amount);

}

int ans = Integer.MAX\_VALUE;

for (int coin : coins){

int curr = 0;

int next = coinChange(coins, amount - coin);

if (next >= 0){

curr = next + 1;

ans = Math.min(ans, curr);

}

}

int count = (ans == Integer.MAX\_VALUE) ? -1 : ans;

map.put(amount, count);

return count;

}

}

Method 2: DP like LIS

class Solution {

public int coinChange(int[] coins, int amount) {

int max = amount + 1;

int[] dp = new int[max];

Arrays.fill(dp, max);

dp[0] = 0;

for (int i = 1; i <= amount; i++){

for (int j = 0; j < coins.length; j++){

if (i - coins[j] >= 0){

dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);

}

}

}

return dp[amount] == max ? -1 : dp[amount];

}

}

Better:

class Solution {

public int coinChange(int[] coins, int amount) {

int[] dp = new int[amount+1];

for (int i = 1; i <= amount; i++){

dp[i] = Integer.MAX\_VALUE;

for (int j = 0; j < coins.length; j++){

if (i >= coins[j] && dp[i - coins[j]] != Integer.MAX\_VALUE){

dp[i] = Math.min(dp[i], dp[i-coins[j]] + 1);

}

}

}

return dp[amount] == Integer.MAX\_VALUE ? -1 : dp[amount];

}

}

# 325\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array nums and a target value k, find the maximum length of a subarray that sums to k. If there isn't one, return 0 instead.

Note:

The sum of the entire nums array is guaranteed to fit within the 32-bit signed integer range.

Example 1:

Input: nums = [1, -1, 5, -2, 3], k = 3

Output: 4

Explanation: The subarray [1, -1, 5, -2] sums to 3 and is the longest.

Example 2:

Input: nums = [-2, -1, 2, 1], k = 1

Output: 2

Explanation: The subarray [-1, 2] sums to 1 and is the longest.

Method: preSum

class Solution {

public int maxSubArrayLen(int[] nums, int k) {

if (nums == null || nums.length == 0){

return 0;

}

Map<Integer, Integer> map = new HashMap<>();

int max = 0;

int sum = 0;

map.put(0, -1);

for (int i = 0; i < nums.length; i++){

sum = sum + nums[i];

if (map.containsKey(sum - k)){

max = Math.max(max, i - map.get(sum - k));

}

if (!map.containsKey(sum)){//check to ensure not overwrite old values

map.put(sum, i);

}

}

return max;

}

}

# 326\_Power.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer, write a function to determine if it is a power of three.

Follow up:

Could you do it without using any loop / recursion?

https://leetcode.com/articles/power-of-three/

class Solution {

public boolean isPowerOfThree(int n) {

if (n <= 0){

return false;

}

while (n > 1){

if (n % 3 != 0){

return false;

}

n /= 3;

}

return true;

}

}

# 327\_Count.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array nums, return the number of range sums that lie in [lower, upper] inclusive.

Range sum S(i, j) is defined as the sum of the elements in nums between indices i and j (i ≤ j), inclusive.

Note:

A naive algorithm of O(n2) is trivial. You MUST do better than that.

Example:

Input: nums = [-2,5,-1], lower = -2, upper = 2,

Output: 3

Explanation: The three ranges are : [0,0], [2,2], [0,2] and their respective sums are: -2, -1, 2.

Method 1:

Time complexity: O(n^2)

class Solution {

public int countRangeSum(int[] nums, int lower, int upper) {

int count = 0;

for (int i = 0; i < nums.length; i++){

long sum = 0;

for (int j = i; j < nums.length; j++){

sum += nums[j];

if (sum < Integer.MIN\_VALUE || sum > Integer.MAX\_VALUE){//avoid overflow

break;

}

if (sum >= lower && sum <= upper){

count++;

}

}

}

return count;

}

}

Method 2:

public class Solution {

private class TreeNode {

long val = 0;

int count = 1;

int leftSize = 0;

int rightSize = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(long v) {

this.val = v;

this.count = 1;

this.leftSize = 0;

this.rightSize = 0;

}

}

private TreeNode insert(TreeNode root, long val) {

if(root == null) {

return new TreeNode(val);

} else if(root.val == val) {

root.count++;

} else if(val < root.val) {

root.leftSize++;

root.left = insert(root.left, val);

} else if(val > root.val) {

root.rightSize++;

root.right = insert(root.right, val);

}

return root;

}

private int countSmaller(TreeNode root, long val) {

if(root == null) {

return 0;

} else if(root.val == val) {

return root.leftSize;

} else if(root.val > val) {

return countSmaller(root.left, val);

} else {

return root.leftSize + root.count + countSmaller(root.right, val);

}

}

private int countLarger(TreeNode root, long val) {

if(root == null) {

return 0;

} else if(root.val == val) {

return root.rightSize;

} else if(root.val < val) {

return countLarger(root.right, val);

} else {

return countLarger(root.left, val) + root.count + root.rightSize;

}

}

private int rangeSize(TreeNode root, long lower, long upper) {

int total = root.count + root.leftSize + root.rightSize;

int smaller = countSmaller(root, lower); // Exclude everything smaller than lower

int larger = countLarger(root, upper); // Exclude everything larger than upper

return total - smaller - larger;

}

public int countRangeSum(int[] nums, int lower, int upper) {

if(nums.length == 0) {

return 0;

}

long[] sums = new long[nums.length + 1];

for(int i = 0; i < nums.length; i++) {

sums[i + 1] = sums[i] + nums[i];

}

TreeNode root = new TreeNode(sums[0]);

int output = 0;

for(int i = 1; i < sums.length; i++) {

output += rangeSize(root, sums[i] - upper, sums[i] - lower);

insert(root, sums[i]);

}

return output;

}

}

https://leetcode.com/problems/count-of-range-sum/discuss/78003/Java-BST-solution-averagely-O(nlogn)

# 329\_LongestIncreasingPath.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down.

You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

Example 1:

nums = [

[9,9,4],

[6,6,8],

[2,1,1]

]

Return 4

The longest increasing path is [1, 2, 6, 9].

Example 2:

nums = [

[3,4,5],

[3,2,6],

[2,2,1]

]

Return 4

The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.

To get max length of increasing sequences:

Do DFS from every cell

Compare every 4 direction and skip cells that are out of boundary or smaller

Get matrix max from every cell’s max

Use matrix[x][y] <= matrix[i][j] so we don’t need a visited[m][n] array

The key is to cache the distance because it’s highly possible to revisit a cell

class Solution {

public int longestIncreasingPath(int[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return 0;

}

int m = matrix.length;

int n = matrix[0].length;

int max = 1;

int[][] cache = new int[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

int countMax = dfsMaxLen(matrix, i, j, cache);

max = Math.max(max, countMax);

}

}

return max;

}

private int dfsMaxLen(int[][] matrix, int i, int j, int[][] cache){

if (cache[i][j] != 0){

return cache[i][j];

}

int m = matrix.length;

int n = matrix[0].length;

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

int max = 1;

for (int k = 0; k < dx.length; k++){

int cx = i + dx[k];

int cy = j + dy[k];

if (cx >= 0 && cx < m && cy >= 0 && cy < n && matrix[cx][cy] > matrix[i][j]){

int len = dfsMaxLen(matrix, cx, cy, cache);

max = Math.max(max, len + 1);

}

}

cache[i][j] = max;

return max;

}

}

Better version:

class Solution {

public int longestIncreasingPath(int[][] matrix) {

if (matrix == null || matrix.length == 0){

return 0;

}

int m = matrix.length;

int n = matrix[0].length;

int max = 1;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

int candi = dfs(matrix, i, j, dirs, map);

max = Math.max(max, candi);

}

}

return max;

}

private int dfs(int[][] matrix, int x, int y, int[][] dirs, Map<Integer, Integer> map){

int m = matrix.length;

int n = matrix[0].length;

int key = x \* n + y;

if (map.containsKey(key)){

return map.get(key);

}

int res = 1;

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && matrix[nx][ny] > matrix[x][y]){

res = Math.max(res, dfs(matrix, nx, ny, dirs, map) + 1);

}

}

map.put(key, res);

return res;

}

}

# 33\_SearchinRotatedSortedArray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Method 1:

class Solution {

public int search(int[] nums, int target) {

if (nums == null || nums.length == 0){

return -1;

}

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end -start) / 2;

if (nums[mid] == target){

return mid;

}

if (nums[mid] > nums[start]){

if (nums[start] <= target && target < nums[mid]){

end = mid;

}else{

start = mid;

}

}else{

if (nums[mid] < target && target <= nums[end]){

start = mid;

}else{

end = mid;

}

}

}

if (nums[start] == target){

return start;

}else if (nums[end] == target){

return end;

}else{

return -1;

}

}

}

Take this:

Best solution:

class Solution {

public int search(int[] nums, int target) {

if (nums == null || nums.length == 0){

return -1;

}

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return mid;

}else if (nums[mid] > nums[end]){

if (nums[mid] > target && target > nums[end]){

end = mid;

}else{

start = mid;

}

}else if (nums[mid] <= nums[end]){

if (nums[mid] < target && target <= nums[end]){ // note that must use <= , not <

start = mid;

}else{

end = mid;

}

}

}

if (nums[start] == target){

return start;

}else if (nums[end] == target){

return end;

}

return -1;

}

}

Method 2:

first find minimum using binary search instead of scanning

then check where the target is and use binary search

class Solution {

public int search(int[] nums, int target) {

if (nums == null || nums.length == 0){

return -1;

}

int start = 0;

int end = nums.length - 1;

int minIndex = findMiniumum(nums);

if (target == nums[start]){

return start;

}else if (target > nums[start]){

start = 0;

if (minIndex == 0){

end = nums.length - 1;

}else{

end = minIndex - 1;

}

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return mid;

}else if (nums[mid] < target){

start = mid;

}else{

end = mid;

}

}

if (nums[start] == target){

return start;

}

if (nums[end] == target){

return end;

}

}else{

start = minIndex;

end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return mid;

}else if (nums[mid] < target){

start = mid;

}else{

end = mid;

}

}

if (nums[start] == target){

return start;

}

if (nums[end] == target){

return end;

}

}

return -1;

}

private static int findMiniumum(int[] nums){

int start = 0;

int end = nums.length - 1;

int target = nums[end];

while (start + 1 < end){

int mid = start + (end - start) /2;

if (nums[mid] == target){

end = mid;

}else if (nums[mid] > target){

start = mid;

}else{

end = mid;

}

}

if (nums[start] <= target){

return start;

}

return end;

}

}

class Solution {

public int search(int[] nums, int target) {

if (nums == null || nums.length == 0){

return -1;

}

int start = 0;

int end = nums.length - 1;

int last = nums[end];

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return mid;

}

int last = nums[end];

if (target > last){

if (nums[mid] < target && nums[mid] > last){

start = mid;

}else{

end = mid;

}

}else{

if (nums[mid] > target && nums[mid] < last){

end = mid;

}else{

start = mid;

}

}

}

if (nums[start] == target){

return start;

}else if (nums[end] == target){

return end;

}

return -1;

}

}

# 334\_IncreasingTripletSubsequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.

Formally the function should:

Return true if there exists i, j, k

such that arr[i] < arr[j] < arr[k] given 0 ≤ i < j < k ≤ n-1 else return false.

Your algorithm should run in O(n) time complexity and O(1) space complexity.

Examples:

Given [1, 2, 3, 4, 5],

return true.

Given [5, 4, 3, 2, 1],

return false.

class Solution {

public boolean increasingTriplet(int[] nums) {

if (nums == null || nums.length < 3){

return false;

}

int firstMin = Integer.MAX\_VALUE;

int secondMin = Integer.MAX\_VALUE;

for (int i = 0; i < nums.length; i++){

if (nums[i] <= firstMin){

firstMin = nums[i];

}else if (nums[i] <= secondMin){

secondMin = nums[i];

}else{

return true;

}

}

return false;

}

}

class Solution {

public boolean increasingTriplet(int[] nums) {

int first = Integer.MAX\_VALUE;

int second = Integer.MAX\_VALUE;

for (int num : nums){

if (num < first){

first = num;

}else if (num != first && num < second){

second = num;

}else if (num != first && num != second){

return true;

}

}

return false;

}

}

# 338\_Counting.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non negative integer number num. For every numbers i in the range 0 ≤ i ≤ num calculate the

number of 1's in their binary representation and return them as an array.

Example:

For num = 5 you should return [0,1,1,2,1,2].

Follow up:

It is very easy to come up with a solution with run time O(n\*sizeof(integer)). But can you do it in

linear time O(n) /possibly in a single pass?

Space complexity should be O(n).

Can you do it like a boss? Do it without using any builtin function like \_\_builtin\_popcount in c++ or in any other language.

Method 1: O(n) DP

f[i] = f[i/2] + i % 2

i/2 == i >> 1

i%2 == i & 1

class Solution {

public int[] countBits(int num) {

int[] dp = new int[num+1];

for (int i = 1; i <= num; i++){

dp[i] = dp[i/2] + i%2;

}

return dp;

}

}

class Solution {

public int[] countBits(int num) {

int[] dp = new int[num+1];

for (int i = 1; i <= num; i++){

dp[i] = dp[i >> 1] + (i & 1);

}

return dp;

}

}

Method 2: TLE

class Solution {

public int[] countBits(int num) {

int[] result = new int[num + 1];

for (int i = 1; i <= num; i++){

int count = 0;

while (i != 0){

count++;

i = i & (i-1);

}

result[i] = count;

}

return result;

}

}

# 339\_Nested.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1:

Given the list [[1,1],2,[1,1]], return 10. (four 1's at depth 2, one 2 at depth 1)

Example 2:

Given the list [1,[4,[6]]], return 27. (one 1 at depth 1, one 4 at depth 2, and one 6 at depth 3; 1 + 4\*2 + 6\*3 = 27)

/\*\*

\* // This is the interface that allows for creating nested lists.

\* // You should not implement it, or speculate about its implementation

\* public interface NestedInteger {

\* // Constructor initializes an empty nested list.

\* public NestedInteger();

\*

\* // Constructor initializes a single integer.

\* public NestedInteger(int value);

\*

\* // @return true if this NestedInteger holds a single integer, rather than a nested list.

\* public boolean isInteger();

\*

\* // @return the single integer that this NestedInteger holds, if it holds a single integer

\* // Return null if this NestedInteger holds a nested list

\* public Integer getInteger();

\*

\* // Set this NestedInteger to hold a single integer.

\* public void setInteger(int value);

\*

\* // Set this NestedInteger to hold a nested list and adds a nested integer to it.

\* public void add(NestedInteger ni);

\*

\* // @return the nested list that this NestedInteger holds, if it holds a nested list

\* // Return null if this NestedInteger holds a single integer

\* public List<NestedInteger> getList();

\* }

\*/

Method: DFS

Time complexity: O(n)

class Solution {

public int depthSum(List<NestedInteger> nestedList) {

return dfs(nestedList, 1);

}

private int dfs(List<NestedInteger> nestedList, int level){

int sum = 0;

for (NestedInteger ni : nestedList){

if (ni.isInteger()){

sum += ni.getInteger() \* level;

}else{

sum += dfs(ni.getList(), level + 1);

}

}

return sum;

}

}

Method 2: BFS

class Solution{

public int depthSum(List<NestedInteger> nestedList){

Queue<NestedInteger> queue = new LinkedList<>();

int sum = 0;

for (NestedInteger ni : nestedList){

queue.offer(ni);

}

int level = 0;

while(!queue.isEmpty()){

int size = queue.size();

level++;

for (int i = 0; i < size; i++){

NestedInteger ni = queue.poll();

if (ni.isInteger()){

sum += level \* ni.getInteger();

}else{

for (NestedInteger i : ni.getList()){

queue.offer(i);

}

}

}

}

return sum;

}

}

# 34\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers nums sorted in ascending order, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of O(log n).

If the target is not found in the array, return [-1, -1].

Example 1:

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

Example 2:

Input: nums = [5,7,7,8,8,10], target = 6

Output: [-1,-1]

class Solution {

public int[] searchRange(int[] nums, int target) {

int[] res = new int[]{-1, -1};

if (nums == null || nums.length == 0){

return res;

}

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;;

if (nums[mid] >= target){

end = mid;

}else{

start = mid;

}

}

if (nums[start] == target){

res[0] = start;

}else if (nums[end] == target){

res[0] = end;

}

start = 0;

end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;;

if (nums[mid] <= target){

start = mid;

}else{

end = mid;

}

}

if (nums[end] == target){

res[1] = end;

}else if (nums[start] == target){

res[1] = start;

}

return res;

}

}

# 342\_Power.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.

Example:

Given num = 16, return true. Given num = 5, return false.

Follow up: Could you solve it without loops/recursion?

class Solution {

public boolean isPowerOfFour(int num) {

if (num <= 0){

return false;

}

while (num > 1){

if (num % 4 != 0){

return false;

}

num /= 4;

}

return true;

}

}

# 343\_Integer.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer n, break it into the sum of at least two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given n = 2, return 1 (2 = 1 + 1); given n = 10, return 36 (10 = 3 + 3 + 4).

Note: You may assume that n is not less than 2 and not larger than 58.

Method 1: O(n^2)

https://leetcode.com/problems/integer-break/discuss/80694/Java-DP-solution

class Solution {

public int integerBreak(int n) {

int[] dp = new int[n+1];

// 1.Init except dp[n], since it cannot be itself and must break into two positive

for (int i = 1; i < n; i++){

dp[i] = i; // note that in the result, itself could be the largest one in product.

}

for (int i = 2; i <= n; i++){

for (int j = 1; j <= i - j; j++){

dp[i] = Math.max(dp[i], dp[j] \* dp[i-j]);

}

}

return dp[n];

}

}

Method 2: O(n)

https://leetcode.com/problems/integer-break/discuss/80689/A-simple-explanation-of-the-math-part-and-a-O(n)-solution

class Solution {

public int integerBreak(int n) {

if (n== 2){

return 1;

}

if (n == 3){

return 2;

}

int ans = 1;

while (n > 4){

ans \*= 3;

n-= 3;

}

return ans \* n;

}

}

# 344\_ReverseString.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a function that takes a string as input and returns the string reversed.

Example:

Given s = "hello", return "olleh".

Method 1:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public String reverseString(String s) {

if (s == null || s.length() == 0){

return s;

}

char[] sc = s.toCharArray();

int start = 0;

int end = s.length() - 1;

while (start < end){

char temp = sc[start];

sc[start] = sc[end];

sc[end] = temp;

start++;

end--;

}

return String.valueOf(sc);

}

}

Method 2:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public String reverseString(String s) {

return new StringBuilder(s).reverse().toString();

}

}

Method 3:

public class Solution {

public String reverseString(String s) {

int length = s.length();

if (length <= 1) return s;

String leftStr = s.substring(0, length / 2);

String rightStr = s.substring(length / 2, length);

return reverseString(rightStr) + reverseString(leftStr);

}

}

Time complexity:

O(n log(n)) (Average Case) and O(n \* log(n)) (Worst Case) where n is the total number character in the

input string.

The recurrence equation is T(n) = 2 \* T(n/2) + O(n). O(n) is due to the fact that concatenation function

takes linear time.

The recurrence equation can be solved to get O(n \* log(n)).

Space complexity:

O(h) space is used where h is the depth of recursion tree generated which is log(n). Space is needed for

activation stack during recursion calls.

# 345\_Reverse.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a function that takes a string as input and reverse only the vowels of a string.

Example 1:

Given s = "hello", return "holle".

Example 2:

Given s = "leetcode", return "leotcede".

Note:

The vowels does not include the letter "y".

class Solution {

public String reverseVowels(String s) {

char[] charArray= s.toCharArray();

int start = 0;

int end = charArray.length - 1;

Set<Character> set = new HashSet<>(Arrays.asList('a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'));

while (start < end){

while (start < end && !set.contains(charArray[start])){

start++;

}

while (start < end && !set.contains(charArray[end])){

end--;

}

char temp = charArray[start];

charArray[start] = charArray[end];

charArray[end] = temp;

start++;

end--;

}

return String.valueOf(charArray);

}

}

# 346\_Moving.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

For example,

MovingAverage m = new MovingAverage(3);

m.next(1) = 1

m.next(10) = (1 + 10) / 2

m.next(3) = (1 + 10 + 3) / 3

m.next(5) = (10 + 3 + 5) / 3

Method 1:

Time complexity: O(n)

Space complexity: O(n)

class MovingAverage {

Queue<Integer> queue;

int size;

int count = 0;

/\*\* Initialize your data structure here. \*/

public MovingAverage(int size) {

queue = new LinkedList<>();

this.size = size;

}

public double next(int val) {

count++;

queue.offer(val);

double sum = 0;

if (count <= size){

for (int i = 0; i < count; i++){

int temp = queue.poll();

sum += temp;

queue.offer(temp);

}

}else{

queue.poll();

for (int i = 0; i < size; i++){

int temp = queue.poll();

sum += temp;

queue.offer(temp);

}

}

return sum / Math.min(count, size);

}

}

/\*\*

\* Your MovingAverage object will be instantiated and called as such:

\* MovingAverage obj = new MovingAverage(size);

\* double param\_1 = obj.next(val);

\*/

Method 2: Best solution

Time complexity: O(1)

Space complexity: O(n)

class MovingAverage {

Queue<Integer> queue;

int size;

int count = 0;

double sum = 0;

/\*\* Initialize your data structure here. \*/

public MovingAverage(int size) {

queue = new LinkedList<>();

this.size = size;

}

public double next(int val) {

count++;

queue.offer(val);

sum += val;

if (count > size){

sum -= queue.poll();

count--;

}

return sum / count;

}

}

/\*\*

\* Your MovingAverage object will be instantiated and called as such:

\* MovingAverage obj = new MovingAverage(size);

\* double param\_1 = obj.next(val);

\*/

# 347\_TopKFrequentElements.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty array of integers, return the k most frequent elements.

For example,

Given [1,1,1,2,2,3] and k = 2, return [1,2].

Note:

You may assume k is always valid, 1 ≤ k ≤ number of unique elements.

Your algorithm's time complexity must be better than O(n log n), where n is the array's size.

Method 1:

Time complexity: O(nlogk)

class Solution {

class Freq{

int val;

int freq;

public Freq(int val, int freq){

this.val = val;

this.freq = freq;

}

}

public List<Integer> topKFrequent(int[] nums, int k) {

List<Integer> result = new ArrayList<>();

if (nums == null || nums.length == 0){

return result;

}

Queue<Freq> minHeap = new PriorityQueue<Freq>(k, new Comparator<Freq>(){

public int compare(Freq a, Freq b){

return a.freq - b.freq;

}

});

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++){

map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);

}

for (Map.Entry<Integer, Integer> entry : map.entrySet()){

if (minHeap.size() < k){

minHeap.offer(new Freq(entry.getKey(), entry.getValue()));

}else{

if (entry.getValue() > minHeap.peek().freq){

minHeap.poll();

minHeap.offer(new Freq(entry.getKey(), entry.getValue()));

}

}

}

while (!minHeap.isEmpty()){

result.add(minHeap.poll().val);

}

return result;

}

}

Method 2:

Time complexity: O(nlogk)

class Solution {

class Freq{

int val;

int freq;

public Freq(int val, int freq){

this.val = val;

this.freq = freq;

}

}

public List<Integer> topKFrequent(int[] nums, int k) {

List<Integer> result = new ArrayList<>();

if (nums == null || nums.length == 0){

return result;

}

Queue<Freq> maxHeap = new PriorityQueue<Freq>(nums.length, new Comparator<Freq>(){

public int compare(Freq a, Freq b){

return b.freq - a.freq;

}

});

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++){

map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);

}

for (Map.Entry<Integer, Integer> entry : map.entrySet()){

maxHeap.offer(new Freq(entry.getKey(), entry.getValue()));

}

for (int i = 0; i < k; i++){

result.add(maxHeap.poll().val);

}

return result;

}

}

class Solution {

class Pair{

String str;

int freq;

public Pair(String str, int freq){

this.str = str;

this.freq = freq;

}

}

public List<String> topKFrequent(String[] words, int k) {

Queue<Pair> maxQueue = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare(Pair a, Pair b){

if (a.freq != b.freq){

return b.freq - a.freq;

}

return a.str.compareTo(b.str);

}

});

List<String> result = new ArrayList<>();

Map<String, Integer> map = new HashMap<>();

for (String word : words){

map.put(word, map.getOrDefault(word, 0) + 1);

}

for (String s : map.keySet()){

maxQueue.offer(new Pair(s, map.get(s)));

}

for (int i = 0; i < k; i++){

result.add(maxQueue.poll().str);

}

return result;

}

}

Better:

Time complexity: O(nlogk)

class Solution {

class Pair {

int val;

int freq;

public Pair (int val, int freq){

this.val = val;

this.freq = freq;

}

}

public List<Integer> topKFrequent(int[] nums, int k) {

List<Integer> res = new ArrayList<>();

Map<Integer, Integer> map = new HashMap<>();

for (int i : nums){

map.put(i, map.getOrDefault(i, 0) + 1);

}

Queue<Pair> pq = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p2.freq - p1.freq;

}

});

for (int key : map.keySet()){

pq.offer(new Pair(key, map.get(key)));

}

int i = 0;

while (!pq.isEmpty() && i < k){

Pair p = pq.poll();

res.add(p.val);

i++;

}

return res;

}

}

Best:

Time complexity: O(klogk)

class Solution {

class Pair {

int val;

int freq;

public Pair (int val, int freq){

this.val = val;

this.freq = freq;

}

}

public List<Integer> topKFrequent(int[] nums, int k) {

List<Integer> res = new ArrayList<>();

Map<Integer, Integer> map = new HashMap<>();

for (int i : nums){

map.put(i, map.getOrDefault(i, 0) + 1);

}

Queue<Pair> pq = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p1.freq - p2.freq;

}

});

for (int key : map.keySet()){

if (pq.size() < k){

pq.offer(new Pair(key, map.get(key)));

}else{

if (pq.peek().freq < map.get(key)){

pq.poll();

pq.offer(new Pair(key, map.get(key)));

}

}

}

Stack<Integer> stack = new Stack<>();

while (!pq.isEmpty()){

stack.push(pq.poll().val);

}

while (!stack.isEmpty()){

res.add(stack.pop());

}

return res;

}

}

# 35\_Search.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it

were inserted in order.

You may assume no duplicates in the array.

Example 1:

Input: [1,3,5,6], 5

Output: 2

Example 2:

Input: [1,3,5,6], 2

Output: 1

Example 3:

Input: [1,3,5,6], 7

Output: 4

Example 1:

Input: [1,3,5,6], 0

Output: 0

Check leetcode: https://leetcode.com/problems/random-pick-with-weight/discuss/154044/Java-accumulated-freq-sum-and-binary-search

https://github.com/optimisea/Leetcode/blob/master/Java/binary.java

Note that for this method, in most cases target should be between low (inclusive) and high (inclusive)

But there are two corner cases

case 1: target less than low, e.g. [1,2,3] target = 0 == > low will be 1, high will be 2

case 2: target greater than high, e.g. [1,2,3] target = 4 == > low will be 2, high will be 3

class Solution {

public int searchInsert(int[] nums, int target) {

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return mid;

}else if (nums[mid] < target){

start = mid;

}else{

end = mid;

}

}

if (nums[start] >= target){//cover corner case 1 and normal case

return start;

}

if (nums[end] >= target){//cover normal case

return end;

}

return end+1; //cover corner case 2

//return nums.length;

}

}

Better version: no need to check corner cases since it already covered.

class Solution {

public int searchInsert(int[] nums, int target) {

int start = 0;

int end = nums.length - 1;

while (start <= end){

int mid = start + (end - start) / 2;

if (target == nums[mid]){

return mid;

}else if (nums[mid] < target){

start = mid + 1;

}else{

end = mid - 1;

}

}

return start;

}

}

# 349\_Intersection.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two arrays, write a function to compute their intersection.

Example:

Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2].

Note:

Each element in the result must be unique.

The result can be in any order.

Best solution:

Method 1:

Time complexity: O(n)

class Solution {

public int[] intersection(int[] nums1, int[] nums2) {

Set<Integer> set = new HashSet<>();

Set<Integer> result = new HashSet<>();

for (int i = 0; i < nums1.length; i++){

if (!set.contains(nums1[i])){

set.add(nums1[i]);

}

}

for (int i = 0; i < nums2.length; i++){

if (set.contains(nums2[i])){

result.add(nums2[i]);

}

}

int[] ans = new int[result.size()];

int index = 0;

for (Integer i : result){

ans[index] = i;

index++;

}

return ans;

}

}

Method 2:

Time complexity: O(nlogn)

public int[] intersection(int[] nums1, int[] nums2) {

Set<Integer> set = new HashSet<>();

Arrays.sort(nums1);

Arrays.sort(nums2);

int i = 0;

int j = 0;

while (i < nums1.length && j < nums2.length) {

if (nums1[i] < nums2[j]) {

i++;

} else if (nums1[i] > nums2[j]) {

j++;

} else {

set.add(nums1[i]);

i++;

j++;

}

}

int[] result = new int[set.size()];

int k = 0;

for (Integer num : set) {

result[k++] = num;

}

return result;

}

Method 3:

Time complexity: O(nlogn)

class Solution {

public int[] intersection(int[] nums1, int[] nums2) {

if (nums1.length == 0){

return nums1;

}

if (nums2.length == 0){

return nums2;

}

Set<Integer> set = new HashSet<>();

Arrays.sort(nums1);

for (int i = 0; i < nums2.length; i++){

if (binarySearch(nums1, nums2[i])){

set.add(nums2[i]);

}

}

int[] ans = new int[set.size()];

int index = 0;

for (Integer i : set){

ans[index] = i;

index++;

}

return ans;

}

private boolean binarySearch(int[] A, int target){

int start = 0;

int end = A.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (A[mid] == target){

return true;

}else if (A[mid] > target){

end = mid;

}else{

start = mid;

}

}

if (A[start] == target || A[end] == target){

return true;

}

return false;

}

}

# 350\_IntersectionII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two arrays, write a function to compute their intersection.

Example:

Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2, 2].

Note:

Each element in the result should appear as many times as it shows in both arrays.

The result can be in any order.

Follow up:

What if the given array is already sorted? How would you optimize your algorithm?

What if nums1's size is small compared to nums2's size? Which algorithm is better?

What if elements of nums2 are stored on disk, and the memory is limited such that you

cannot load all elements into the memory at once?

The correction answer for the last question is to build index for the nums2 and for loop nums1 to use regional binary search

to find the num in nums2 on disk.

External Sort:http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250\_Weiss/L17-ExternalSortEX2.htm

From a data engineer’s perspective, basically there are three ideas to solve the question:

1. Store the two strings in distributed system(whether self designed or not), then

using MapReduce technique to solve the problem;

2. Processing the Strings by chunk, which fits the memory, then deal with each chunk of data at a time;

3. Processing the Strings by streaming, then check.

Method 1:

Time complexity: O(nlogn)

Space complexity: O(1)

class Solution {

public int[] intersect(int[] nums1, int[] nums2) {

List<Integer> result = new ArrayList<>();

Arrays.sort(nums1);

Arrays.sort(nums2);

int i = 0;

int j = 0;

while (i < nums1.length && j < nums2.length){

if (nums1[i] < nums2[j]){

i++;

}else if (nums1[i] > nums2[j]){

j++;

}else{

result.add(nums1[i]);

i++;

j++;

}

}

int[] ans = new int[result.size()];

int index = 0;

for (Integer n : result){

ans[index++] = n;

}

return ans;

}

}

Method 2:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int[] intersect(int[] nums1, int[] nums2) {

List<Integer> result = new ArrayList<>();

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums1.length; i++){

map.put(nums1[i], map.getOrDefault(nums1[i], 0) + 1);

}

for (int i = 0; i < nums2.length; i++){

if (map.containsKey(nums2[i]) && map.get(nums2[i]) > 0){

result.add(nums2[i]);

map.put(nums2[i], map.get(nums2[i]) - 1);

}

}

int[] ans = new int[result.size()];

int index = 0;

for (Integer n : result){

ans[index++] = n;

}

return ans;

}

}

# 351\_Android.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an Android 3x3 key lock screen and two integers m and n, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns o

f the Android lock screen, which consist of minimum of m keys and maximum n keys.

Rules for a valid pattern:

Each pattern must connect at least m keys and at most n keys.

All the keys must be distinct.

If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously

selected in the pattern. No jumps through non selected key is allowed.

The order of keys used matters.

Explanation:

| 1 | 2 | 3 |

| 4 | 5 | 6 |

| 7 | 8 | 9 |

Invalid move: 4 - 1 - 3 - 6

Line 1 - 3 passes through key 2 which had not been selected in the pattern.

Invalid move: 4 - 1 - 9 - 2

Line 1 - 9 passes through key 5 which had not been selected in the pattern.

Valid move: 2 - 4 - 1 - 3 - 6

Line 1 - 3 is valid because it passes through key 2, which had been selected in the pattern

Valid move: 6 - 5 - 4 - 1 - 9 - 2

Line 1 - 9 is valid because it passes through key 5, which had been selected in the pattern.

Example:

Given m = 1, n = 1, return 9.

The optimization idea is that 1,3,7,9 are symmetric, 2,4,6,8 are also symmetric. Hence we only calculate one among each group

and multiply by 4.

public class Solution {

// cur: the current position

// remain: the steps remaining

int DFS(boolean vis[], int[][] skip, int cur, int remain) {

if(remain < 0) return 0;

if(remain == 0) return 1;

vis[cur] = true;

int rst = 0;

for(int i = 1; i <= 9; ++i) {

// If vis[i] is not visited and (two numbers are adjacent or skip number is already visited)

if(!vis[i] && (skip[cur][i] == 0 || (vis[skip[cur][i]]))) {

rst += DFS(vis, skip, i, remain - 1);

}

}

vis[cur] = false;

return rst;

}

public int numberOfPatterns(int m, int n) {

// Skip array represents number to skip between two pairs

int skip[][] = new int[10][10];

skip[1][3] = skip[3][1] = 2;

skip[1][7] = skip[7][1] = 4;

skip[3][9] = skip[9][3] = 6;

skip[7][9] = skip[9][7] = 8;

skip[1][9] = skip[9][1] = skip[2][8] = skip[8][2] = skip[3][7] = skip[7][3] = skip[4][6] = skip[6][4] = 5;

boolean vis[] = new boolean[10];

int rst = 0;

// DFS search each length from m to n

for(int i = m; i <= n; ++i) {

rst += DFS(vis, skip, 1, i - 1) \* 4; // 1, 3, 7, 9 are symmetric

rst += DFS(vis, skip, 2, i - 1) \* 4; // 2, 4, 6, 8 are symmetric

rst += DFS(vis, skip, 5, i - 1); // 5

}

return rst;

}

}

# 354\_Russian.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

Example:

Given envelopes = [[5,4],[6,4],[6,7],[2,3]], the maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).

The same as Longest Increase Subsequence

Method 1: Dynamic Programming

class Solution {

public int maxEnvelopes(int[][] envelopes) {

Arrays.sort(envelopes, new Comparator<int[]>(){

public int compare (int[] e1, int[] e2){

if (e1[0] == e2[0]){

return e1[1] - e2[1];

}

return e1[0] - e2[0];

}

});

int n = envelopes.length;

int[] dp = new int[n];

for (int i = 0; i < n; i++){

dp[i] = 1;

}

for (int i = 1; i < n; i++){

for (int j = 0; j < i; j++){

if (envelopes[j][0] < envelopes[i][0] && envelopes[j][1] < envelopes[i][1]){

dp[i] = Math.max(dp[i], dp[j] + 1);

}

}

}

int ans = 0;

for (int i = 0; i < n; i++){

ans = Math.max(ans, dp[i]);

}

return ans;

}

}

Method 2: Binary Search

class Solution {

public int maxEnvelopes(int[][] envelopes) {

Arrays.sort(envelopes, new Comparator<int[]>(){

public int compare (int[] e1, int[] e2){

if (e1[0] == e2[0]){

return e2[1] - e1[1];

}

return e1[0] - e2[0];

}

});

int n = envelopes.length;

int len = 0;

int[] dp = new int[n];

for (int[] envelope : envelopes){

int index = Arrays.binarySearch(dp, 0, len, envelope[1]);

if (index < 0){

index = - (index+1);

}

dp[index] = envelope[1];

if (index == len){

len++;

}

}

return len;

}

}

# 356\_Line.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflect the given points.

Example 1:

Input: [[1,1],[-1,1]]

Output: true

Example 2:

Input: [[1,1],[-1,-1]]

Output: false

Follow up:

Could you do better than O(n2) ?

Note that (min + max) / 2 == (left + right) / 2 == > right = sum - left

class Solution {

public boolean isReflected(int[][] points) {

int max = Integer.MIN\_VALUE;

int min = Integer.MAX\_VALUE;

Set<String> set = new HashSet<>();

for (int[] point : points){

max = Math.max(max, point[0]);

min = Math.min(min, point[0]);

set.add(point[0] + "\_" + point[1]);

}

int sum = max + min;

for (int[] point : points){

if (!set.contains((sum - point[0]) + "\_" + point[1])){

return false;

}

}

return true;

}

}

# 357\_Count.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-negative integer n, count all numbers with unique digits, x, where 0 ≤ x < 10n.

Example:

Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding [11,22,33,44,55,66,77,88,99])

Following the hint. Let f(n) = count of number with unique digits of length n.

f(1) = 10. (0, 1, 2, 3, ...., 9)

f(2) = 9 \* 9. Because for each number i from 1, ..., 9, we can pick j to form a 2-digit number ij and there are 9 numbers that

are different from i for j to choose from.

f(3) = f(2) \* 8 = 9 \* 9 \* 8. Because for each number with unique digits of length 2, say ij, we can pick k to form a 3 digit number

ijk and there are 8 numbers that are different from i and j for k to choose from.

Similarly f(4) = f(3) \* 7 = 9 \* 9 \* 8 \* 7....

...

f(10) = 9 \* 9 \* 8 \* 7 \* 6 \* ... \* 1

f(11) = 0 = f(12) = f(13)....

any number with length > 10 couldn't be unique digits number.

The problem is asking for numbers from 0 to 10^n. Hence return f(1) + f(2) + .. + f(n)

class Solution {

public int countNumbersWithUniqueDigits(int n) {

if (n == 0){

return 1;

}

int ans = 10;

int unique = 9;

int available = 9;

while (n > 1 && available > 0){

unique \*= available;

ans += unique;

available--;

n--;

}

return ans;

}

}

# 358\_Rearrange.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty string s and an integer k, rearrange the string such that the same characters are at least distance k from each other.

All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string "".

Example 1:

Input: s = "aabbcc", k = 3

Output: "abcabc"

Explanation: The same letters are at least distance 3 from each other.

Example 2:

Input: s = "aaabc", k = 3

Output: ""

Explanation: It is not possible to rearrange the string.

Example 3:

Input: s = "aaadbbcc", k = 2

Output: "abacabcd"

Explanation: The same letters are at least distance 2 from each other.

The same as Reorganize String

https://github.com/optimisea/Leetcode/blob/master/Java/767\_Reorganize.java

Method 1: PQ + Greedy

Time complexity: O(N\*logN) N is the s.length()

class Solution {

class Pair {

char ch;

int num;

public Pair (char ch, int num){

this.ch = ch;

this.num = num;

}

}

public String rearrangeString(String s, int k) {

if (k == 0){

return s;

}

Map<Character, Integer> map = new HashMap<>();

for (char c : s.toCharArray()){

map.put(c, map.getOrDefault(c, 0) + 1);

}

Queue<Pair> maxPQ = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p2.num - p1.num;

}

});

for (char c : map.keySet()){

maxPQ.offer(new Pair(c, map.get(c)));

}

StringBuilder sb = new StringBuilder();

int[] hash = new int[26];

while (!maxPQ.isEmpty()){

Pair curr = maxPQ.poll();

char c = curr.ch;

int num = curr.num;

if (sb.length() >= k){

hash[sb.charAt(sb.length()-k) - 'a']--;

}

if (hash[c - 'a'] == 0){

hash[c - 'a']++;

sb.append(c);

num--;

if (num > 0){

maxPQ.offer(new Pair(c, num));

}

}else{

Set<Pair> tmp = new HashSet<>();

tmp.add(curr);

while (!maxPQ.isEmpty()){

Pair p = maxPQ.poll();

if (hash[p.ch - 'a'] == 0){

hash[p.ch - 'a']++;

sb.append(p.ch);

p.num--;

if (p.num > 0){

maxPQ.offer(new Pair(p.ch, p.num));

}

for (Pair t : tmp){

maxPQ.offer(new Pair(t.ch, t.num));

}

break;

}else{

tmp.add(p);

}

}

if (maxPQ.isEmpty()){

return "";

}

}

}

return sb.toString();

}

}

Method 2: Better version

class Solution {

class Pair {

char ch;

int num;

public Pair (char ch, int num){

this.ch = ch;

this.num = num;

}

}

public String rearrangeString(String s, int k) {

if (k == 0){

return s;

}

Map<Character, Integer> map = new HashMap<>();

for (char c : s.toCharArray()){

map.put(c, map.getOrDefault(c, 0) + 1);

}

Queue<Pair> maxPQ = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p2.num - p1.num;

}

});

for (char c : map.keySet()){

maxPQ.offer(new Pair(c, map.get(c)));

}

StringBuilder sb = new StringBuilder();

Queue<Pair> waitQ = new LinkedList<>();

while (!maxPQ.isEmpty()){

Pair curr = maxPQ.poll();

char c = curr.ch;

int num = curr.num;

sb.append(c);

waitQ.offer(new Pair(c, num - 1));

if (waitQ.size() < k){

continue;

}

Pair p = waitQ.poll();

if (p.num > 0){

maxPQ.offer(new Pair(p.ch, p.num));

}

}

return sb.length() == s.length() ? sb.toString() : "";

}

}

# 359\_DesingLoggerSystem.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is

not printed in the last 10 seconds.

Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp,

otherwise returns false.

It is possible that several messages arrive roughly at the same time.

Example:

Logger logger = new Logger();

// logging string "foo" at timestamp 1

logger.shouldPrintMessage(1, "foo"); returns true;

// logging string "bar" at timestamp 2

logger.shouldPrintMessage(2,"bar"); returns true;

// logging string "foo" at timestamp 3

logger.shouldPrintMessage(3,"foo"); returns false;

// logging string "bar" at timestamp 8

logger.shouldPrintMessage(8,"bar"); returns false;

// logging string "foo" at timestamp 10

logger.shouldPrintMessage(10,"foo"); returns false;

// logging string "foo" at timestamp 11

logger.shouldPrintMessage(11,"foo"); returns true;

Method 1: for small data

class Logger {

Map<String, Integer> map;

/\*\* Initialize your data structure here. \*/

public Logger() {

map = new HashMap<String, Integer>();

}

/\*\* Returns true if the message should be printed in the given timestamp, otherwise returns false.

If this method returns false, the message will not be printed.

The timestamp is in seconds granularity. \*/

public boolean shouldPrintMessage(int timestamp, String message) {

if (!map.containsKey(message) || timestamp - map.get(message) >= 10){

map.put(message, timestamp);

return true;

}

return false;

}

}

Method 2: for large data

class Logger {

private int N;

private int[] times;

private String[] strs;

/\*\* Initialize your data structure here. \*/

public Logger() {

N = 10;

times = new int[N];

strs = new String[N];

}

/\*\* Returns true if the message should be printed in the given timestamp, otherwise returns false.

If this method returns false, the message will not be printed.

The timestamp is in seconds granularity. \*/

public boolean shouldPrintMessage(int timestamp, String message) {

int index = timestamp % 10;

for (int i = 0; i < N; i++){

if (strs[i].equals(message)){

if (timstamp - time[i] <= 10){

return false;

}

strs[i] = null;

time[i] = 0;

}

}

strs[index] = message;

time[index] = timstamp;

return true;

}

}

/\*\*

\* Your Logger object will be instantiated and called as such:

\* Logger obj = new Logger();

\* boolean param\_1 = obj.shouldPrintMessage(timestamp,message);

\*/

# 36\_ValidSudoku.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules.

The Sudoku board could be partially filled, where empty cells are filled with the character '.'.

A partially filled sudoku which is valid.

Note:

A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

Time complexity: O(N \* N)

class Solution {

public boolean isValidSudoku(char[][] board) {

for (int i = 0; i < 9; i++){

if (!isValid(board, i, 0, i, 8)){

return false;

}

if (!isValid(board, 0, i, 8, i)){

return false;

}

}

for (int i = 0; i < 3; i++){

for (int j = 0; j < 3; j++){

if (!isValid(board, 3\*i, 3\*j, 3\*i+2, 3\*j+2)){

return false;

}

}

}

return true;

}

private boolean isValid(char[][] board, int x1, int y1, int x2, int y2){

Set<Character> set = new HashSet<>();

for (int i = x1; i <= x2; i++){

for (int j = y1; j <= y2; j++){

if (board[i][j] != '.'){

if (!set.add(board[i][j])){

return false;

}

}

}

}

return true;

}

}

# 360\_Sort.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted array of integers nums and integer values a, b and c. Apply a quadratic function of the form f(x) = ax2 + bx + c to

each element x in the array.

The returned array must be in sorted order.

Expected time complexity: O(n)

Example:

nums = [-4, -2, 2, 4], a = 1, b = 3, c = 5,

Result: [3, 9, 15, 33]

nums = [-4, -2, 2, 4], a = -1, b = 3, c = 5

Result: [-23, -5, 1, 7]

class Solution {

public int[] sortTransformedArray(int[] nums, int a, int b, int c) {

int n = nums.length;

int[] result = new int[n];

int left = 0;

int right = n - 1;

int index = a >= 0 ? n - 1: 0;

while (left <= right){

int leftResult = calculate(nums[left], a, b, c);

int rightResult = calculate(nums[right], a, b, c);

if (a >= 0){

if (leftResult >= rightResult){

result[index--] = leftResult;

left++;

}else{

result[index--] = rightResult;

right--;

}

}else{

if (leftResult >= rightResult){

result[index++] = rightResult;

right--;

}else{

result[index++] = leftResult;

left++;

}

}

}

return result;

}

private int calculate(int num, int a, int b, int c){

return a\*num\*num + b\*num + c;

}

}

# 361\_Bomb.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2D grid, each cell is either a wall 'W', an enemy 'E' or empty '0' (the number zero), return the maximum enemies you can kill

using one bomb.

The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong

to be destroyed.

Note: You can only put the bomb at an empty cell.

Example:

Input: [["0","E","0","0"],["E","0","W","E"],["0","E","0","0"]]

Output: 3

Explanation: For the given grid,

0 E 0 0

E 0 W E

0 E 0 0

Placing a bomb at (1,1) kills 3 enemies.

Method 1: Brute Force

Time compleixty: O(mn \* (m+n))

class Solution {

public int maxKilledEnemies(char[][] grid) {

if (grid == null || grid.length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

int res = 0;

for (int i = 0; i < m; ++i){

for (int j = 0; j < n; ++j){

if (grid[i][j] == '0'){

int num = count(grid, i, j);

res = Math.max(res, num);

}

}

}

return res;

}

private int count(char[][] grid, int row, int col){

int res = 0;

int m = grid.length;

int n = grid[0].length;

int i = row;

while (i >= 0){

if (grid[i][col] == 'E'){

res++;

}else if (grid[i][col] == 'W'){

break;

}

i--;

}

i = row;

while (i < m){

if (grid[i][col] == 'E'){

res++;

}else if (grid[i][col] == 'W'){

break;

}

i++;

}

int j = col;

while (j >= 0){

if (grid[row][j] == 'E'){

res++;

}else if (grid[row][j] == 'W'){

break;

}

j--;

}

j = col;

while (j < n){

if (grid[row][j] == 'E'){

res++;

}else if (grid[row][j] == 'W'){

break;

}

j++;

}

return res;

}

}

Method 2:

Time complexity: O(mn \* factor)), worst case could be O(mn \* (m or n))

Space complexity: O(n)

http://www.cnblogs.com/grandyang/p/5599289.html

public int maxKilledEnemies(char[][] grid){

int res = 0;

int m = grid.length;

int n = grid[0].length;

int rowKills = 0;

int[] colKills = new int[n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (j == 0 || grid[i][j-1] == 'W'){

rowKills = 0;

for (int k = j; j < n && grid[i][k] != 'W'; k++){

rowKills += grid[i][k] == 'E';

}

}

if (i == 0 || grid[i-1][j] == 'W'){

colKills[j] = 0;

for (int k = i; k < m && grid[k][j] != 'W'; k++){

colKills[j] += grid[k][j] == 'E';

}

}

if (grid[i][j] == '0'){

res = Math.max(res, rowKills + colKills[j]);

}

}

}

return res;

}

Method 3: Best solution

Time complexty: O(mn)

Space complexity: O(mn)

http://www.cnblogs.com/grandyang/p/5599289.html

preSum to calculate left, right, up, down first

public int maxKilledEnemies(char[][] grid){

int m = grid.length;

int n = grid[0].length;

int[][] left = new int[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (j == 0 || grid[i][j]=='W'){

left[i][j] = 0;

}else{

left[i][j] = left[i][j-1] + (grid[i][j] == 'E');

}

}

}

int[][] right = new int[m][n];

for (int i = 0; i < m; i++){

for (int j = n-1; j >= 0; j--){

if (j == n-1 || grid[i][j] == 'W'){

right[i][j] = 0;

}else{

right[i][j] = right[i][j+1] + (grid[i][j] == 'E');

}

}

}

int[][] up = new int[m][n];

for (int j = 0; j < n; j++){

for (int i = 0; i < m; i++){

if (i == 0 || grid[i][j] == 'W'){

up[i][j] = 0;

}else{

up[i][j] = up[i-1][j] + (grid[i][j] == 'E');

}

}

}

int[][] down = new int[m][n];

for (int j = 0; j < n; j++){

for (int i = m-1; i>=0; i--){

if (i == 0 || grid[i][j] == 'W'){

down[i][j] = 0;

}else{

down[i][j] = down[i+1][j] + (grid[i][j] == 'E');

}

}

}

int res = 0;

for (int i = 0; i < m; i++){

for (int j = 0;j < n; j++){

if (grid[i][j] == '0'){

res = Math.max(res, left[i][j] + right[i][j] + up[i][j] + down[i][j]);

}

}

}

return res;

}

# 362\_SlidingWindow.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of n integer with duplicate number, and a moving window(size k),

move the window at each iteration from the start of the array, find the maximum number inside the window at each moving.

Have you met this question in a real interview?

Example

For array [1, 2, 7, 7, 8], moving window size k = 3. return [7, 7, 8]

At first the window is at the start of the array like this

[|1, 2, 7| ,7, 8] , return the maximum 7;

then the window move one step forward.

[1, |2, 7 ,7|, 8], return the maximum 7;

then the window move one step forward again.

[1, 2, |7, 7, 8|], return the maximum 8;

Challenge

o(n) time and O(k) memory

public class Solution {

/\*

\* @param nums: A list of integers

\* @param k: An integer

\* @return: The maximum number inside the window at each moving

\*/

public ArrayList<Integer> maxSlidingWindow(int[] nums, int k) {

ArrayList<Integer> result = new ArrayList<>();

Deque<Integer> deque = new ArrayDeque<>();

if (nums == null || nums.length == 0){

return result;

}

for (int i = 0; i < k - 1; i++){

inDeque(deque, i, nums);

}

for (int i = k - 1; i < nums.length; i++){

inDeque(deque, i, nums);

result.add(nums[deque.peekFirst()]);

outDeque(deque, i-k+1, nums);

}

return result;

}

private void inDeque(Deque<Integer> deque, int i, int[] nums){

while (!deque.isEmpty() && nums[i] > nums[deque.peekLast()]){

deque.pollLast();

}

deque.offerLast(i);

}

private void outDeque(Deque<Integer> deque, int i, int[] nums){

if (nums[i] == nums[deque.peekFirst()]){

deque.pollFirst();

}

}

};

# 362\_SlidingWindowMaximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of n integer with duplicate number, and a moving window(size k),

move the window at each iteration from the start of the array, find the maximum number inside the window at each moving.

Have you met this question in a real interview?

Example

For array [1, 2, 7, 7, 8], moving window size k = 3. return [7, 7, 8]

At first the window is at the start of the array like this

[|1, 2, 7| ,7, 8] , return the maximum 7;

then the window move one step forward.

[1, |2, 7 ,7|, 8], return the maximum 7;

then the window move one step forward again.

[1, 2, |7, 7, 8|], return the maximum 8;

Challenge

o(n) time and O(k) memory

https://www.jiuzhang.com/solution/sliding-window-maximum/

public class Solution {

/\*

\* @param nums: A list of integers

\* @param k: An integer

\* @return: The maximum number inside the window at each moving

\*/

public ArrayList<Integer> maxSlidingWindow(int[] nums, int k) {

ArrayList<Integer> result = new ArrayList<>();

Deque<Integer> deque = new ArrayDeque<>();

if (nums == null || nums.length == 0){

return result;

}

for (int i = 0; i < k - 1; i++){

inDeque(deque, i, nums);

}

for (int i = k - 1; i < nums.length; i++){

inDeque(deque, i, nums);

result.add(nums[deque.peekFirst()]);

outDeque(deque, i-k+1, nums);

}

return result;

}

private void inDeque(Deque<Integer> deque, int i, int[] nums){

while (!deque.isEmpty() && nums[i] > nums[deque.peekLast()]){

deque.pollLast();

}

deque.offerLast(i);

}

private void outDeque(Deque<Integer> deque, int i, int[] nums){

if (nums[i] == nums[deque.peekFirst()]){

deque.pollFirst();

}

}

};

Method 2:

class Solution {

public int[] maxSlidingWindow(int[] nums, int k) {

if (nums == null || nums.length == 0 || k == 0){

return new int[0];

}

int n = nums.length;

int[] maxLeft = new int[n];

int[] maxRight = new int[n];

maxLeft[0] = nums[0];

maxRight[n-1] = nums[n-1];

for (int i = 1; i < nums.length; i++){

maxLeft[i] = i % k == 0 ? nums[i] : Math.max(maxLeft[i-1], nums[i]);

int j = n - 1 - i;

maxRight[j] = j % k == 0 ? nums[j] : Math.max(maxRight[j+1], nums[j]);

}

int[] res = new int[n-k+1];

for (int i = 0; i < n-k+1; i++){

res[i] = Math.max(maxRight[i], maxLeft[i+k-1]);

}

return res;

}

}

# 363\_Max Sum of Rectangle No Larger Than K.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty 2D matrix matrix and an integer k, find the max sum of a rectangle in the matrix such that its sum is no larger than k.

Example:

Given matrix = [

[1, 0, 1],

[0, -2, 3]

]

k = 2

The answer is 2. Because the sum of rectangle [[0, 1], [-2, 3]] is 2 and 2 is the max number no larger than k (k = 2).

Note:

The rectangle inside the matrix must have an area > 0.

What if the number of rows is much larger than the number of columns?

Method 1: Use TreeSet for binary search

https://leetcode.com/problems/max-sum-of-rectangle-no-larger-than-k/discuss/83597/Java-Binary-Search-solution-time-complexity-min(mn)2\*max(mn)\*log(max(mn))

HashMap may not be very good since the problem is asking for the rectangular area no larger than K...

From map we could only get the exact value of (val - target).

Not like 560 Subarray Sum Equals K.java, which could use HashMap

first consider the situation matrix is 1D

we can save every sum of 0~i(0<=i<len) and binary search previous sum to find

possible result for every index, time complexity is O(NlogN).

so in 2D matrix, we can sum up all values from row i to row j and create a 1D array

to use 1D array solution.

If col number is less than row number, we can sum up all values from col i to col j

then use 1D array solution.

Time complexity: O(min(row, col)^2 \* max(row, col)\*log(max(row, col)))

Space complexity: O(max(row, col)

class Solution {

public int maxSumSubmatrix(int[][] matrix, int k) {

int row = matrix.length;

int col = matrix[0].length;

int m = Math.min(row, col);

int n = Math.max(row, col);

boolean colLarge = col > row;

int res = Integer.MIN\_VALUE;

for (int i = 0; i < m; i++){

int[] arr = new int[n];

for (int j = i; j >= 0; j--){

int preSum = 0;

TreeSet<Integer> set = new TreeSet<>();

set.add(0);// 0 means the sum equals k

for (int c = 0; c < n; c++){

arr[c] = arr[c] + (colLarge == true ? matrix[j][c] : matrix[c][j]);

preSum += arr[c];

Integer Int = set.ceiling(preSum - k);

if (Int != null){

res = Math.max(res, preSum - Int);

}

set.add(preSum);

}

}

}

return res;

}

}

# 364\_[Nested List Weight Sum II](https://leetcode.com/problems/nested-list-weight-sum-ii).java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Different from the previous question where weight is increasing from root to leaf, now the weight is defined from bottom up.

i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

Example 1:

Given the list [[1,1],2,[1,1]], return 8. (four 1's at depth 1, one 2 at depth 2)

Example 2:

Given the list [1,[4,[6]]], return 17. (one 1 at depth 3, one 4 at depth 2, and one 6 at depth 1; 1\*3 + 4\*2 + 6\*1 = 17)

/\*\*

\* // This is the interface that allows for creating nested lists.

\* // You should not implement it, or speculate about its implementation

\* public interface NestedInteger {

\* // Constructor initializes an empty nested list.

\* public NestedInteger();

\*

\* // Constructor initializes a single integer.

\* public NestedInteger(int value);

\*

\* // @return true if this NestedInteger holds a single integer, rather than a nested list.

\* public boolean isInteger();

\*

\* // @return the single integer that this NestedInteger holds, if it holds a single integer

\* // Return null if this NestedInteger holds a nested list

\* public Integer getInteger();

\*

\* // Set this NestedInteger to hold a single integer.

\* public void setInteger(int value);

\*

\* // Set this NestedInteger to hold a nested list and adds a nested integer to it.

\* public void add(NestedInteger ni);

\*

\* // @return the nested list that this NestedInteger holds, if it holds a nested list

\* // Return null if this NestedInteger holds a single integer

\* public List<NestedInteger> getList();

\* }

\*/

Method 1: Two pass DFS

class Solution {

public int depthSumInverse(List<NestedInteger> nestedList) {

int depth = getDepth(nestedList);

return dfs(nestedList, depth);

}

private int dfs(List<NestedInteger> nestedList, int depth){

int sum = 0;

for (NestedInteger ni : nestedList){

if (ni.isInteger()){

sum += ni.getInteger() \* depth;

}else{

sum += dfs(ni.getList(), depth - 1);

}

}

return sum;

}

private int getDepth(List<NestedInteger> nestedList){

int max = 0;

for (NestedInteger ni : nestedList){

if (!ni.isInteger()){

int depth = getDepth(ni.getList());

max = Math.max(max, depth);

}

}

return max + 1;

}

}

Method 2: BFS

/\*\*

\* // This is the interface that allows for creating nested lists.

\* // You should not implement it, or speculate about its implementation

\* public interface NestedInteger {

\* // Constructor initializes an empty nested list.

\* public NestedInteger();

\*

\* // Constructor initializes a single integer.

\* public NestedInteger(int value);

\*

\* // @return true if this NestedInteger holds a single integer, rather than a nested list.

\* public boolean isInteger();

\*

\* // @return the single integer that this NestedInteger holds, if it holds a single integer

\* // Return null if this NestedInteger holds a nested list

\* public Integer getInteger();

\*

\* // Set this NestedInteger to hold a single integer.

\* public void setInteger(int value);

\*

\* // Set this NestedInteger to hold a nested list and adds a nested integer to it.

\* public void add(NestedInteger ni);

\*

\* // @return the nested list that this NestedInteger holds, if it holds a nested list

\* // Return null if this NestedInteger holds a single integer

\* public List<NestedInteger> getList();

\* }

\*/

class Solution {

public int depthSumInverse(List<NestedInteger> nestedList) {

Queue<NestedInteger> queue = new LinkedList<>();

int sum = 0;

int prevSum = 0;

for (NestedInteger ni : nestedList){

queue.offer(ni);

}

while (!queue.isEmpty()){

int size = queue.size();

int currSum = 0;

for (int i = 0; i < size; i++){

NestedInteger ni = queue.poll();

if (ni.isInteger()){

currSum += ni.getInteger();

}else{

for (NestedInteger j : ni.getList()){

queue.offer(j);

}

}

}

prevSum += currSum;

sum += prevSum;

}

return sum;

}

}

/\*\*

\* // This is the interface that allows for creating nested lists.

\* // You should not implement it, or speculate about its implementation

\* public interface NestedInteger {

\* // Constructor initializes an empty nested list.

\* public NestedInteger();

\*

\* // Constructor initializes a single integer.

\* public NestedInteger(int value);

\*

\* // @return true if this NestedInteger holds a single integer, rather than a nested list.

\* public boolean isInteger();

\*

\* // @return the single integer that this NestedInteger holds, if it holds a single integer

\* // Return null if this NestedInteger holds a nested list

\* public Integer getInteger();

\*

\* // Set this NestedInteger to hold a single integer.

\* public void setInteger(int value);

\*

\* // Set this NestedInteger to hold a nested list and adds a nested integer to it.

\* public void add(NestedInteger ni);

\*

\* // @return the nested list that this NestedInteger holds, if it holds a nested list

\* // Return null if this NestedInteger holds a single integer

\* public List<NestedInteger> getList();

\* }

\*/

class Solution {

public int depthSumInverse(List<NestedInteger> nestedList) {

Queue<NestedInteger> queue = new LinkedList<>();

int sum = 0;

int prevSum = 0;

for (NestedInteger ni : nestedList){

queue.offer(ni);

}

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

NestedInteger ni = queue.poll();

if (ni.isInteger()){

prevSum += ni.getInteger();

}else{

for (NestedInteger j : ni.getList()){

queue.offer(j);

}

}

}

sum += prevSum;

}

return sum;

}

}

# 364\_TrappingRainWaterII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n x m non-negative integers representing an elevation map 2d where the area of each cell is 1 x 1,

compute how much water it is able to trap after raining.

Given 5\*4 matrix

[12,13,0,12]

[13,4,13,12]

[13,8,10,12]

[12,13,12,12]

[13,13,13,13]

return 14.

Method: PriorityQueue + BFS

Advanced Algorithm jiuzhang

public class Solution {

/\*\*

\* @param heights: a matrix of integers

\* @return: an integer

\*/

class Pair{

int x;

int y;

int h;

public Pair(int x, int y, int h){

this.x = x;

this.y = y;

this.h = h;

}

}

public int trapRainWater(int[][] heights) {

Queue<Pair> q = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare(Pair a, Pair b){

return a.h - b.h;

}

});

int ans = 0;

int m = heights.length;

int n = heights[0].length;

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

q.offer(new Pair(i, 0, heights[i][0]));

q.offer(new Pair(i, n-1, heights[i][n-1]));

visited[i][0] = true;

visited[i][n-1] = true;

}

for (int j = 0; j < n; j++){

q.offer(new Pair(0, j, heights[0][j]));

q.offer(new Pair(m-1, j, heights[m-1][j]));

visited[0][j] = true;

visited[m-1][j] = true;

}

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

while (!q.isEmpty()){

Pair curr = q.poll();

for (int i = 0; i < dx.length; i++){

int x = curr.x + dx[i];

int y = curr.y + dy[i];

if (x >= 0 && x < m && y >= 0 && y < n && !visited[x][y]){

visited[x][y] = true;

ans += Math.max(0, curr.h - heights[x][y]);

q.offer(new Pair(x, y, Math.max(curr.h, heights[x][y])));

}

}

}

return ans;

}

}

Best solution:

class Solution {

class Node {

int x;

int y;

int h;

public Node (int x, int y, int h){

this.x = x;

this.y = y;

this.h = h;

}

}

public int trapRainWater(int[][] heightMap) {

if (heightMap == null || heightMap.length == 0){

return 0;

}

int m = heightMap.length;

int n = heightMap[0].length;

Queue<Node> pq = new PriorityQueue<Node>(new Comparator<Node>(){

public int compare (Node n1, Node n2){

return n1.h - n2.h;

}

});

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

pq.offer(new Node(i, 0, heightMap[i][0]));

visited[i][0] = true;

pq.offer(new Node(i, n-1, heightMap[i][n-1]));

visited[i][n-1] = true;

}

for (int j = 1; j < n-1; j++){

pq.offer(new Node(0, j, heightMap[0][j]));

visited[0][j] = true;

pq.offer(new Node(m-1, j, heightMap[m-1][j]));

visited[m-1][j] = true;

}

int[][] dirs = {{1,0}, {0,1}, {-1,0}, {0,-1}};

int res = 0;

while (!pq.isEmpty()){

Node node = pq.poll();

for (int[] dir : dirs){

int nx = node.x + dir[0];

int ny = node.y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

if (heightMap[nx][ny] < node.h){

res += node.h - heightMap[nx][ny];

//note that this node must be pushed in queue and set height as node.h not heightMap[nx][ny]

//because the water will be filled up to the level of node.h not heightMap[nx][ny]

pq.offer(new Node(nx, ny, node.h));

}else{

pq.offer(new Node(nx, ny, heightMap[nx][ny]));

}

visited[nx][ny] = true;

}

}

}

return res;

}

}

# 365\_Water.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given two jugs with capacities x and y litres. There is an infinite amount of water supply available. You need to determine whether it is possible to measure exactly z litres using these two jugs.

If z liters of water is measurable, you must have z liters of water contained within one or both buckets by the end.

Operations allowed:

Fill any of the jugs completely with water.

Empty any of the jugs.

Pour water from one jug into another till the other jug is completely full or the first jug itself is empty.

Example 1: (From the famous "Die Hard" example)

Input: x = 3, y = 5, z = 4

Output: True

Example 2:

Input: x = 2, y = 6, z = 5

Output: False

https://leetcode.com/problems/water-and-jug-problem/discuss/83715/Math-solution-Java-solution

class Solution {

public boolean canMeasureWater(int x, int y, int z) {

if (x + y < z){

return false;

}

if (x == z || y == z || x +y == z){

return true;

}

return z % gcd(x, y) == 0;

}

private int gcd (int a, int b){

if (b == 0){

return a;

}

return gcd(b, a%b);

}

}

366\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, collect a tree's nodes as if you were doing this: Collect and remove all leaves, repeat until the tree is empty.

Example:

Given binary tree

1

/ \

2 3

/ \

4 5

Returns [4, 5, 3], [2], [1].

Explanation:

1. Removing the leaves [4, 5, 3] would result in this tree:

1

/

2

2. Now removing the leaf [2] would result in this tree:

1

3. Now removing the leaf [1] would result in the empty tree:

[]

Returns [4, 5, 3], [2], [1].

Method 1:

Time complexity: O(n^2)

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<Integer>> findLeaves(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

while (root != null){

List<Integer> item = new ArrayList<>();

root = dfs(item, root);

result.add(item);

}

return result;

}

private TreeNode dfs(List<Integer> item, TreeNode root){

if (root == null){

return root;

}

if (root.left == null && root.right == null){

item.add(root.val);

root = null;

return null;

}

root.left = dfs(item, root.left);

root.right = dfs(item, root.right);

return root;

}

}

The following method is wrong because:

Java does manipulate objects by reference, and all object variables are references.

However, Java doesn't pass method arguments by reference; it passes them by value.

Java copies and passes the reference by value, not the object.

Hence, your just modified the copy of root and set it to null instead of the original root. So your code will run forever and TLE.

The workaround is to return the root so it can be modified as shown in method 1

WRONG SOLUTION!!!

class Solution {

public List<List<Integer>> findLeaves(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

while (root != null){

List<Integer> item = new ArrayList<>();

dfs(item, root);

result.add(item);

}

return result;

}

private void dfs(List<Integer> item, TreeNode root){

if (root == null){

return;

}

if (root.left == null && root.right == null){

item.add(root.val);

root = null;

return;

}

dfs(item, root.left);

dfs(item, root.right);

}

}

Method 2:

Time complexity: O(n)

bottom-up approach. The key is to find the height of each node. Here the definition of height is:

The height of a node is the number of edges from the node to the deepest leaf. --CMU 15-121 Binary Trees

According to the definition, the height of leaf is 0. h(node) = 1 + max(h(node.left), h(node.right)).

The height of a node is also the its index in the result list (res). For example, leaves, whose heights are 0, are stored in res[0].

Once we find the height of a node, we can put it directly into the result.

class Solution {

public List<List<Integer>> findLeaves(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

height(result, root);

return result;

}

private int height(List<List<Integer>> result, TreeNode root){

if (root == null){

return -1;

}

int left = height(result, root.left);

int right = height(result, root.right);

int currHeight = 1 + Math.max(left, right);

if (currHeight == result.size()){

result.add(new ArrayList<Integer>());

}

result.get(currHeight).add(root.val);

root = null;

return currHeight;

}

}

Best solution:

https://github.com/optimisea/Leetcode/blob/master/Java/650\_BinaryTreeLeavesOrderTraversal.java

class Solution {

public List<List<Integer>> findLeaves(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

if (root == null){

return res;

}

Map<Integer, List<Integer>> map = new HashMap<>();

depth(root, map);

for (List<Integer> list : map.values()){

res.add(list);

}

return res;

}

private int depth(TreeNode root, Map<Integer, List<Integer>> map){

if (root == null){

return 0;

}

int left = depth(root.left, map);

int right = depth(root.right, map);

int depth = Math.max(left, right) + 1;

if (!map.containsKey(depth)){

map.put(depth, new ArrayList<>());

}

map.get(depth).add(root.val);

return depth;

}

}

# 367\_Valid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer num, write a function which returns True if num is a perfect square else False.

Note: Do not use any built-in library function such as sqrt.

Example 1:

Input: 16

Returns: True

Example 2:

Input: 14

Returns: False

class Solution {

public boolean isPerfectSquare(int num) {

int start = 1;

int end = num;

int target = num;

while (start <= end){

long mid = start + (end - start) / 2;

if (mid \* mid == target){

return true;

}else if (mid \* mid < target){

start = (int) mid+1;

}else{

end = (int) mid-1;

}

}

return false;

}

}

note is that we have to use long for mid to avoid mid\*mid

from overflow. Also, you can use long type for low and high to avoid type casting for mid from long to int.

# 368\_LargestDivisibleSubset.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a set of distinct positive integers, find the largest subset such that every pair (Si, Sj) of

elements in this subset satisfies: Si % Sj = 0 or Sj % Si = 0.

If there are multiple solutions, return any subset is fine.

Example 1:

nums: [1,2,3]

Result: [1,2] (of course, [1,3] will also be ok)

Example 2:

nums: [1,2,4,8]

Result: [1,2,4,8]

class Solution {

public List<Integer> largestDivisibleSubset(int[] nums) {

List<Integer> result = new ArrayList<>();

if (nums == null || nums.length == 0){

return result;

}

Arrays.sort(nums);

int[] f = new int[nums.length]; // record longest chain at i

int[] prev = new int[nums.length]; // record the index of previous one at i in the longest chain

for (int i = 0; i < nums.length; i++){

f[i] = 1;

prev[i] = i;

}

for (int i = 0; i < nums.length; i++){

for (int j = 0; j < i; j++){

if (nums[i] % nums[j] == 0){

if (f[i] < f[j] + 1){

f[i] = f[j] + 1;

prev[i] = j;

}

}

}

}

int max = 0;

int max\_i = 0;

for (int i = 0; i < nums.length; i++){

if (f[i] > max){

max = f[i];

max\_i = i;

}

}

while (prev[max\_i] != max\_i){

result.add(0, nums[max\_i]);

max\_i = prev[max\_i];

}

result.add(0, nums[max\_i]);

return result;

}

}

class Solution {

public List<Integer> largestDivisibleSubset(int[] nums) {

List<Integer> res = new ArrayList<>();

if (nums == null || nums.length == 0){

return res;

}

int n = nums.length;

Arrays.sort(nums);

int[] count = new int[n];

int[] prev = new int[n];

for (int i = 0; i < n; i++){

count[i] = 1;

prev [i] = i;

}

int max = 0;

int index = 0;

for (int i = 0; i < n; i++){

for (int j = 0; j < i; j++){

if (nums[i] % nums[j] == 0){

if (count[i] < count[j] + 1){

count[i] = count[j] + 1;

prev[i] = j;

}

}

if (max < count[i]){

max = count[i];

index = i;

}

}

}

while (prev[index] != index){

res.add(nums[index]);

index = prev[index];

}

res.add(nums[index]);

return res;

}

}

# 369\_Plus.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-negative integer represented as non-empty a singly linked list of digits, plus one to the integer.

You may assume the integer do not contain any leading zero, except the number 0 itself.

The digits are stored such that the most significant digit is at the head of the list.

Example:

Input:

1->2->3

Output:

1->2->4

Method 1: reverse + plus one for array (66\_PlusOne.java)

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode plusOne(ListNode head) {

if (head == null){

return head;

}

ListNode reverseHead = reverse(head);

ListNode node = reverseHead;

while (node != null){

if (node.val < 9){

node.val += 1;

return reverse(reverseHead);

}

node.val = 0;

node = node.next;

}

ListNode newHead = new ListNode(1);

newHead.next = reverseHead;

return newHead;

}

private ListNode reverse(ListNode head){

ListNode prev = null;

while (head != null){

ListNode temp = head.next;

head.next = prev;

prev = head;

head = temp;

}

return prev;

}

}

Method 2:

how to add value reversely: Use DFS

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode plusOne(ListNode head) {

if (head == null){

return head;

}

if (dfs(head) == 0){

return head;

}else{

ListNode newHead = new ListNode(1);

newHead.next = head;

return newHead;

}

}

private int dfs(ListNode node){

if (node == null){

return 1;

}

int carry = dfs(node.next);

if (carry == 0){

return 0;

}

int val = node.val + 1;

node.val = val % 10;

return val / 10;

}

}

Best solution:

class Solution {

public ListNode plusOne(ListNode head){

if (head == null){

return head;

}

int carry = dfs(head);

if (carry == 0){

return head;

}

ListNode newHead = ListNode(1);

newHead.next = head;

return newHead;

}

private int dfs(ListNode node){

if (node == null){

return 1;

}

int carry = dfs(node.next);

if (carry == 0){

return 0;

}

int val = node.val + carry;

node.val = val%10;

return val/10;

}

}

Method 3:

Iterative Two-Pointers with dummy node Java O(n) time, O(1) space

i stands for the most significant digit that is going to be incremented if there exists a carry

dummy node can handle cases such as "9->9>-9" automatically

public class Solution {

public ListNode plusOne(ListNode head) {

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode i = dummy;

ListNode j = dummy;

while (j.next != null) {

j = j.next;

if (j.val != 9) {

i = j;

}

}

if (j.val != 9) {

j.val++;

} else {

i.val++;

i = i.next;

while (i != null) {

i.val = 0;

i = i.next;

}

}

if (dummy.val == 0) {

return dummy.next;

}

return dummy;

}

}

# 37\_PartitionArraybyOddandEven.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Partition an integers array into odd number first and even number second.

Have you met this question in a real interview? Yes

Example

Given [1, 2, 3, 4], return [1, 3, 2, 4]

public class Solution {

/\*

\* @param nums: an array of integers

\* @return: nothing

\*/

public void partitionArray(int[] nums) {

if (nums == null || nums.length == 0){

return;

}

int start = 0;

int end = nums.length - 1;

while (start < end){

while (start < end && nums[start] % 2 == 1){

start++;

}

while (start < end && nums[end] % 2 == 0){

end--;

}

if (start < end){

int temp = nums[start];

nums[start] = nums[end];

nums[end] = temp;

start++;

end--;

}

}

}

}

# 37\_Sudoku.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:

Each of the digits 1-9 must occur exactly once in each row.

Each of the digits 1-9 must occur exactly once in each column.

Each of the the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

Empty cells are indicated by the character '.'.

A sudoku puzzle...

...and its solution numbers marked in red.

Note:

The given board contain only digits 1-9 and the character '.'.

You may assume that the given Sudoku puzzle will have a single unique solution.

The given board size is always 9x9.

Method: backtracking

Time complexity:Try 1 through 9 for each cell. The time complexity should be 9 ^ m

(m represents the number of blanks to be filled in), since each blank can have 9 choices.

class Solution {

public void solveSudoku(char[][] board) {

if (board == null || board.length == 0){

return;

}

solve(board);

}

private boolean solve(char[][] board){

for (int i = 0; i < board.length; i++){

for (int j = 0; j < board[0].length; j++){

if (board[i][j] == '.'){

for (char c = '1'; c <= '9'; c++){

if (isValid(board, i, j, c)){

board[i][j] = c;

if (solve(board)){

return true;

}

board[i][j] = '.';

}

}

return false;

}

}

}

return true;

}

private boolean isValid(char[][] board, int r, int c, char ch){

for (int i = 0; i < board.length; i++){

if (board[i][c] == ch){

return false;

}

}

for (int j = 0; j < board[0].length; j++){

if (board[r][j] == ch){

return false;

}

}

int rStart = r/3 \* 3;

int cStart = c/3 \* 3;

for (int i = 0; i < 3; i++){

for (int j = 0; j < 3; j++){

if (board[rStart+i][cStart+j] == ch){

return false;

}

}

}

return true;

}

}

# 370\_[Range Addition](https://leetcode.com/problems/range-addition).java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Assume you have an array of length n initialized with all 0's and are given k update operations.

Each operation is represented as a triplet: [startIndex, endIndex, inc] which increments each element of subarray

A[startIndex ... endIndex] (startIndex and endIndex inclusive) with inc.

Return the modified array after all k operations were executed.

Example:

Given:

length = 5,

updates = [

[1, 3, 2],

[2, 4, 3],

[0, 2, -2]

]

Output:

[-2, 0, 3, 5, 3]

Explanation:

Initial state:

[ 0, 0, 0, 0, 0 ]

After applying operation [1, 3, 2]:

[ 0, 2, 2, 2, 0 ]

After applying operation [2, 4, 3]:

[ 0, 2, 5, 5, 3 ]

After applying operation [0, 2, -2]:

[-2, 0, 3, 5, 3 ]

Method 1:

Time complexity: O(nk)

class Solution {

public int[] getModifiedArray(int length, int[][] updates) {

int k = updates.length;

int[] result = new int[length];

for (int i = 0; i < k; i++){

for (int j = updates[i][0] ; j <= updates[i][1]; j++){

result[j] += updates[i][2];

}

}

return result;

}

}

Method 2: Range Caching

Time complexity: O(n+k)

class Solution {

public int[] getModifiedArray(int length, int[][] updates) {

int k = updates.length;

int[] result = new int[length];

for (int i = 0; i < k; i++){

int start = updates[i][0];

int end = updates[i][1];

result[start] += updates[i][2];

if (end < length - 1){

result[end+1] -= updates[i][2];

}

}

int sum = 0;

for (int i = 0; i < length; i++){

sum += result[i];

result[i] = sum;

}

return result;

}

}

Intuition

There is only one read query on the entire range, and it occurs at the end of all update queries. Additionally, the order

of processing update queries is irrelevant.

Cumulative sums or partial\_sum operations apply the effects of past elements to the future elements in the sequence.

Algorithm

The algorithm makes use of the above intuition to simply store changes at the borders of the update ranges

(instead of processing the entire range). Finally a single post processing operation is carried out over the

entire output array.

The two steps that are required are as follows:

For each update query (start,end,val)(start, end, val)(start,end,val) on the array arrarrarr, we need to do only two operations:

Update startstartstart boundary of the range:

arrstart=arrstart+val arr\_{start} = arr\_{start} + val arr​start​​=arr​start​​+val

Update just beyond the endendend boundary of the range:

arrend+1=arrend+1−val arr\_{end+1} = arr\_{end+1} - val arr​end+1​​=arr​end+1​​−val

Final Transformation. The cumulative sum of the entire array is taken (0 - based indexing)

arri=arri+arri−1∀i∈[1,n) arr\_i = arr\_i + arr\_{i-1} \quad \forall \quad i \in [1, n) arr​i​​=arr​i​​+arr​i−1​​∀i∈[1,n)

For each update query (start,end,val)(start, end, val)(start,end,val) on the array arrarrarr, the goal is to achieve the result:

arri=arri+val∀i∈[start,end] arr\_i = arr\_i + val \quad \forall \quad i \in [start, end] arr​i​​=arr​i​​+val∀i∈[start,end]

Applying the final transformation, ensures two things:

It carries over the +val+val+val increment over to every element arri∀i≥start arr\_i \; \forall \; i \ge start arr​i​​∀i≥start.

It carries over the −val-val−val increment (equivalently, a +val+val+val decrement) over to every element arrj∀j>end arr\_j \; \forall \; j \gt end arr​j​​∀j>end.

The net result is that:

arriarrj=arri+val=arrj+val−val=arrj∀i∈[start,end]∀i∈(end,length)

which meets our end goal. It is easy to see that the updates over a range did not carry over beyond it due to the compensating effect of the −val-val−val increment over the +val+val+val increment.

It is good to note that this works for multiple update queries because the particular binary operations here

(namely addition and subtraction):

are closed over the entire domain of Integers. (A counter example is division which is not closed over all Integers).

are complementary operations. (As a counter example multiplication and division are not always complimentary due to

possible loss of precision when dividing Integers).

371\_SumTwoIntegers.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Calculate the sum of two integers a and b, but you are not allowed to use the operator + and -.

Example:

Given a = 1 and b = 2, return 3.

class Solution {

public int getSum(int a, int b) {

if (b == 0){

return a;

}

while (b != 0){

int carry = a & b;

a = a ^ b;

b = carry << 1;

}

return a;

}

}

https://leetcode.com/problems/sum-of-two-integers/discuss/84290/Java-simple-easy-understand-solution-with-explanation

public int getSum(int a, int b) {

if(b == 0){//没有进为的时候完成运算

return a;

}

int sum,carry;

sum = a^b;//完成第一步加发的运算

carry = (a&b)<<1;//完成第二步进位并且左移运算

return getSum(sum,carry);//

}

# 373\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given two integer arrays nums1 and nums2 sorted in ascending order and an integer k.

Define a pair (u,v) which consists of one element from the first array and one element from the second array.

Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums.

Example 1:

Given nums1 = [1,7,11], nums2 = [2,4,6], k = 3

Return: [1,2],[1,4],[1,6]

The first 3 pairs are returned from the sequence:

[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]

Example 2:

Given nums1 = [1,1,2], nums2 = [1,2,3], k = 2

Return: [1,1],[1,1]

The first 2 pairs are returned from the sequence:

[1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]

Example 3:

Given nums1 = [1,2], nums2 = [3], k = 3

Return: [1,3],[2,3]

All possible pairs are returned from the sequence:

[1,3],[2,3]

Method 1: Brute force

Time complexity: O(mn)

Space complexity: O(mn)

class Solution {

public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {

List<int[]> result = new ArrayList<>();

int m = nums1.length;

int n = nums2.length;

int[][] nums = new int[m\*n][2];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

nums[i\*n+j][0] = nums1[i];

nums[i\*n+j][1] = nums2[j];

}

}

Arrays.sort(nums, new Comparator<int[]>(){

public int compare(int[] a, int[] b){

return a[0] + a[1] - (b[0] + b[1]);

}

});

for(int i = 0; i < k && i < m\*n; i++){

result.add(nums[i]);

}

return result;

}

}

Method 2: PQ with using the given information of sorted array

Time complexity: O(k \* log (min(m, k))

Space complexity: O(k)

class Solution {

public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {

List<int[]> res = new ArrayList<>();

if (nums1 == null || nums1.length == 0 || nums2 == null || nums2.length == 0){

return res;

}

Queue<int[]> pq = new PriorityQueue<int[]>(new Comparator<int[]>(){

public int compare (int[] a, int[] b){

return (a[0] + a[1]) - (b[0] + b[1]);

}

});

for (int i = 0; i < nums1.length && i < k; i++){

pq.offer(new int[]{nums1[i], nums2[0], 0});

}

while (!pq.isEmpty() && k > 0){

int[] curr = pq.poll();

res.add(new int[]{curr[0], curr[1]});

int nextIdx = curr[2] + 1;

if (nextIdx < nums2.length){

pq.offer(new int[]{curr[0], nums2[nextIdx], nextIdx});

}

k--;

}

return res;

}

}

Method 3:

Time complexity: O(klogk)

Space complexity: O(k)

class Solution {

public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {

List<int[]> result = new ArrayList<>();

if (nums1.length == 0 || nums2.length == 0 || k == 0){

return result;

}

Queue<int[]> minIndexHeap = new PriorityQueue<int[]>(new Comparator<int[]>(){

public int compare(int[] a, int[] b){

return nums1[a[0]] + nums2[a[1]] - (nums1[b[0]] + nums2[b[1]]);

}

});

minIndexHeap.offer(new int[]{0, 0});

while (!minIndexHeap.isEmpty() && k > 0){

int[] curIndex = minIndexHeap.poll();

result.add(new int[]{nums1[curIndex[0]], nums2[curIndex[1]]});

if (curIndex[1] != nums2.length - 1){

minIndexHeap.offer(new int[]{curIndex[0], curIndex[1] + 1});

}

if (curIndex[0] != nums1.length - 1 && curIndex[1] == 0){

minIndexHeap.offer(new int[]{curIndex[0] + 1, 0});

}

k--;

}

return result;

}

}

Best solution:

class Solution {

public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {

List<int[]> res = new ArrayList<>();

if (nums1 == null || nums1.length == 0 || nums2 == null || nums2.length == 0){

return res;

}

Queue<int[]> pqInd = new PriorityQueue<>(new Comparator<int[]>(){

public int compare (int[] a, int[] b){

return nums1[a[0]] + nums2[a[1]] - (nums1[b[0]] + nums2[b[1]]);

}

});

for (int i = 0; i < nums1.length; i++){

pqInd.offer(new int[]{i, 0});

}

while (!pqInd.isEmpty() && k > 0){

int[] curr = pqInd.poll();

res.add(new int[]{nums1[curr[0]], nums2[curr[1]]});

k--;

if (curr[1] < nums2.length - 1){

pqInd.offer(new int[]{curr[0], curr[1] + 1});

}

}

return res;

}

}

# 374\_[Guess Number Higher or Lower](https://leetcode.com/problems/guess-number-higher-or-lower).java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are playing the Guess Game. The game is as follows:

I pick a number from 1 to n. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number is higher or lower.

You call a pre-defined API guess(int num) which returns 3 possible results (-1, 1, or 0):

-1 : My number is lower

1 : My number is higher

0 : Congrats! You got it!

Example:

n = 10, I pick 6.

Return 6.

/\* The guess API is defined in the parent class GuessGame.

@param num, your guess

@return -1 if my number is lower, 1 if my number is higher, otherwise return 0

int guess(int num); \*/

public class Solution extends GuessGame {

public int guessNumber(int n) {

int start = 1;

int end = n;

while (start + 1 < end){

int mid = start + (end - start) / 2;

int res = guess(mid);

if (res == 0){

return mid;

}else if (res < 0){

end = mid;

}else {

start = mid;

}

}

if (guess(end) == 0){

return end;

}

return start;

}

}

# 376\_[Wiggle Subsequence](https://leetcode.com/problems/wiggle-subsequence).java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A sequence of numbers is called a wiggle sequence if the differences between successive numbers strictly alternate between positive

and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is

trivially a wiggle sequence.

For example, [1,7,4,9,2,5] is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast,

[1,4,7,2,5] and [1,7,4,5,5] are not wiggle sequences, the first because its first two differences are positive and the second because

its last difference is zero.

Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by

deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original

order.

Examples:

Input: [1,7,4,9,2,5]

Output: 6

The entire sequence is a wiggle sequence.

Input: [1,17,5,10,13,15,10,5,16,8]

Output: 7

There are several subsequences that achieve this length. One is [1,17,10,13,10,16,8].

Input: [1,2,3,4,5,6,7,8,9]

Output: 2

Follow up:

Can you do it in O(n) time?

https://leetcode.com/articles/wiggle-subsequence/

up[i] refers to the length of the longest wiggle subsequence obtained so far considering i^{th}i

the element as the last element of the wiggle subsequence and ending with a rising wiggle.

Similarly, down[i]down[i] refers to the length of the longest wiggle subsequence obtained so far considering i^{th}i

the element as the last element of the wiggle subsequence and ending with a falling wiggle.

Method 1:

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public int wiggleMaxLength(int[] nums) {

if (nums.length < 2){

return nums.length;

}

int n = nums.length;

int[] up = new int[n]; //dp[i] denote longest result ending at i, (note result must include the element at i)

int[] down = new int[n];

Arrays.fill(up, 1);

Arrays.fill(down, 1);

for (int i = 1; i < n; i++){

for (int j = 0; j < i; j++){

if (nums[i] > nums[j]){

up[i] = Math.max(up[i], down[j] + 1);

}else if (nums[i] < nums[j]){

down[i] = Math.max(down[i], up[j] + 1);

}

}

}

return Math.max(up[n-1], down[n-1]);

}

}

Similar as LIS

class Solution {

public int wiggleMaxLength(int[] nums) {

if (nums.length < 2){

return nums.length;

}

int n = nums.length;

int[] up = new int[n]; //dp[i] denote longest result ending at i, (note result must include the element at i)

int[] down = new int[n];

Arrays.fill(up, 1);

Arrays.fill(down, 1);

int max = 1;

for (int i = 1; i < n; i++){

for (int j = 0; j < i; j++){

if (nums[i] > nums[j]){

up[i] = Math.max(up[i], down[j] + 1);

}else if (nums[i] < nums[j]){

down[i] = Math.max(down[i], up[j] + 1);

}

max = Math.max(max, up[i]);

max = Math.max(max, down[i]);

}

}

return max;

}

}

Method 2:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int wiggleMaxLength(int[] nums) {

if (nums.length < 2){

return nums.length;

}

int n = nums.length;

int[] up = new int[n]; //dp[i] denote longest result ending at i, (note result may or may include the element at i)

int[] down = new int[n];

up[0] = 1;

down[0] = 1;

for (int i = 1; i < n; i++){

if (nums[i] > nums[i-1]){

up[i] = down[i-1] + 1;

down[i] = down[i-1];

}else if (nums[i] < nums[i-1]){

down[i] = up[i-1] + 1;

up[i] = up[i-1];

}else{

up[i] = up[i-1];

down[i] = down[i-1];

}

}

return Math.max(up[n-1], down[n-1]);

}

}

Method 3: Best solution

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public int wiggleMaxLength(int[] nums) {

if (nums.length < 2){

return nums.length;

}

int n = nums.length;

int up = 1;

int down = 1;

for (int i = 1; i < n; i++){

if (nums[i] > nums[i-1]){

up = down + 1;

}else if (nums[i] < nums[i-1]){

down = up + 1;

}

}

return Math.max(up, down);

}

}

# 377\_[Combination Sum IV](https://leetcode.com/problems/combination-sum-iv).java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations

that add up to a positive integer target.

Example:

nums = [1, 2, 3]

target = 4

The possible combination ways are:

(1, 1, 1, 1)

(1, 1, 2)

(1, 2, 1)

(1, 3)

(2, 1, 1)

(2, 2)

(3, 1)

Note that different sequences are counted as different combinations.

Therefore the output is 7.

Follow up:

What if negative numbers are allowed in the given array?

How does it change the problem?

What limitation we need to add to the question to allow negative numbers?

Method 1: DFS TLE

class Solution {

public int combinationSum4(int[] nums, int target) {

List<List<Integer>> result = new ArrayList<>();

Arrays.sort(nums);

dfs(result, new ArrayList<>(), nums, target);

return result.size();

}

private void dfs(List<List<Integer>> result, List<Integer> item, int[] nums, int target){

if (target == 0){

result.add(new ArrayList<>(item));

return;

}

for (int i = 0; i < nums.length; i++){

if (target < nums[i]){

break;

}

item.add(nums[i]);

dfs(result, item, nums, target - nums[i]);

item.remove(item.size() - 1);

}

}

}

Method 2: recursion + memorization (this is also kind of DP -- top down)

class Solution {

Map<Integer, Integer> map = new HashMap<>();

public int combinationSum4(int[] nums, int target) {

if (map.containsKey(target)){

return map.get(target);

}

if (target == 0){

return 1;

}

int ans = 0;

for (int i = 0; i < nums.length; i++){

if (target >= nums[i]){

ans += combinationSum4(nums, target - nums[i]);

}

}

map.put(target, ans);

return ans;

}

}

https://leetcode.com/problems/combination-sum-iv/discuss/85036/1ms-Java-DP-Solution-with-Detailed-Explanation

Now for a DP solution, we just need to figure out a way to store the intermediate results, to avoid the same

combination sum being calculated many times. We can use an array to save those results, and check if there is

already a result before calculation. We can fill the array with -1 to indicate that

the result hasn't been calculated yet. 0 is not a good choice because it means there is no combination sum for the target.

Different from backpack DP, similar as LIS

Method 3: DP iteration -- bottom up

class Solution {

public int combinationSum4(int[] nums, int target) {

int[] dp = new int[target+1];

dp[0] = 1; // when the target is 0, the dp should be 1 not 0

for (int i = 1; i <= target; i++){

for (int j = 0; j < nums.length; j++){

if (i >= nums[j]){

dp[i] += dp[i - nums[j]];

}

}

}

return dp[target];

}

}

# 378\_ConvertBSTtoDoublyLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

onvert a binary search tree to doubly linked list with in-order traversal.

Have you met this question in a real interview? Yes

Example

Given a binary search tree:

4

/ \

2 5

/ \

1 3

return 1<->2<->3<->4<->5

/\*\*

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\* Definition for Doubly-ListNode.

\* public class DoublyListNode {

\* int val;

\* DoublyListNode next, prev;

\* DoublyListNode(int val) {

\* this.val = val;

\* this.next = this.prev = null;

\* }

\* }

\*/

public class Solution {

/\*

\* @param root: The root of tree

\* @return: the head of doubly list node

\*/

public DoublyListNode bstToDoublyList(TreeNode root) {

if (root == null){

return null;

}

Queue<Integer> queue = new LinkedList<Integer>();

traverse(root, queue);

DoublyListNode dummy = new DoublyListNode(0);

DoublyListNode head = dummy;

while (!queue.isEmpty()){

DoublyListNode temp = new DoublyListNode(queue.poll());

head.next = temp;

temp.prev = head;

head = head.next;

}

return dummy.next;

}

private void traverse(TreeNode root, Queue<Integer> queue){

if (root == null){

return;

}

traverse(root.left, queue);

queue.offer(root.val);

traverse(root.right,queue);

}

}

Iteration:

/\*\*

\* Definition for Doubly-ListNode.

\* public class DoublyListNode {

\* int val;

\* DoublyListNode next, prev;

\* DoublyListNode(int val) {

\* this.val = val;

\* this.next = this.prev = null;

\* }

\* } \* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

public class Solution {

/\*\*

\* @param root: The root of tree

\* @return: the head of doubly list node

\*/

public DoublyListNode bstToDoublyList(TreeNode root) {

Stack<TreeNode> stack = new Stack<>();

TreeNode node = root;

DoublyListNode dummy = new DoublyListNode(-1);

DoublyListNode prev = dummy;

while (!stack.isEmpty() || node != null){

while (node != null){

stack.push(node);

node = node.left;

}

TreeNode curr = stack.pop();

DoublyListNode temp = new DoublyListNode(curr.val);

prev.next = temp;

temp.prev = prev;

prev = temp;

node = curr.right;

}

return dummy.next;

}

}

# 38\_CountandSay.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The count-and-say sequence is the sequence of integers with the first five terms as following:

1. 1

2. 11

3. 21

4. 1211

5. 111221

1 is read off as "one 1" or 11.

11 is read off as "two 1s" or 21.

21 is read off as "one 2, then one 1" or 1211.

Given an integer n, generate the nth term of the count-and-say sequence.

Note: Each term of the sequence of integers will be represented as a string.

Example 1:

Input: 1

Output: "1"

Example 2:

Input: 4

Output: "1211"

class Solution {

public String countAndSay(int n) {

StringBuilder cur = new StringBuilder("1");

StringBuilder prev = new StringBuilder();

for (int i = 1; i < n; i++){

prev = cur;

cur = new StringBuilder();

char say = prev.charAt(0);

int count = 1;

for (int j = 1; j < prev.length(); j++){

if (prev.charAt(j) != say){

cur.append(String.valueOf(count)).append(say);

say = prev.charAt(j);

count = 1;

}else{

count++;

}

}

cur.append(String.valueOf(count)).append(say);

}

return cur.toString();

}

}

Better version:

class Solution {

public String countAndSay(int n) {

String str = "1";

for (int i = 1; i < n; i++){

int j = 0;

StringBuilder sb = new StringBuilder();

while (j < str.length()){

char c = str.charAt(j);

int k = j + 1;

int times = 1;

while (k < str.length() && str.charAt(k) == c){

k++;

times++;

}

sb.append(times);

sb.append(c);

j = k;

}

str = sb.toString();

}

return str;

}

}

class Solution {

public String countAndSay(int n) {

String curr = "1";

for (int i = 1; i < n; i++){

StringBuilder sb = new StringBuilder();

int j = 0;

while (j < curr.length()){

int k = j;

while (k < curr.length() && curr.charAt(k) == curr.charAt(j)){

k++;

}

sb.append(k-j);

sb.append(curr.charAt(j));

j = k;

}

curr = sb.toString();

}

return curr;

}

}

# 38\_Search2DMatrixII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write an efficient algorithm that searches for a value in an m x n matrix, return the occurrence of it.

This matrix has the following properties:

Integers in each row are sorted from left to right.

Integers in each column are sorted from up to bottom.

No duplicate integers in each row or column.

Have you met this question in a real interview? Yes

Example

Consider the following matrix:

[

[1, 3, 5, 7],

[2, 4, 7, 8],

[3, 5, 9, 10]

]

Given target = 3, return 2.

Challenge

O(m+n) time and O(1) extra space

• 左下角这个元素非常特殊(记为x)

– 如果target < x 最后的一行还有继续比较的意义吗?

– 如果target > x 最左的一列还有继续比较的意义吗?

– 如果target = x最后一行 & 最左的一列还有继续比较的意义吗?

• 所以算法是:

– target < x 砍掉最后一行

– target > x 砍掉最左一列

– target = x 计数+1,砍掉最后一行&砍掉最左一列

• 时间复杂度O(n+m)

– 盯住哪个元素会成为左下角元素x,x每次向上or向右or右上走一步 – 在矩阵中最多只能走n+m步

public class Solution {

/\*

\* @param matrix: A list of lists of integers

\* @param target: An integer you want to search in matrix

\* @return: An integer indicate the total occurrence of target in the given matrix

\*/

public int searchMatrix(int[][] matrix, int target) {

int r = matrix.length - 1;

int c = 0;

int count = 0;

while (r >= 0 && c < matrix[0].length){

if (matrix[r][c] == target){

r--;

c++;

count++;

}else if (matrix[r][c] < target){

c++;

}else{

r--;

}

}

return count;

}

}

# 380\_Load\_Balancer.java [Insert Delete GetRandom O(1)](https://leetcode.com/problems/insert-delete-getrandom-o1)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a data structure that supports all following operations in average O(1) time.

insert(val): Inserts an item val to the set if not already present.

remove(val): Removes an item val from the set if present.

getRandom: Returns a random element from current set of elements. Each element must have the same probability of

being returned.

Example:

// Init an empty set.

RandomizedSet randomSet = new RandomizedSet();

// Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomSet.insert(1);

// Returns false as 2 does not exist in the set.

randomSet.remove(2);

// Inserts 2 to the set, returns true. Set now contains [1,2].

randomSet.insert(2);

// getRandom should return either 1 or 2 randomly.

randomSet.getRandom();

// Removes 1 from the set, returns true. Set now contains [2].

randomSet.remove(1);

// 2 was already in the set, so return false.

randomSet.insert(2);

// Since 2 is the only number in the set, getRandom always return 2.

randomSet.getRandom();

class RandomizedSet {

private Map<Integer, Integer> map;

private Random random;

private List<Integer> list;

/\*\* Initialize your data structure here. \*/

public RandomizedSet() {

map = new HashMap<Integer, Integer>(); //first integer records the val, second integer records the index in the array

list = new ArrayList<Integer>();

random = new Random();

}

/\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/

public boolean insert(int val) {

if (!map.containsKey(val)){

map.put(val, list.size());

list.add(val);

return true;

}

return false;

}

/\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/

public boolean remove(int val) {

if (map.containsKey(val)){

int index = map.get(val);

int lastNum = list.get(list.size() - 1);

map.put(lastNum, index);

list.set(index, lastNum);

map.remove(val);

list.remove(list.size() - 1);

return true;

}

return false;

}

/\*\* Get a random element from the set. \*/

public int getRandom() {

return list.get(random.nextInt(list.size()));

}

}

/\*\*

\* Your RandomizedSet object will be instantiated and called as such:

\* RandomizedSet obj = new RandomizedSet();

\* boolean param\_1 = obj.insert(val);

\* boolean param\_2 = obj.remove(val);

\* int param\_3 = obj.getRandom();

\*/

# 381\_[Insert Delete GetRandom O(1) - Duplicates allowed](https://leetcode.com/problems/insert-delete-getrandom-o1-duplicates-allowed).java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a data structure that supports all following operations in average O(1) time.

Note: Duplicate elements are allowed.

insert(val): Inserts an item val to the collection.

remove(val): Removes an item val from the collection if present.

getRandom: Returns a random element from current collection of elements. The probability of each element being returned is linearly related to the number of same value the collection contains.

Example:

// Init an empty collection.

RandomizedCollection collection = new RandomizedCollection();

// Inserts 1 to the collection. Returns true as the collection did not contain 1.

collection.insert(1);

// Inserts another 1 to the collection. Returns false as the collection contained 1. Collection now contains [1,1].

collection.insert(1);

// Inserts 2 to the collection, returns true. Collection now contains [1,1,2].

collection.insert(2);

// getRandom should return 1 with the probability 2/3, and returns 2 with the probability 1/3.

collection.getRandom();

// Removes 1 from the collection, returns true. Collection now contains [1,2].

collection.remove(1);

// getRandom should return 1 and 2 both equally likely.

collection.getRandom();

class RandomizedCollection {

Random random;

List<Integer> list;

Map<Integer, Set<Integer>> map;

/\*\* Initialize your data structure here. \*/

public RandomizedCollection() {

random = new Random();

list = new ArrayList<>();

map = new HashMap<>();

}

/\*\* Inserts a value to the collection. Returns true if the collection did not already contain the specified element. \*/

public boolean insert(int val) {

if (map.containsKey(val)){

map.get(val).add(list.size());

list.add(val);

return false;

}

map.put(val, new HashSet<>());

map.get(val).add(list.size());

list.add(val);

return true;

}

/\*\* Removes a value from the collection. Returns true if the collection contained the specified element. \*/

public boolean remove(int val) {

if (!map.containsKey(val) || map.get(val).size() == 0){

return false;

}

Set<Integer> set = map.get(val);

int index = set.iterator().next();

int lastIndex = list.size() - 1;

int lastNum = list.get(lastIndex);

list.set(index, lastNum);

if (val != lastNum){

map.get(lastNum).add(index);

map.get(val).remove(index);

}

map.get(lastNum).remove(lastIndex);

list.remove(lastIndex);

return true;

}

/\*\* Get a random element from the collection. \*/

public int getRandom() {

return list.get(random.nextInt(list.size()));

}

}

/\*\*

\* Your RandomizedCollection object will be instantiated and called as such:

\* RandomizedCollection obj = new RandomizedCollection();

\* boolean param\_1 = obj.insert(val);

\* boolean param\_2 = obj.remove(val);

\* int param\_3 = obj.getRandom();

\*/

# 382\_RandomNode.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a singly linked list, return a random node's value from the linked list. Each node must have the same

probability of being chosen.

Follow up:

What if the linked list is extremely large and its length is unknown to you? Could you solve this efficiently

without using extra space?

Example:

// Init a singly linked list [1,2,3].

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

Solution solution = new Solution(head);

// getRandom() should return either 1, 2, or 3 randomly. Each element should have equal probability of returning.

solution.getRandom();

Method 1: Calculate length first

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

private ListNode node;

private Random random;

/\*\* @param head The linked list's head.

Note that the head is guaranteed to be not null, so it contains at least one node. \*/

public Solution(ListNode head) {

node = head;

random = new Random();

}

/\*\* Returns a random node's value. \*/

public int getRandom() {

ListNode cur = node;

int len = 0;

while (cur != null){

len++;

cur = cur.next;

}

int targetIndex = random.nextInt(len);

cur = node;

int i = 0;

while (cur != null && i < targetIndex){

cur = cur.next;

i++;

}

return cur.val;

}

}

/\*\*

\* Your Solution object will be instantiated and called as such:

\* Solution obj = new Solution(head);

\* int param\_1 = obj.getRandom();

\*/

Method 2 (good): don't calculate length (reveroir sampling)

https://leetcode.com/problems/linked-list-random-node/discuss/85662/Java-Solution-with-cases-explain

http://blog.jobbole.com/42550/

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

private ListNode head;

private Random random;

/\*\* @param head The linked list's head.

Note that the head is guaranteed to be not null, so it contains at least one node. \*/

public Solution(ListNode head) {

this.head = head;

random = new Random();

}

/\*\* Returns a random node's value. \*/

public int getRandom() {

ListNode cur = head;

int i = 0;

int ans = cur.val;

while (cur != null){

if (random.nextInt(i + 1) == i){

ans = cur.val;

}

cur = cur.next;

i++;

}

return ans;

}

}

/\*\*

\* Your Solution object will be instantiated and called as such:

\* Solution obj = new Solution(head);

\* int param\_1 = obj.getRandom();

\*/

# 382\_TriangleCount.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers, how many three numbers can be found in the array, so that we can

build an triangle whose three edges length is the three numbers that we find?

Have you met this question in a real interview? Yes

Example

Given array S = [3,4,6,7], return 3. They are:

[3,4,6]

[3,6,7]

[4,6,7]

Given array S = [4,4,4,4], return 4. They are:

[4(1),4(2),4(3)]

[4(1),4(2),4(4)]

[4(1),4(3),4(4)]

[4(2),4(3),4(4)]

public class Solution {

/\*

\* @param S: A list of integers

\* @return: An integer

\*/

public int triangleCount(int[] S) {

if (S == null || S.length == 0){

return 0;

}

Arrays.sort(S);

int ans = 0;

for (int i = 2; i < S.length; i++){

int left = 0;

int right = i - 1;

while (left < right){

if (S[left] + S[right] > S[i]){

ans += right - left;

right--;

}else{

left++;

}

}

}

return ans;

}

}

# 383\_Ransom.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an arbitrary ransom note string and another string containing letters from all the magazines,

write a function that will return true if the ransom note can be constructed from the magazines ; otherwise, it will return false.

Each letter in the magazine string can only be used once in your ransom note.

Note:

You may assume that both strings contain only lowercase letters.

canConstruct("a", "b") -> false

canConstruct("aa", "ab") -> false

canConstruct("aa", "aab") -> true

class Solution {

public boolean canConstruct(String ransomNote, String magazine) {

int[] hash = new int[26];

for (int i = 0; i < magazine.length(); i++){

hash[magazine.charAt(i) - 'a']++;

}

for (int i = 0; i < ransomNote.length(); i++){

hash[ransomNote.charAt(i) - 'a']--;

if (hash[ransomNote.charAt(i) - 'a'] < 0){

return false;

}

}

return true;

}

}

384\_Longest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, find the length of the longest substring without repeating characters.

Have you met this question in a real interview?

Example

For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3.

For "bbbbb" the longest substring is "b", with the length of 1.

class Solution {

public int lengthOfLongestSubstring(String s) {

Map<Character, Integer> map = new HashMap<>();

int count = 0;

int start = 0;

int end = 0;

int ans = 0;

while (end < s.length()){

char endCh = s.charAt(end);

map.put(endCh, map.getOrDefault(endCh, 0) + 1);

if (map.get(endCh) == 2){

count++;

}

end++;

while (count > 0){

char startCh = s.charAt(start);

if (map.get(startCh) == 2){

count--;

}

map.put(startCh, map.get(startCh) - 1);

start++;

}

ans = Math.max(ans, end - start);

}

return ans;

}

}

# 384\_ShuffleArray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Shuffle a set of numbers without duplicates.

Example:

// Init an array with set 1, 2, and 3.

int[] nums = {1,2,3};

Solution solution = new Solution(nums);

// Shuffle the array [1,2,3] and return its result. Any permutation of [1,2,3] must equally likely to be returned.

solution.shuffle();

// Resets the array back to its original configuration [1,2,3].

solution.reset();

// Returns the random shuffling of array [1,2,3].

solution.shuffle();

Random return integer between each between zero (inclusive) and one (exclusive).

class Solution {

private int[] nums;

private Random random;

public Solution(int[] nums) {

this.nums = nums;

random = new Random();

}

/\*\* Resets the array to its original configuration and return it. \*/

public int[] reset() {

return nums;

}

/\*\* Returns a random shuffling of the array. \*/

public int[] shuffle() {

int[] temp = new int[nums.length];

for (int i = 0; i < nums.length; i++){

int j = random.nextInt(i+1);

temp[i] = temp[j];

temp[j] = nums[i];

}

return temp;

}

}

/\*\*

\* Your Solution object will be instantiated and called as such:

\* Solution obj = new Solution(nums);

\* int[] param\_1 = obj.reset();

\* int[] param\_2 = obj.shuffle();

\*/

# 385\_Mini.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a nested list of integers represented as a string, implement a parser to deserialize it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Note: You may assume that the string is well-formed:

String is non-empty.

String does not contain white spaces.

String contains only digits 0-9, [, - ,, ].

Example 1:

Given s = "324",

You should return a NestedInteger object which contains a single integer 324.

Example 2:

Given s = "[123,[456,[789]]]",

Return a NestedInteger object containing a nested list with 2 elements:

1. An integer containing value 123.

2. A nested list containing two elements:

i. An integer containing value 456.

ii. A nested list with one element:

a. An integer containing value 789.

Method:

/\*\*

\* // This is the interface that allows for creating nested lists.

\* // You should not implement it, or speculate about its implementation

\* public interface NestedInteger {

\* // Constructor initializes an empty nested list.

\* public NestedInteger();

\*

\* // Constructor initializes a single integer.

\* public NestedInteger(int value);

\*

\* // @return true if this NestedInteger holds a single integer, rather than a nested list.

\* public boolean isInteger();

\*

\* // @return the single integer that this NestedInteger holds, if it holds a single integer

\* // Return null if this NestedInteger holds a nested list

\* public Integer getInteger();

\*

\* // Set this NestedInteger to hold a single integer.

\* public void setInteger(int value);

\*

\* // Set this NestedInteger to hold a nested list and adds a nested integer to it.

\* public void add(NestedInteger ni);

\*

\* // @return the nested list that this NestedInteger holds, if it holds a nested list

\* // Return null if this NestedInteger holds a single integer

\* public List<NestedInteger> getList();

\* }

\*/

class Solution {

public NestedInteger deserialize(String s) {

if (s == null || s.length() == 0){

return null;

}

if (s.charAt(0) != '['){

return new NestedInteger(Integer.valueOf(s));

}

NestedInteger res = null;

Stack<NestedInteger> stack = new Stack<>();

int num = 0;

int sign = 1;

boolean parseNum = false;

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (c == '['){

if (res != null){

stack.push(res);

}

res = new NestedInteger();

}else if (c == ','){

if (parseNum == true){

res.add(new NestedInteger(num));

num = 0;

sign = 1;

parseNum = false;

}

}else if (Character.isDigit(c)){

num = num \* 10 + sign \* (int)(c - '0');

parseNum = true;

}else if (c == '-'){

sign = -1;

}else if (c == ']'){

if (parseNum){

res.add(new NestedInteger(num));

num = 0;

sign = 1;

parseNum = false;

}

if (!stack.isEmpty()){

stack.peek().add(res);

res = stack.pop();

}

}

}

return res;

}

}

# 386\_Lexicographical.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer n, return 1 - n in lexicographical order.

For example, given 13, return: [1,10,11,12,13,2,3,4,5,6,7,8,9].

Please optimize your algorithm to use less time and space. The input size may be as large as 5,000,000.

https://leetcode.com/problems/lexicographical-numbers/discuss/86242/Java-O(n)-time-O(1)-space-iterative-solution-130ms

class Solution {

public List<Integer> lexicalOrder(int n) {

List<Integer> result = new ArrayList<>();

int curr = 1;

for (int i = 1; i <= n; i++){

result.add(curr);

if (curr \* 10 <= n){

curr \*= 10;

}else if (curr % 10 != 9 && curr + 1 <= n){

curr++;

}else{

while ((curr/10) % 10 == 9){

curr /= 10;

}

curr = curr / 10 + 1;

}

}

return result;

}

}

# 387\_FirstUniqueChar.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

Examples:

s = "leetcode"

return 0.

s = "loveleetcode",

return 2.

Note: You may assume the string contain only lowercase letters.

class Solution {

public int firstUniqChar(String s) {

if (s == null || s.length() == 0){

return -1;

}

int MAX\_CHAR = 26;

int[] alphabet = new int[MAX\_CHAR];

for (int i = 0; i < s.length(); i++){

alphabet[s.charAt(i) - 'a']++;

}

for (int i = 0; i < s.length(); i++){

if (alphabet[s.charAt(i) - 'a'] == 1){

return i;

}

}

return -1;

}

}

class Solution {

public int firstUniqChar(String s) {

Map<Character, Integer> map = new HashMap<>();

for (char c : s.toCharArray()){

map.put(c, map.getOrDefault(c, 0) + 1);

}

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (map.get(c) == 1){

return i;

}

}

return -1;

}

}

# 389\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings s and t which consist of only lowercase letters.

String t is generated by random shuffling string s and then add one more letter at a random position.

Find the letter that was added in t.

Example:

Input:

s = "abcd"

t = "abcde"

Output:

e

Explanation:

'e' is the letter that was added.

Method 1:

class Solution {

public char findTheDifference(String s, String t) {

int[] hash = new int[26];

for (int i = 0; i < s.length(); i++){

hash[s.charAt(i) - 'a']++;

}

for (int i = 0; i < t.length(); i++){

hash[t.charAt(i) - 'a']--;

if (hash[t.charAt(i) - 'a'] < 0){

return t.charAt(i);

}

}

return '\0';

}

}

Method 2: Use ASCII code

class Solution {

public char findTheDifference(String s, String t) {

int charCode = (int) t.charAt(s.length()); // store the last character of t as ASCII code

for (int i = 0; i < s.length(); i++){

charCode -= (int) s.charAt(i);

charCode += (int) t.charAt(i);

}

return (char) (charCode);

}

}

Method 3: the same concept as single number I

class Solution {

public char findTheDifference(String s, String t) {

char c = 0;

for (int i = 0; i < s.length(); i++){

c ^= s.charAt(i);

}

for (int i = 0; i < t.length(); i++){

c ^= t.charAt(i);

}

return c;

}

}

# 39\_CombinationSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a set of candidate numbers (C) (without duplicates) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

The same repeated number may be chosen from C unlimited number of times.

Note:

All numbers (including target) will be positive integers.

The solution set must not contain duplicate combinations.

For example, given candidate set [2, 3, 6, 7] and target 7,

A solution set is:

[

[7],

[2, 2, 3]

]

Permutation no need sort

class Solution {

public List<List<Integer>> combinationSum(int[] candidates, int target) {

List<List<Integer>> res = new ArrayList<>();

dfs(res, new ArrayList<>(), candidates, target, 0);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, int[] candidates, int target, int pos){

if (target == 0){

res.add(new ArrayList<>(item));

return;

}

for (int i = pos; i < candidates.length; i++){

if (candidates[i] > target){

continue;

}

item.add(candidates[i]);

dfs(res, item, candidates, target - candidates[i], i);

item.remove(item.size() - 1);

}

}

}

class Solution {

public List<List<Integer>> combinationSum(int[] candidates, int target) {

List<List<Integer>> result = new ArrayList<>();

if (candidates == null || candidates.length == 0){

return result;

}

Arrays.sort(candidates);

dfs(result, new ArrayList<Integer>(), target, candidates, 0);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int target, int[] candidates, int start){

if (target == 0){

result.add(new ArrayList<Integer>(item));

return;

}

for (int i = start; i < candidates.length; i++){

if (candidates[i] > target){

break;

}

item.add(candidates[i]);

dfs(result, item, target - candidates[i], candidates, i);

item.remove(item.size() - 1);

}

}

}

# 390\_Elimination.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a list of sorted integers from 1 to n. Starting from left to right, remove the first number and every other number afterward until you reach the end of the list.

Repeat the previous step again, but this time from right to left, remove the right most number and every other number from the remaining numbers.

We keep repeating the steps again, alternating left to right and right to left, until a single number remains.

Find the last number that remains starting with a list of length n.

Example:

Input:

n = 9,

1 2 3 4 5 6 7 8 9

2 4 6 8

2 6

6

Output:

6

Method:

https://leetcode.com/problems/elimination-game/discuss/87119/JAVA:-Easiest-solution-O(logN)-with-explanation

Time complexity: O(logn)

Space complexity: O(1)

class Solution {

public int lastRemaining(int n) {

int head = 1;

int step = 1;

boolean left = true;

int remain = n;

while (remain > 1){

if (left || remain % 2 == 1){

head += step;

}

step \*= 2;

remain /= 2;

left = !left;

}

return head;

}

}

# 391\_NumberSky.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an interval list which are flying and landing time of the flight. How many airplanes are on the sky at most?

Notice

If landing and flying happens at the same time, we consider landing should happen at first.

Example

For interval list

[

[1,10],

[2,3],

[5,8],

[4,7]

]

Return 3

Method: sweep line O(nlogn), n is the number of points

/\*\*

\* Definition of Interval:

\* public classs Interval {

\* int start, end;

\* Interval(int start, int end) {

\* this.start = start;

\* this.end = end;

\* }

\*/

public class Solution {

/\*

\* @param airplanes: An interval array

\* @return: Count of airplanes are in the sky.

\*/

class Point{

int time;

int flag;

public Point(int time, int flag){

this.time = time;

this.flag = flag;

}

}

public int countOfAirplanes(List<Interval> airplanes) {

List<Point> pointList = new ArrayList<>();

for (Interval i : airplanes){

pointList.add(new Point(i.start, 1));

pointList.add(new Point(i.end, 0));

}

Collections.sort(pointList, new Comparator<Point>(){

public int compare(Point a, Point b){

if (a.time == b.time){

return a.flag - b.flag;

}

return a.time - b.time;

}

});

int count = 0;

int max = 0;

for (Point p : pointList){

if (p.flag == 1){

count++;

}else{

count--;

}

max = Math.max(max, count);

}

return max;

}

}

Method 2 treemap

public int countOfAirplanes(List<Interval> airplanes){

TreeMap<Integer, Integer> treemap = new TreeMap<>();

for (Interval plane : airplanes){

treemap.put(plane, treemap.getOrDefault(plane, 0) + 1);

treemap.put(plane, treemap.getOrDefault(plane, 0) - 1);

}

int active = 0;

int max = 0;

for (int i : treemap.values){

active += i;

max = Math.max(max, active);

}

return max;

}

# 392\_IsSubsequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s and a string t, check if s is subsequence of t.

You may assume that there is only lower case English letters in both s and t. t is potentially a very long (length ~= 500,000) string, and s is a short string (<=100).

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ace" is a subsequence of "abcde" while "aec" is not).

Example 1:

s = "abc", t = "ahbgdc"

Return true.

Example 2:

s = "axc", t = "ahbgdc"

Return false.

Follow up:

If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B, and you want to check one by one to see if T has its subsequence. In this scenario, how would you change your code?

Method 1: 50ms

O(mn)

class Solution {

public boolean isSubsequence(String s, String t) {

if (s.length() == 0){

return true;

}

int indexS = 0;

int indexT = 0;

while (indexT < t.length()){

if (s.charAt(indexS) == t.charAt(indexT)){

indexS++;

if (indexS == s.length()){

return true;

}

}

indexT++;

}

return false;

}

}

https://leetcode.com/problems/is-subsequence/discuss/87297/Java.-Only-2ms.-Much-faster-than-normal-2-pointers.

Difference between Method 1 and Method 2:

the origin code of func "indexOf" and "charAt". These two solution both traversed the char of String one by one to search

the first occurrence specific char.

The difference is that indexOf only call once function then traversed in "String.value[]" arr,

but we used multiple calling function "charAt" to get the value in "String.value[]" arr.

The time expense of calling function made the difference.

Good solution

Method 2: 2ms

class Solution {

public boolean isSubsequence(String s, String t) {

if (t.length() < s.length()){

return false;

}

int prev = 0;

int indexS = 0;

while (indexS < s.length()){

prev = t.indexOf(s.charAt(indexS), prev);

if (prev == - 1){

return false;

}

indexS++;

prev++;

}

return true;

}

}

# 393\_utf8.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A character in UTF8 can be from 1 to 4 bytes long, subjected to the following rules:

For 1-byte character, the first bit is a 0, followed by its unicode code.

For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10.

This is how the UTF-8 encoding would work:

Char. number range | UTF-8 octet sequence

(hexadecimal) | (binary)

--------------------+---------------------------------------------

0000 0000-0000 007F | 0xxxxxxx

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Given an array of integers representing the data, return whether it is a valid utf-8 encoding.

Note:

The input is an array of integers. Only the least significant 8 bits of each integer is used to store the data. This means

each integer represents only 1 byte of data.

Example 1:

data = [197, 130, 1], which represents the octet sequence: 11000101 10000010 00000001.

Return true.

It is a valid utf-8 encoding for a 2-bytes character followed by a 1-byte character.

Example 2:

data = [235, 140, 4], which represented the octet sequence: 11101011 10001100 00000100.

Return false.

The first 3 bits are all one's and the 4th bit is 0 means it is a 3-bytes character.

The next byte is a continuation byte which starts with 10 and that's correct.

But the second continuation byte does not start with 10, so it is invalid.

https://leetcode.com/problems/utf-8-validation/discuss/87485/O(n)-JAVA-solution-with-detailed-explaination

\* Method:

\* Start from index 0, determine each byte's type and check its validity.

\*

\* There are five kinds of valid byte type: 0\*\*, 10\*\*, 110\*\*,1110\*\* and 11110\*\*

\* Give them type numbers, 0, 1, 2, 3, 4 which are the index of the first 0 from left.

\* So, the index of the first 0 determines the byte type.

\*

\* if a byte belongs to one of them:

1 : if it is type 0, continue

2 : if it is type 2 or 3 or 4, check whether the following 1, 2, and 3 byte(s) are of type 1 or not

if not, return false;

\* else if a byte is type 1 or not of valid type, return false

Type 0: 0xxxxxxx

Type 1: 10xxxxxx

Type 2: 110xxxxx

Type 3: 1110xxxx

Type 4: 11110xxx

Palantir and FB phone interview Question

class Solution {

private int[] mask = {128, 64, 32, 16, 8};

public boolean validUtf8(int[] data) {

int len = data.length;

for (int i = 0; i < len; i++){

int num = data[i];

int type = getType(num);

if (type == 0){

continue;

}else if (type > 1 && type + i <= len){

while (type > 1){

i++;

int next = getType(data[i]);

if (next != 1){

return false;

}

type--;

}

}else{

return false;

}

}

return true;

}

private int getType(int num){

for (int i = 0; i < mask.length; i++){

if ((mask[i] & num) == 0){

return i;

}

}

return -1;

}

}

Palantir Phone interview: when input is string instead of int[] data

class Solution {

private int[] mask = {128, 64, 32, 16, 8};

public boolean validUtf8(String str) {

if (str == null || str.length() == 0){

return false;

}

if (str.length() % 8 != 0){

return false;

}

int len = str.length() / 8;

int[] data = new int[len];

int start = 0;

for (int i = 0; i < n; i++){

data[i] = Integer.parseInt(str.substring(start, start+8));

start += 8;

}

//below is the same as the original one

for (int i = 0; i < len; i++){

int num = data[i];

int type = getType(num);

if (type == 0){

continue;

}else if (type > 1 && type + i <= len){

while (type > 1){

i++;

int next = getType(data[i]);

if (next != 1){

return false;

}

type--;

}

}else{

return false;

}

}

return true;

}

private int getType(int num){

for (int i = 0; i < mask.length; i++){

if ((mask[i] & num) == 0){

return i;

}

}

return -1;

}

}

# 394\_CoinsLine.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are n coins in a line. Two players take turns to take one or two coins from right side

until there are no more coins left. The player who take the last coin wins.

Could you please decide the first play will win or lose?

Have you met this question in a real interview?

Example

n = 1, return true.

n = 2, return true.

n = 3, return false.

n = 4, return true.

n = 5, return true.

Challenge

O(n) time and O(1) memory

http://www.jiuzhang.com/solutions/coins-in-a-line/

dp[i] denotes if the first player will win when there are i coins left

public class Solution {

/\*\*

\* @param n: An integer

\* @return: A boolean which equals to true if the first player will win

\*/

public boolean firstWillWin(int n) {

boolean[] dp = new boolean[n+1];

if (n <= 0){

return false;

}else if (n == 1){

return true;

}else if (n ==2){

return true;

}else if (n == 3){

return false;

}

dp[0] = false;

dp[1] = true;

dp[2] = true;

dp[3] = false;

for (int i = 4; i <= n; i++){

dp[i] = (dp[i-4] && dp[i-3]) || (dp[i-3] && dp[i-2]);

}

return dp[n];

}

}

# 394\_Decode.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an encoded string, return it's decoded string.

The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets

is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square

brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that

digits are only for those repeat numbers, k. For example, there won't be input like 3a or 2[4].

Examples:

s = "3[a]2[bc]", return "aaabcbc".

s = "3[a2[c]]", return "accaccacc".

s = "2[abc]3[cd]ef", return "abcabccdcdcdef".

Method 1:

class Solution {

public String decodeString(String s) {

if (s == null || s.length() == 0){

return s;

}

Stack<Integer> numStack = new Stack<>();

Stack<String> strStack = new Stack<>();

int len = s.length();

int digit = 0;

StringBuilder sb = new StringBuilder();

for (int i = 0; i < len; i++){

char c = s.charAt(i);

if (Character.isDigit(c)){

digit = 10 \* digit + (int) (c - '0');

}else if (Character.isLetter(c)){

sb.append(c);

}else if (c == '['){

numStack.push(digit);

strStack.push(sb.toString());

sb = new StringBuilder();

digit = 0;

}else if (c == ']'){

int num = numStack.pop();

String prev = strStack.pop();

String str = sb.toString();

sb = new StringBuilder(prev);

while (num > 0) {

sb.append(str);

num--;

}

}

}

return sb.toString();

}

}

Best solution:

Can also use Stack<StringBuilder>

class Solution {

public String decodeString(String s) {

if (s == null || s.length() == 0){

return s;

}

Stack<Integer> numStack = new Stack<>();

Stack<StringBuilder> strStack = new Stack<>();

int len = s.length();

int digit = 0;

StringBuilder sb = new StringBuilder();

for (int i = 0; i < len; i++){

char c = s.charAt(i);

if (Character.isDigit(c)){

digit = 10 \* digit + (int) (c - '0');

}else if (Character.isLetter(c)){

sb.append(c);

}else if (c == '['){

numStack.push(digit);

strStack.push(sb);

sb = new StringBuilder();

digit = 0;

}else if (c == ']'){

int num = numStack.pop();

StringBuilder prev = strStack.pop();

StringBuilder str = sb;

sb = new StringBuilder(prev);

while (num > 0) {

sb.append(str);

num--;

}

}

}

return sb.toString();

}

}

class Solution {

public String decodeString(String s) {

Stack<Integer> numStack = new Stack<>();

Stack<String> strStack = new Stack<>();

int curr = 0;

String str = "";

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (Character.isDigit(c)){

curr = curr \* 10 + (int)(c - '0');

}else if (c == '['){

numStack.push(curr);

strStack.push(str);

curr = 0;

str = "";

}else if (Character.isLetter(c)){

str += c;

}else if (c == ']'){

int num = numStack.pop();

String prevStr = strStack.pop();

StringBuilder sb = new StringBuilder();

while (num > 0){

sb.append(str);

num--;

}

str = prevStr + sb.toString();

}

}

return str;

}

}

# 395\_CoinsII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are n coins with different value in a line. Two players take turns to take one or two coins

from left side until there are no more coins left. The player who take the coins with the most value wins.

Could you please decide the first player will win or lose?

Have you met this question in a real interview?

Example

Given values array A = [1,2,2], return true.

Given A = [1,2,4], return false.

http://www.jiuzhang.com/solutions/coins-in-a-line-ii/

Method 1:

State:

• dp[i] 现在还剩i个硬币,现在先手取硬币的人最后最多取硬币价值

Function:

• i 是所有硬币数目

• coin[n-i] 表示倒数第i个硬币

• dp[i] = max(min(dp[i-2], dp[i-3])+coin[n-i] ) , (min(dp[i-3],dp[i-4])+coin[n-i]+coin[n-i+1] )

Initialize:

• dp[0] = 0

• dp[1] = coin[i-1]

• dp[2] = coin[i-2] + coin[i-1] • dp[3] = coin[i-2] + coin[i-3]

Answer:

• dp[n] > sum/2

Take 5 coins as an example

Note that opponent always wants to minimize yours

you want to maximize yours

public class Solution {

/\*

\* @param values: a vector of integers

\* @return: a boolean which equals to true if the first player will win

\*/

public boolean firstWillWin(int[] values) {

int n = values.length;

int[] f = new int[n+1];

if (n <= 0){

return false;

}else if (n == 1){

return true;

}else if (n == 2){

return true;

}else if (n == 3){

return (values[0] + values[1]) > values[2];

}

f[0] = 0;

f[1] = values[n-1];

f[2] = values[n-1] + values[n-2];

f[3] = values[n-2] + values[n-3];

for (int i = 4; i <= n; i++){

f[i] = Math.max(Math.min(f[i-2], f[i-3]) + values[n-i],

Math.min(f[i-3], f[i-4]) + values[n-i] + values[n-i+1]);

}

int sum = 0;

for (int i = 0; i < n; i++){

sum += values[i];

}

return 2\* f[n] > sum;

}

}

# 396\_RotateFunction.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers A and let n to be its length.

Assume Bk to be an array obtained by rotating the array A k positions clock-wise,

we define a "rotation function" F on A as follow:

F(k) = 0 \* Bk[0] + 1 \* Bk[1] + ... + (n-1) \* Bk[n-1].

Calculate the maximum value of F(0), F(1), ..., F(n-1).

Note:

n is guaranteed to be less than 105.

Example:

A = [4, 3, 2, 6]

F(0) = (0 \* 4) + (1 \* 3) + (2 \* 2) + (3 \* 6) = 0 + 3 + 4 + 18 = 25

F(1) = (0 \* 6) + (1 \* 4) + (2 \* 3) + (3 \* 2) = 0 + 4 + 6 + 6 = 16

F(2) = (0 \* 2) + (1 \* 6) + (2 \* 4) + (3 \* 3) = 0 + 6 + 8 + 9 = 23

F(3) = (0 \* 3) + (1 \* 2) + (2 \* 6) + (3 \* 4) = 0 + 2 + 12 + 12 = 26

So the maximum value of F(0), F(1), F(2), F(3) is F(3) = 26.

This is essentially a Math problem.

Consider the array [ A, B, C, D ] with very simple coefficients as following:

f(0) = 0A + 1B + 2C + 3D

f(1) = 3A + 0B + 1C + 2D

f(2) = 2A + 3B + 0C + 1D

f(3) = 1A + 2B + 3C + 0D

We can see from above that:

f(0) -> f(1) -> f(2) -> f(3)

f(i) = f(i - 1) - SUM(A -> D) + N \* A[i - 1]

class Solution {

public int maxRotateFunction(int[] A) {

int n = A.length;

int sum = 0;

int candidate = 0;

for (int i = 0; i < A.length; i++){

sum += A[i];

candidate += A[i] \* i;

}

int best = candidate;

for (int i = 0; i < A.length; i++){

candidate = candidate - sum + n \* A[i];

best = Math.max(best, candidate);

}

return best;

}

}

# 397\_Integer.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer n and you can do operations as follow:

If n is even, replace n with n/2.

If n is odd, you can replace n with either n + 1 or n - 1.

What is the minimum number of replacements needed for n to become 1?

Example 1:

Input:

8

Output:

3

Explanation:

8 -> 4 -> 2 -> 1

Example 2:

Input:

7

Output:

4

Explanation:

7 -> 8 -> 4 -> 2 -> 1

or

7 -> 6 -> 3 -> 2 -> 1

Method 1:

class Solution {

public int integerReplacement(int n) {

if (n == Integer.MAX\_VALUE){ //n = 2^31 -1

return 32;

}

int count = 0;

while (n > 1){

if (n % 2 == 0){

n /= 2;

}else{

if ((n + 1) % 4 == 0 && n != 3){

n++;

}else{

n--;

}

}

count++;

}

return count;

}

}

Method: DP TLE

class Solution {

public int integerReplacement(int n) {

if (n == 1){

return 0;

}

int[] dp = new int[n+1];

for (int i = 2; i <= n; i++){

if (i % 2 == 0){

dp[i] = dp[i/2] + 1;

}else{

dp[i] = Math.min(dp[i-1], dp[(i+1)/2] + 1) + 1;

}

}

return dp[n];

}

}

https://leetcode.com/problems/integer-replacement/discuss/87928/Java-12-line-4(5)ms-iterative-solution-with-explanations.-No-other-data-structures.

# 398\_RandomPick.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers with possible duplicates, randomly output the index of a given target number.

You can assume that the given target number must exist in the array.

Note:

The array size can be very large. Solution that uses too much extra space will not pass the judge.

Example:

int[] nums = new int[] {1,2,3,3,3};

Solution solution = new Solution(nums);

// pick(3) should return either index 2, 3, or 4 randomly. Each index should have equal probability of returning.

solution.pick(3);

// pick(1) should return 0. Since in the array only nums[0] is equal to 1.

solution.pick(1);

class Solution {

private int[] nums;

private Random random;

public Solution(int[] nums) {

this.nums = nums;

random = new Random();

}

public int pick(int target) {

int ans = 0;

int count = 0;

for (int i = 0; i < nums.length; i++){

/\* if (nums[i] == target && count == random.nextInt(++count)){

ans = i;

}\*/

if (nums[i] == target){

if (count == random.nextInt(count + 1)){

ans = i;

}

count++;

}

}

return ans;

}

}

/\*\*

\* Your Solution object will be instantiated and called as such:

\* Solution obj = new Solution(nums);

\* int param\_1 = obj.pick(target);

\*/

# 399\_Evaluate.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Equations are given in the format A / B = k, where A and B are variables represented as strings, and k is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return -1.0.

Example:

Given a / b = 2.0, b / c = 3.0.

queries are: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .

return [6.0, 0.5, -1.0, 1.0, -1.0 ].

The input is: vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries , where equations.size() == values.size(), and the values are positive. This represents the equations. Return vector<double>.

According to the example above:

equations = [ ["a", "b"], ["b", "c"] ],

values = [2.0, 3.0],

queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].

The input is always valid. You may assume that evaluating the queries will result in no division by zero and there is no contradiction.

Method: Graph

class Solution {

public double[] calcEquation(String[][] equations, double[] values, String[][] queries) {

int N = values.length;

double[] res = new double[queries.length];

Map<String, Map<String, Double>> graph = new HashMap<>();

//build graph

for (int i = 0; i < N; i++){

String[] equation = equations[i];

String dividend = equation[0];

String divisor = equation[1];

if (!graph.containsKey(dividend)){

graph.put(dividend, new HashMap<String, Double>());

graph.get(dividend).put(dividend, 1.0);

}

graph.get(dividend).put(divisor, values[i]);

if (!graph.containsKey(divisor)){

graph.put(divisor, new HashMap<String, Double>());

graph.get(divisor).put(divisor, 1.0);

}

graph.get(divisor).put(dividend, 1.0/values[i]);

}

//dfs to calculate queries

for (int i = 0; i < queries.length; i++){

String[] query = queries[i];

String dividend = query[0];

String divisor = query[1];

if (!graph.containsKey(dividend) || !graph.containsKey(divisor)){

res[i] = -1.0;

}else{

res[i] = dfs(graph, dividend, divisor, new HashSet<String>());

}

}

return res;

}

private double dfs(Map<String, Map<String, Double>> graph, String dividend, String divisor, Set<String> seen){

if (dividend.equals(divisor)){

return 1.0;

}

seen.add(dividend);

Map<String, Double> map = graph.get(dividend);

for (String key : map.keySet()){

if (!seen.contains(key)){

double val = dfs(graph, key, divisor, seen);

if (val > 0){

return map.get(key) \* val;

}

}

}

return -1.0;

}

}

class Solution {

public double[] calcEquation(String[][] equations, double[] values, String[][] queries) {

int N = queries.length;

//build graph

double[] res = new double[N];

Map<String, Map<String, Double>> graph = new HashMap<>();

for (int i = 0; i < equations.length; i++){

String dividend = equations[i][0];

String divisor = equations[i][1];

if (!graph.containsKey(dividend)){

graph.put(dividend, new HashMap<>());

graph.get(dividend).put(dividend, 1.0);

}

graph.get(dividend).put(divisor, values[i]);

if (!graph.containsKey(divisor)){

graph.put(divisor, new HashMap<>());

graph.get(divisor).put(divisor, 1.0);

}

graph.get(divisor).put(dividend, 1.0/values[i]);

}

//dfs

for (int i = 0; i < N; i++){

String dividend = queries[i][0];

String divisor = queries[i][1];

if (!graph.containsKey(dividend) || !graph.containsKey(divisor)){

res[i] = -1.0;

}else{

Set<String> seen = new HashSet<>();

seen.add(dividend);

res[i] = dfs(graph, dividend, divisor, seen);

}

}

return res;

}

private double dfs(Map<String, Map<String, Double>> graph, String dividend, String divisor, Set<String> seen){

if (dividend.equals(divisor)){

return 1.0;

}

Map<String, Double> map = graph.get(dividend);

for (String key : map.keySet()){

if (!seen.contains(key)){

seen.add(key);

double val = dfs(graph, key, divisor, seen);

if (val > 0){

return map.get(key) \* val;

}

}

}

return -1.0;

}

}

Best solution:

class Solution {

public double[] calcEquation(String[][] equations, double[] values, String[][] queries) {

int N = queries.length;

Map<String, Map<String, Double>> graph = new HashMap<>();

for (int i = 0; i < values.length; i++){

String dividend = equations[i][0];

String divisor = equations[i][1];

if (!graph.containsKey(dividend)){

graph.put(dividend, new HashMap<>());

}

if (!graph.containsKey(divisor)){

graph.put(divisor, new HashMap<>());

}

graph.get(dividend).put(divisor, values[i]);

graph.get(divisor).put(dividend, 1.0 / values[i]);

}

double[] res = new double[N];

for (int i = 0; i < N; i++){

String dividend = queries[i][0];

String divisor = queries[i][1];

if (!graph.containsKey(dividend) || !graph.containsKey(divisor)){

res[i] = -1.0;

}else{

Set<String> seen = new HashSet<>();

res[i] = dfs(graph, dividend, divisor, seen);

}

}

return res;

}

private double dfs(Map<String, Map<String, Double>> graph, String start, String end, Set<String> seen){

if (start.equals(end)){

return 1.0;

}

seen.add(start);

Map<String, Double> map = graph.get(start);

for (String next : map.keySet()){

if (!seen.contains(next)){

double val = dfs(graph, next, end, seen);

if (val > 0){

return map.get(next) \* val;

}

}

}

seen.remove(start);

return -1.0;

}

}

# 4\_MedianofTwoSortedArrays.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are two sorted arrays nums1 and nums2 of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

Example 1:

nums1 = [1, 3]

nums2 = [2]

The median is 2.0

Example 2:

nums1 = [1, 2]

nums2 = [3, 4]

The median is (2 + 3)/2 = 2.5

Best solution: O(log(m + n)), O(log(k))

class Solution {

public double findMedianSortedArrays(int[] nums1, int[] nums2) {

int m = nums1.length;

int n = nums2.length;

if ((m+n) % 2 == 0){

return (findKth(nums1, 0, nums2, 0, (m+n)/2) + findKth(nums1, 0, nums2, 0, (m+n)/2 + 1)) / 2.0;

}

return (double)findKth(nums1, 0, nums2, 0, (m+n)/2 + 1);

}

private int findKth(int[] nums1, int start1, int[] nums2, int start2, int k){

if (start1 >= nums1.length){

return nums2[start2 + k - 1];

}

if (start2 >= nums2.length){

return nums1[start1 + k - 1];

}

if (k == 1){

return Math.min(nums1[start1], nums2[start2]);

}

int mid1 = start1 + k/2 - 1 < nums1.length ? nums1[start1 + k/2 - 1] : Integer.MAX\_VALUE;

int mid2 = start2 + k/2 - 1 < nums2.length ? nums2[start2 + k/2 - 1] : Integer.MAX\_VALUE;

if (mid1 < mid2){//median must be within the 2nd half of nums1 and 1st half of nums2

return findKth(nums1, start1 + k/2, nums2, start2, k - k/2);

}else{

return findKth(nums1, start1, nums2, start2 + k/2, k - k /2);

}

}

}

Method 1: O(log(m + n)), O(log(k))

class Solution {

public double findMedianSortedArrays(int[] nums1, int[] nums2) {

int n = nums1.length + nums2.length;

if (n % 2 == 1){

return findKth(nums1, 0, nums2, 0, n/2 + 1);

}else{

return (findKth(nums1, 0, nums2, 0, n/2) + findKth(nums1, 0, nums2, 0, n/2 + 1))/2.0;

}

}

private double findKth(int[] A, int startA, int[] B, int startB, int k){

if (startA >= A.length){

return B[startB + k - 1];

}

if (startB >= B.length){

return A[startA + k - 1];

}

if (k == 1){

return Math.min(A[startA], B[startB]);

}

int Amid = startA + k/2 - 1 < A.length ? A[startA + k/2 - 1] : Integer.MAX\_VALUE;

int Bmid = startB + k/2 - 1 < B.length ? B[startB + k/2 - 1] : Integer.MAX\_VALUE;

if (Amid < Bmid){

return findKth(A, startA + k/2 , B, startB, k - k/2);

}else{

return findKth(A, startA, B, startB + k/2, k - k/2);

}

}

}

Method 2: Time complexity O(k)

public class Solution {

/\*

\* @param A: An integer array

\* @param B: An integer array

\* @return: a double whose format is \*.5 or \*.0

\*/

public double findMedianSortedArrays(int[] A, int[] B) {

int len = A.length + B.length;

if (len % 2 == 0){

return (findKth(A, B, len /2) + findKth(A, B, len / 2 + 1))/2.0;

}else{

return findKth(A, B, len / 2 + 1);

}

}

private int findKth(int[] A, int[] B, int k){

int i = 0, j = 0;

int index = 0;

while (i < A.length && j < B.length){

if (A[i] < B[j]){

index++;

if (index == k){

return A[i];

}else{

i++;

}

}else{

index++;

if (index == k){

return B[j];

}else{

j++;

}

}

}

if (i < A.length){

return A[i + (k - index - 1)];

}else{

return B[j + (k - index - 1)];

}

}

}

# 40\_CombinationSumII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers

sums to T.

Each number in C may only be used once in the combination.

Note:

All numbers (including target) will be positive integers.

The solution set must not contain duplicate combinations.

For example, given candidate set [10, 1, 2, 7, 6, 1, 5] and target 8,

A solution set is:

[

[1, 7],

[1, 2, 5],

[2, 6],

[1, 1, 6]

]

class Solution {

public List<List<Integer>> combinationSum2(int[] candidates, int target) {

List<List<Integer>> result = new ArrayList<>();

if (candidates == null || candidates.length == 0){

return result;

}

Arrays.sort(candidates);

dfs(result, new ArrayList<Integer>(), candidates, target, 0);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int[] candidates, int target, int start){

if (target == 0){

result.add(new ArrayList<Integer>(item));

return;

}

for (int i = start; i < candidates.length; i++){

if (candidates[i] > target){

break;

}

if (i != start && candidates[i] == candidates[i-1]){

continue;

}

item.add(candidates[i]);

dfs(result, item, candidates, target - candidates[i], i + 1);

item.remove(item.size() - 1);

}

}

}

class Solution {

public List<List<Integer>> combinationSum2(int[] candidates, int target) {

List<List<Integer>> res = new ArrayList<>();

Arrays.sort(candidates);

boolean[] visited = new boolean[candidates.length];

dfs(res, new ArrayList<Integer>(), candidates, target, visited, 0);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, int[] candidates, int target, boolean[] visited, int start){

if (target == 0){

res.add(new ArrayList<Integer>(item));

return;

}

for (int i = start; i < candidates.length; i++){

if (candidates[i] > target){

break;

}

if (i > 0 && candidates[i] == candidates[i-1] && !visited[i-1]){

continue;

}

item.add(candidates[i]);

visited[i] = true;

dfs(res, item, candidates, target - candidates[i], visited, i + 1);

visited[i] = false;

item.remove(item.size() - 1);

}

}

}

# 40\_ImplementQueuebyStacks.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

As the title described, you should only use two stacks to implement a queue's actions.

The queue should support push(element), pop() and top() where pop is pop the first(a.k.a front) element in the queue.

Both pop and top methods should return the value of first element.

Have you met this question in a real interview? Yes

Example

push(1)

pop() // return 1

push(2)

push(3)

top() // return 2

pop() // return 2

public class MyQueue {

private Stack<Integer> stack1;

private Stack<Integer> stack2;

public MyQueue() {

stack1 = new Stack<Integer>();

stack2 = new Stack<Integer>();

}

/\*

\* @param element: An integer

\* @return: nothing

\*/

public void push(int element) {

stack1.push(element);

}

/\*

\* @return: An integer

\*/

public int pop() {

if (stack2.isEmpty()){

moveStack1To2();

}

return stack2.pop();

}

/\*

\* @return: An integer

\*/

public int top() {

if (stack2.empty()){

moveStack1To2();

}

return stack2.peek();

}

private void moveStack1To2(){

while (!stack1.empty()){

stack2.push(stack1.pop());

}

}

}

# 400\_Nth.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the nth digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

Note:

n is positive and will fit within the range of a 32-bit signed integer (n < 231).

Example 1:

Input:

3

Output:

3

Example 2:

Input:

11

Output:

0

Explanation:

The 11th digit of the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... is a 0, which is part of the number 10.

class Solution {

public int findNthDigit(int n) {

int digits = 1;

long factor = 9; // must use long because it may overflow

int start = 1;

while (n > digits \* factor){

n -= digits \* factor;

factor \*= 10;

digits += 1;

start \*= 10;

}

int pos = (n - 1) / digits; // -1 to convert to base 0

int num = start + pos;

int offset = (n- 1) % digits;

String s = Integer.toString(num);

return (int)(s.charAt(offset) -'0');

}

}

# 401\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A binary watch has 4 LEDs on the top which represent the hours (0-11), and the 6 LEDs on the bottom represent the minutes (0-59).

Each LED represents a zero or one, with the least significant bit on the right.

For example, the above binary watch reads "3:25".

Given a non-negative integer n which represents the number of LEDs that are currently on, return all possible times the watch could represent.

Example:

Input: n = 1

Return: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0:32"]

Note:

The order of output does not matter.

The hour must not contain a leading zero, for example "01:00" is not valid, it should be "1:00".

The minute must be consist of two digits and may contain a leading zero, for example "10:2" is not valid, it should be "10:02

Method 1: permutation

class Solution {

public List<String> readBinaryWatch(int num) {

List<String> res = new ArrayList<>();

int[] weightHrs = {8, 4, 2, 1};

int[] weightMins = {32, 16, 8, 4, 2, 1};

for (int i = 0; i <= num; i++){

List<Integer> hours = generateDigits(weightHrs, i);

List<Integer> mins = generateDigits(weightMins, num - i);

for (int hour : hours){

if (hour >= 12){

continue;

}

for (int min : mins){

if (min >= 60){

continue;

}

res.add(hour + ":" + (min >= 10 ? min : "0" + min));

}

}

}

return res;

}

private List<Integer> generateDigits(int[] weights, int count){

List<Integer> res = new ArrayList<>();

dfs(res, weights, count, 0, 0);

return res;

}

private void dfs(List<Integer> res, int[] weights, int count, int index, int sum){

if (count == 0){

res.add(sum);

return;

}

for (int i = index; i < weights.length; i++){

dfs(res, weights, count - 1, i + 1, sum + weights[i]);

}

}

}

Method 2: Best solution

recursion

class Solution {

public final int[] weights = {1, 2, 4, 8, 32, 16, 8, 4, 2, 1};//order does not matter as long as the first 4 are hours

public List<String> readBinaryWatch(int num) {

List<String> res = new ArrayList<>();

dfs(res, num, 0, 0, 0);

return res;

}

private void dfs(List<String> res, int num, int index, int hour, int min){

if (hour >= 12 || min >= 60){

return;

}

if (num == 0){

res.add(hour + ":" + (min >= 10 ? min : "0" + min));

return;

}

for (int i = index; i < weights.length; i++){

if (i < 4){

dfs(res, num - 1, i + 1, hour + weights[i], min);

}else{

dfs(res, num - 1, i + 1, hour, min + weights[i]);

}

}

}

}

class Solution {

public final int[] weights = {1, 2, 4, 8, 1, 2, 4, 8, 16, 32};

public List<String> readBinaryWatch(int num) {

List<String> res = new ArrayList<>();

dfs(res, num, 0, 0, 0);

return res;

}

private void dfs(List<String> res, int num, int index, int hour, int minute){

if (num == 0){

res.add(hour + ":" + (minute < 10 ? "0" + minute : minute));

}

for (int i = index; i < weights.length; i++){

if (i < 4){

if (hour + weights[i] >= 12){

continue;

}

dfs(res, num - 1, i + 1, hour + weights[i], minute);

}else{

if (minute + weights[i] >= 60){

continue;

}

dfs(res, num - 1, i + 1, hour, minute + weights[i]);

}

}

}

}

backtracking:

class Solution {

public final int[] weights = {8, 4, 2, 1, 32, 16, 8, 4, 2, 1};

public List<String> readBinaryWatch(int num) {

List<String> res = new ArrayList<>();

dfs(res, num, 0, 0, 0);

return res;

}

private void dfs(List<String> res, int num, int index, int hour, int min){

if (hour >= 12 || min >= 60){

return;

}

if (num == 0){

res.add(hour + ":" + (min >= 10 ? min : "0" + min));

return;

}

for (int i = index; i < weights.length; i++){

if (i < 4){

hour += weights[i];

dfs(res, num - 1, i + 1, hour, min);

hour -= weights[i];

}else{

min += weights[i];

dfs(res, num - 1, i + 1, hour, min);

min -= weights[i];

}

}

}

}

# 401\_KthSmallestNumberinSortedMatrix.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the kth smallest number in at row and column sorted matrix.

Have you met this question in a real interview? Yes

Example

Given k = 4 and a matrix:

[

[1 ,5 ,7],

[3 ,7 ,8],

[4 ,8 ,9],

]

return 5

Method 1: BFS

Time complexity: O(k\*logk)

Space complexity: O(m\*n)

public class Solution {

/\*

\* @param matrix: a matrix of integers

\* @param k: An integer

\* @return: the kth smallest number in the matrix

\*/

private class Pair{

private int x;

private int y;

private int val;

private Pair(int x, int y, int val){

this.x = x;

this.y = y;

this.val = val;

}

}

public int kthSmallest(int[][] matrix, int k) {

Queue<Pair> minHeap = new PriorityQueue<Pair>(k, new Comparator<Pair>(){

public int compare(Pair a, Pair b){

return a.val - b.val;

}

});

int[] dx = {0, 1};

int[] dy = {1, 0};

int m = matrix.length;

int n = matrix[0].length;

boolean[][] visited = new boolean[m][n];

minHeap.offer(new Pair(0, 0, matrix[0][0]));

for (int i = 1; i < k; i++){

Pair cur = minHeap.poll();

for (int j = 0; j < 2; j++){

int x = cur.x + dx[j];

int y = cur.y + dy[j];

if (x < m && y < n && !visited[x][y]){

minHeap.offer(new Pair(x, y, matrix[x][y]));

visited[x][y] = true;

}

}

}

return minHeap.poll().val;

}

}

Method 2: Best solution

https://leetcode.com/problems/kth-smallest-element-in-a-sorted-matrix/discuss/85177/Java-1ms-nlog(max-min)-solution

Time complexity: O(nlog(max-min))

Space complexity: O(1)

class Solution {

public int kthSmallest(int[][] matrix, int k) {

int m = matrix.length;

int n = matrix[0].length;

int low = matrix[0][0];

int high = matrix[m-1][n-1];

while (low <= high){

int mid = low + (high - low) / 2;

int count = getLessEqual(matrix, mid);

if (count <= k - 1){

low = mid + 1; //find the last one to meet the condition

}else{

high = mid - 1;

}

}

return low;

}

private int getLessEqual(int[][] matrix, int val){

int res = 0;

int m = matrix.length;

int n = matrix[0].length;

int i = m - 1;

int j = 0;

while (i >= 0 && j < n){

if (matrix[i][j] > val){

i--;

}else{

res += i + 1;

j++;

}

}

return res;

}

}

class Solution {

public int kthSmallest(int[][] matrix, int k) {

int n = matrix.length;

int start = matrix[0][0];

int end = matrix[n-1][n-1];

while (start <= end){

int mid = start + (end - start) / 2;

int num = getLessEqual(mid, matrix);

if (num <= k - 1){

start = mid + 1;

}else{

end = mid - 1;

}

}

return start;

}

private int getLessEqual(int target, int[][] matrix){

int n = matrix.length;

int r = n - 1;

int c = 0;

int count = 0;

while (r >= 0 && c < n){

if (matrix[r][c] <= target){

count += r + 1;

c++;

}else{

r--;

}

}

return count;

}

}

class Solution {

public int kthSmallest(int[][] matrix, int k) {

int m = matrix.length;

int n = matrix[0].length;

int low = matrix[0][0];

int high = matrix[m-1][n-1];

while (low + 1 < high){

int mid = low + (high - low) / 2;

int count = getLessEqual(matrix, mid);

if (count <= k - 1){

low = mid; //find the last one to meet the condition

}else{

high = mid;

}

}

if (getLessEqual(matrix, low) <= k - 1){

return high;

}

return low;

}

private int getLessEqual(int[][] matrix, int val){

int res = 0;

int m = matrix.length;

int n = matrix[0].length;

int i = m - 1;

int j = 0;

while (i >= 0 && j < n){

if (matrix[i][j] > val){

i--;

}else{

res += i + 1;

j++;

}

}

return res;

}

}

# 402\_Continuous.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array, find a continuous subarray where the sum of numbers is the biggest.

Your code should return the index of the first number and the index of the last number.

(If their are duplicate answer, return anyone)

Have you met this question in a real interview?

Example

Give [-3, 1, 3, -3, 4], return [1,4].

https://www.lintcode.com/problem/continuous-subarray-sum/description

Similar to https://github.com/optimisea/Leetcode/blob/master/Java/53\_MaximumSubarray.java

local denotes the maximum value of continuous subarray that ends at first i element

public class Solution {

/\*

\* @param A: An integer array

\* @return: A list of integers includes the index of the first number and the index of the last number

\*/

public List<Integer> continuousSubarraySum(int[] A) {

List<Integer> result = new ArrayList<>();

result.add(0, 0);

result.add(1, 0);

int local = 0;

int global = Integer.MIN\_VALUE;

int left = 0;

int right = 0;

for (int i = 0; i < A.length; i++){

if (local < 0){

local = A[i];

left = i;

right = i;

}else{

local += A[i];

right = i;

}

if (local >= global){

global = local;

result.set(0, left);

result.set(1, right);

}

}

return result;

}

}

public class Solution {

/\*

\* @param A: An integer array

\* @return: A list of integers includes the index of the first number and the index of the last number

\*/

public List<Integer> continuousSubarraySum(int[] A) {

List<Integer> res = new ArrayList<>();

res.add(0, 0);

res.add(1, 0);

int global = Integer.MIN\_VALUE;

int local = 0;

int start = 0;

int end = 0;

for (int i = 0; i < A.length; i++){

if (local < 0){

start = i;

local = A[i];

}else{

local += A[i];

}

if (local > global){

end = i;

global = local;

res.set(0, start);

res.set(1, end);

}

}

return res;

}

}

public class Solution {

/\*

\* @param A: An integer array

\* @return: A list of integers includes the index of the first number and the index of the last number

\*/

public List<Integer> continuousSubarraySum(int[] A) {

List<Integer> res = new ArrayList<>();

res.add(0);

res.add(1);

int sum = 0;

int max = Integer.MIN\_VALUE;

int start = 0;

int end = 0;

for (int i = 0; i < A.length; i++){

sum += A[i];

if (sum > max){

max = sum;

end = i;

res.set(0, start);

res.set(1, end);

}

if (sum < 0){

sum = 0;

start = i + 1;

}

}

return res;

}

}

# 402\_Remove.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-negative integer num represented as a string, remove k digits from the number so that the new number is the smallest possible.

Note:

The length of num is less than 10002 and will be ≥ k.

The given num does not contain any leading zero.

Example 1:

Input: num = "1432219", k = 3

Output: "1219"

Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.

Example 2:

Input: num = "10200", k = 1

Output: "200"

Explanation: Remove the leading 1 and the number is 200. Note that the output must not contain leading zeroes.

Example 3:

Input: num = "10", k = 2

Output: "0"

Explanation: Remove all the digits from the number and it is left with nothing which is 0.

// k keeps track of how many characters we can remove

// if the previous character in stk is larger than the current one

// then removing it will get a smaller number

// but we can only do so when k is larger than 0

class Solution {

public String removeKdigits(String num, int k) {

int digits = num.length() - k;

char[] res = new char[num.length()]; // must define as num.length(), e.g. 123456, k = 1

int top = 0;

for (int i = 0; i < num.length(); i++){

char c = num.charAt(i);

while (top > 0 && k > 0 && res[top-1] > c){

top--;

k--;

}

res[top++] = c;

}

int idx = 0;

while (idx < digits && res[idx] == '0'){

idx++;

}

if (idx == digits){

return "0";

}

return new String(res, idx, digits - idx);

}

}

class Solution {

public String removeKdigits(String num, int k) {

Stack<Character> stack = new Stack<>();

for (int i = 0; i < num.length(); i++){

char c = num.charAt(i);

while (!stack.isEmpty() && k > 0 && stack.peek() > c){

stack.pop();

k--;

}

stack.push(c);

}

StringBuilder sb = new StringBuilder();

boolean isLeadZero = true;

for (Character c : stack){

if (c == '0' && isLeadZero){

continue;

}

isLeadZero = false;

if (sb.length() == stack.size() - k){

break;

}

sb.append(c);

}

return sb.length() == 0 ? "0" : sb.toString();

}

}

class Solution {

public String removeKdigits(String num, int k) {

Stack<Character> stack = new Stack<>(); //montonic increasing stack

for (int i = 0; i < num.length(); i++){

char c = num.charAt(i);

while (!stack.isEmpty() && k > 0 && stack.peek() > c){

stack.pop();

k--;

}

stack.push(c);

}

while (k > 0){//corner case for monotonic increasing e.g, 123456

stack.pop();

k--;

}

StringBuilder sb = new StringBuilder();

while (!stack.isEmpty()){

sb.append(stack.pop());

}

sb.reverse();

while (sb.length() > 1 && sb.charAt(0) == '0'){

sb.deleteCharAt(0);

}

return sb.length() == 0 ? "0" : sb.toString();

}

}

Best Solution:

class Solution {

public String removeKdigits(String num, int k) {

Stack<Integer> stack = new Stack<>();

char[] charArr = num.toCharArray();

int n = charArr.length;

for (int i = 0; i < n; i++){

//greedy: pop the larger one

int digit = (int)(charArr[i] - '0');

while (!stack.isEmpty() && k > 0 && digit < stack.peek()){

stack.pop();

k--;

}

stack.push(digit);

}

while (k > 0){

stack.pop();

k--;

}

StringBuilder sb = new StringBuilder();

while (!stack.isEmpty()){

sb.append(stack.pop());

}

sb.reverse(); //sb.reverse() is assigned back to sb

int i = 0;

while (sb.length() > 0 && sb.charAt(0) == '0'){

sb.deleteCharAt(0);

}

return sb.length() == 0 ? "0" : sb.toString();

}

}

# 403\_Frog.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A frog is crossing a river. The river is divided into x units and at each unit there may or may not exist a stone. The frog can

jump on a stone, but it must not jump into the water.

Given a list of stones' positions (in units) in sorted ascending order, determine if the frog is able to cross the river by landing

on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.

If the frog's last jump was k units, then its next jump must be either k - 1, k, or k + 1 units. Note that the frog can only jump

in the forward direction.

Note:

The number of stones is ≥ 2 and is < 1,100.

Each stone's position will be a non-negative integer < 231.

The first stone's position is always 0.

Example 1:

[0,1,3,5,6,8,12,17]

There are a total of 8 stones.

The first stone at the 0th unit, second stone at the 1st unit,

third stone at the 3rd unit, and so on...

The last stone at the 17th unit.

Return true. The frog can jump to the last stone by jumping

1 unit to the 2nd stone, then 2 units to the 3rd stone, then

2 units to the 4th stone, then 3 units to the 6th stone,

4 units to the 7th stone, and 5 units to the 8th stone.

Example 2:

[0,1,2,3,4,8,9,11]

Return false. There is no way to jump to the last stone as

the gap between the 5th and 6th stone is too large.

https://leetcode.com/problems/frog-jump/discuss/88824/Very-easy-to-understand-JAVA-solution-with-explanations

Time complexity: O(kN)

Space complexity: O(N)

class Solution {

public boolean canCross(int[] stones) {

Map<Integer, Set<Integer>> map = new HashMap<>();//store the steps that can jump at stone[i]

int n = stones.length;

for (int i = 1; i < n - 1; i++){

map.put(stones[i], new HashSet<>());

}

map.put(stones[0], new HashSet<>());

map.get(stones[0]).add(1);

for (int i = 0; i < n - 1; i++){

for (int step : map.get(stones[i])){

int reach = stones[i] + step;

if (reach == stones[n-1]){

return true;

}

if (map.containsKey(reach)){

Set<Integer> set = map.get(reach);

set.add(step);

if (step > 1){

set.add(step-1);

}

set.add(step+1);

}

}

}

return false;

}

}

# 405\_Convert.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer, write an algorithm to convert it to hexadecimal. For negative integer, two’s complement method is used.

Note:

All letters in hexadecimal (a-f) must be in lowercase.

The hexadecimal string must not contain extra leading 0s. If the number is zero, it is represented by a single zero character '0';

otherwise, the first character in the hexadecimal string will not be the zero character.

The given number is guaranteed to fit within the range of a 32-bit signed integer.

You must not use any method provided by the library which converts/formats the number to hex directly.

Example 1:

Input:

26

Output:

"1a"

Example 2:

Input:

-1

Output:

"ffffffff"

class Solution {

public String toHex(int num) {

if (num == 0){

return "0";

}

char[] map = {'0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'};

StringBuilder sb = new StringBuilder();

while (num != 0){

sb.append(map[num & 15]);

num = num >>> 4;

}

return sb.reverse().toString();

}

}

15 decimal is 1111 in binary. By anding (&) 1111 and the input number, you basically get the last 4 binary digits of the input number.

e.g. input number is decimal 33. In binary 33 is 0010 0001. So, 0010 0001 & 0000 1111 = 0000 0001

# 406\_MinimumSizeSubarraySum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray

of which the sum ≥ s. If there isn't one, return -1 instead.

Have you met this question in a real interview? Yes

Example

Given the array [2,3,1,2,4,3] and s = 7, the subarray [4,3] has the minimal length under the problem constraint.

Challenge

If you have figured out the O(n) solution, try coding another solution of which the time complexity is O(n log n).

Method 1: two points, sliding window

Differnt from Leetcode 862 which could be negative integer

https://github.com/optimisea/Leetcode/blob/master/Java/862\_Shortest.java

time complexity: O(n)

class Solution {

public int minSubArrayLen(int s, int[] nums) {

int ans = Integer.MAX\_VALUE;

int start = 0;

int end = 0;

int sum = 0;

while (end < nums.length){

sum += nums[end];

end++;

while (sum >= s && start < end){

ans = Math.min(ans, end - start);

sum -= nums[start];

start++;

}

}

if (ans == Integer.MAX\_VALUE){

return 0;

}

return ans;

}

}

# 406\_Queue.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers

(h, k), where h is the height of the person and k is the number of people in front of this person who have

a height greater than or equal to h. Write an algorithm to reconstruct the queue.

Note:

The number of people is less than 1,100.

Example

Input:

[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:

[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

https://leetcode.com/problems/queue-reconstruction-by-height/discuss/89350/Java-solution-using-Arrays.sort()-and-%22insert-sorting%22-idea

Method 1: O(n^2)

class Solution {

public int[][] reconstructQueue(int[][] people) {

Arrays.sort(people, new Comparator<int[]>(){

public int compare(int[] a, int[] b){

if (a[0] == b[0]){

return a[1] - b[1];

}

return b[0] - a[0];

}

});

List<int[]> tmp = new ArrayList<>();

for (int i = 0; i < people.length; i++){

tmp.add(people[i][1], new int[]{people[i][0], people[i][1]});

}

int[][] result = new int[people.length][2];

for (int i = 0; i < people.length; i++){

result[i][0] = tmp.get(i)[0];

result[i][1] = tmp.get(i)[1];

}

// int i = 0;

// for (int[] t : tmp){

// result[i][0] = t[0];

// result[i++][1] = t[1];

// }

//return tmp.toArray(new int[people.length][2]);

return result;

}

}

# 408\_AddBinary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two binary strings, return their sum (also a binary string).

Have you met this question in a real interview? Yes

Example

a = 11

b = 1

Return 100

Method: Treat as 10 bit addition except for 2

public class Solution {

/\*

\* @param a: a number

\* @param b: a number

\* @return: the result

\*/

public String addBinary(String a, String b) {

String ans = "";

int carry = 0;

int sum;

for (int i = a.length()-1, j = b.length()-1; i>=0 || j>=0; i--,j--){

sum = carry;

sum += (i>=0) ? a.charAt(i) - '0' : 0;

sum += (j>=0) ? b.charAt(j) - '0' : 0;

ans = sum % 2 + ans;

carry = sum / 2;

}

if (carry != 0){

ans = carry + ans;

}

return ans;

}

}

class Solution {

public String addBinary(String a, String b) {

StringBuilder sb = new StringBuilder();

int carry = 0;

int sum = 0;

for (int i = a.length() - 1, j = b.length() - 1; i >=0 || j>= 0; i--, j--){

sum = carry;

sum += i >= 0 ? a.charAt(i) - '0' : 0;

sum += j >= 0 ? b.charAt(j) - '0' : 0;

carry = sum / 2;

sb.insert(0, sum %2);

}

if (carry != 0){

sb.insert(0, carry);

}

return sb.toString();

}

}

Best solution:

class Solution {

public String addBinary(String a, String b) {

StringBuilder sb = new StringBuilder();

int i = a.length() - 1;

int j = b.length() - 1;

int carry = 0;

while (i >= 0 || j >= 0){

int numA = i >= 0 ? (int)(a.charAt(i) - '0') : 0;

int numB = j >= 0 ? (int)(b.charAt(j) - '0') : 0;

int digit = numA + numB + carry;

sb.append(digit%2);

carry = digit/2;

i--;

j--;

}

if (carry != 0){

sb.append(carry);

}

return sb.reverse().toString();

}

}

# 408\_Valid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty string s and an abbreviation abbr, return whether the string matches with the given abbreviation.

A string such as "word" contains only the following valid abbreviations:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Notice that only the above abbreviations are valid abbreviations of the string "word". Any other string is not a valid abbreviation of "word".

Note:

Assume s contains only lowercase letters and abbr contains only lowercase letters and digits.

Example 1:

Given s = "internationalization", abbr = "i12iz4n":

Return true.

Example 2:

Given s = "apple", abbr = "a2e":

Return false.

https://github.com/optimisea/Leetcode/blob/master/Java/637\_CheckWordAbbreviation.java

class Solution {

public boolean validWordAbbreviation(String word, String abbr) {

int i = 0;

int j = 0;

while (i < word.length() && j < abbr.length()){

if (word.charAt(i) == abbr.charAt(j)){

i++;

j++;

continue;

}

if (abbr.charAt(j) <= '0' || abbr.charAt(j) > '9'){

return false;

}

int num = 0;

while (j < abbr.length() && Character.isDigit(abbr.charAt(j))){

num = num \* 10 + (int) (abbr.charAt(j) - '0');

j++;

}

i += num;

}

return i == word.length() && j == abbr.length();

}

}

public boolean validWordAbbreviation(String word, String abbr) {

int i = 0, j = 0;

while (i < word.length() && j < abbr.length()) {

if (word.charAt(i) == abbr.charAt(j)) {

++i;++j;

continue;

}

if (abbr.charAt(j) <= '0' || abbr.charAt(j) > '9') {

return false;

}

int start = j;

while (j < abbr.length() && abbr.charAt(j) >= '0' && abbr.charAt(j) <= '9') {

++j;

}

int num = Integer.valueOf(abbr.substring(start, j));

i += num;

}

return i == word.length() && j == abbr.length();

}

Better solution:

public boolean validWordAbbreviation(String word, String abbr){

int i = 0;

int j = 0;

while (i < word.length() && j < abbr.length()){

if (Character.isDigit(abbr.charAt(j)){

if (abbr.charAt(j) == '0'){

return false;

}

int num = 0;

while (j < abbr.length() && Character.isDigit(abbr.charAt(j))){

num = 10 \* num + (int)(abbr.charAt(j) - '0');

j++;

}

i += num;

}else{

if (word.charAt(i) != abbr.charAt(j)){

return false;

}

i++;

j++;

}

}

return i == word.length() && j == abbr.length();

}

# 41\_FirstMissingPositive.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an unsorted integer array, find the first missing positive integer.

For example,

Given [1,2,0] return 3,

and [3,4,-1,1] return 2.

Your algorithm should run in O(n) time and uses constant space.

The key here is to use swapping to keep constant space and also make use of the length of the array,

which means there can be at most n positive integers.

So each time we encounter an valid integer, find its correct position and swap. Otherwise we continue.

class Solution {

public int firstMissingPositive(int[] nums) {

int i = 0;

while (i < nums.length){

if (nums[i] <= 0 || nums[i] > nums.length || nums[i] == nums[nums[i] -1]){

i++;

}else if (nums[i] != nums[nums[i] - 1]){

swap(nums, i, nums[i] - 1);

}

}

i = 0;

while (i < nums.length && nums[i] == i + 1){

i++;

}

return i+1;

}

private void swap(int[] nums, int i, int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

Best solution:

The same as https://github.com/optimisea/Leetcode/blob/master/Java/442\_Find.java

class Solution {

public int firstMissingPositive(int[] nums) {

for (int i = 0; i < nums.length; i++){

while (nums[i] > 0 && nums[i] <= nums.length && i != nums[i] - 1 && nums[i] != nums[nums[i] - 1]){

swap(nums, i, nums[i]-1);

}

}

int i = 0;

while (i < nums.length && i == nums[i] - 1){

i++;

}

return i+1;

}

private void swap (int[] A, int i, int j){

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

# 410\_Split.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array which consists of non-negative integers and an integer m, you can split the array into

m non-empty continuous subarrays. Write an algorithm to minimize the largest sum among these m subarrays.

Note:

If n is the length of array, assume the following constraints are satisfied:

1 ≤ n ≤ 1000

1 ≤ m ≤ min(50, n)

Examples:

Input:

nums = [7,2,5,10,8]

m = 2

Output:

18

Explanation:

There are four ways to split nums into two subarrays.

The best way is to split it into [7,2,5] and [10,8],

where the largest sum among the two subarrays is only 18.

https://leetcode.com/problems/split-array-largest-sum/discuss/89817/Clear-Explanation:-8ms-Binary-Search-Java?page=2

The same as Leetcode 1014 Capacity To Ship Packages Within D Days

https://github.com/optimisea/Leetcode/blob/master/Java/1014\_Capacity.java

875 Koko Eating Bananas

https://github.com/optimisea/Leetcode/blob/master/Java/875\_Koko.java

class Solution {

public int splitArray(int[] nums, int m) {

int max = 0;

int sum = 0;

for (int num : nums){

max = Math.max(max, num);

sum += num;

}

int start = max;

int end = sum;

while (start <= end){

int mid = start + (end - start) / 2;

if (lessThanMid(mid, nums, m)){

end = mid - 1;

}else{

start = mid + 1;

}

}

return start;

}

private boolean lessThanMid(int[] nums, int target, int m){

int sum = 0;

int count = 0;

for (int num : nums){

sum += num;

if (sum > target){

count++;

sum = num;

if (count == m){

return false;

}

}

}

return true;

}

}

# 412\_FizzBuzz.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a program that outputs the string representation of numbers from 1 to n.

But for multiples of three it should output “Fizz” instead of the number and for the multiples of five output “Buzz”.

For numbers which are multiples of both three and five output “FizzBuzz”.

Example:

n = 15,

Return:

[

"1",

"2",

"Fizz",

"4",

"Buzz",

"Fizz",

"7",

"8",

"Fizz",

"Buzz",

"11",

"Fizz",

"13",

"14",

"FizzBuzz"

]

class Solution {

public List<String> fizzBuzz(int n) {

List<String> result = new ArrayList<>();

if (n <= 0){

return result;

}

for (int i = 1; i <= n; i++){

if (i % 3 == 0 && i % 5 == 0){

result.add("FizzBuzz");

}else if (i % 5 == 0){

result.add("Buzz");

}else if (i % 3 == 0){

result.add("Fizz");

}else{

result.add(String.valueOf(i));

}

}

return result;

}

}

# 413\_ArithmeticSlices.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A sequence of number is called arithmetic if it consists of at least three elements and if the difference

between any two consecutive elements is the same.

For example, these are arithmetic sequence:

1, 3, 5, 7, 9

7, 7, 7, 7

3, -1, -5, -9

The following sequence is not arithmetic.

1, 1, 2, 5, 7

A zero-indexed array A consisting of N numbers is given. A slice of that array is any pair of integers (P, Q)

such that 0 <= P < Q < N.

A slice (P, Q) of array A is called arithmetic if the sequence:

A[P], A[p + 1], ..., A[Q - 1], A[Q] is arithmetic. In particular, this means that P + 1 < Q.

The function should return the number of arithmetic slices in the array A.

Example:

A = [1, 2, 3, 4]

return: 3, for 3 arithmetic slices in A: [1, 2, 3], [2, 3, 4] and [1, 2, 3, 4] itself.

class Solution {

public int numberOfArithmeticSlices(int[] A) {

if (A == null || A.length < 3){

return 0;

}

int sum = 0;

int curr = 0;

for (int i = 2 ; i < A.length; i++){

if (A[i] - A[i-1] == A[i-1] -A[i-2]){

curr++;

sum += curr;

}else{

curr = 0;

}

}

return sum;

}

}

DP

dp denotes the number of slices ending at i

class Solution {

public int numberOfArithmeticSlices(int[] A) {

int res = 0;

int[] dp = new int[A.length];

for (int i = 1; i < A.length - 1; i++){

if (A[i-1] - A[i] == A[i] - A[i+1]){

dp[i+1] = dp[i] + 1;

}

res += dp[i+1];

}

return res;

}

}

class Solution {

public int numberOfArithmeticSlices(int[] A) {

int res = 0;

int[] dp = new int[A.length];

for (int i = 2; i < A.length; i++){

if (A[i-2] - A[i-1] == A[i-1] - A[i]){

dp[i] = dp[i-1] + 1;

}

res += dp[i];

}

return res;

}

}

# 414\_ThirdMaximumNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty array of integers, return the third maximum number in this array.

If it does not exist, return the maximum number. The time complexity must be in O(n).

Example 1:

Input: [3, 2, 1]

Output: 1

Explanation: The third maximum is 1.

Example 2:

Input: [1, 2]

Output: 2

Explanation: The third maximum does not exist, so the maximum (2) is returned instead.

Example 3:

Input: [2, 2, 3, 1]

Output: 1

Explanation: Note that the third maximum here means the third maximum distinct number.

Both numbers with value 2 are both considered as second maximum.

class Solution {

public int thirdMax(int[] nums) {

long firstMax = Long.MIN\_VALUE;

long secondMax = Long.MIN\_VALUE;

long thirdMax = Long.MIN\_VALUE;

for (int i = 0; i < nums.length; i++){

if (nums[i] == firstMax || nums[i] == secondMax || nums[i] == thirdMax){

continue;

}

if (nums[i] > firstMax){

thirdMax = secondMax;

secondMax = firstMax;

firstMax = nums[i];

}else if (nums[i] > secondMax){

thirdMax = secondMax;

secondMax = nums[i];

}else if (nums[i] > thirdMax){

thirdMax = nums[i];

}

}

if (thirdMax == Long.MIN\_VALUE){

return (int) firstMax;

}

return (int) thirdMax;

}

}

class Solution {

public int thirdMax(int[] nums) {

long first = Long.MIN\_VALUE;

long second = Long.MIN\_VALUE;

long res = Long.MIN\_VALUE;

for (int num : nums){

if (num > first){

res = second;

second = first;

first = num;

}else if (num > second && num < first){

res = second;

second = num;

}else if (num > res && num < second){

res = num;

}

}

if (res == Long.MIN\_VALUE){

return (int)first;

}

return (int)res;

}

}

# 415\_Add.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two non-negative integers num1 and num2 represented as string, return the sum of num1 and num2.

Note:

The length of both num1 and num2 is < 5100.

Both num1 and num2 contains only digits 0-9.

Both num1 and num2 does not contain any leading zero.

You must not use any built-in BigInteger library or convert the inputs to integer directly.

class Solution {

public String addStrings(String num1, String num2) {

StringBuilder sb = new StringBuilder();

int i = num1.length() - 1;

int j = num2.length() - 1;

int sum = 0;

int carry = 0;

int digit = 0;

while (i >= 0 && j >= 0){

sum = carry + (int) (num1.charAt(i) - '0') + (int) (num2.charAt(j) - '0');

carry = sum / 10;

digit = sum % 10;

sb.append(digit);

i--;

j--;

}

if (i >= 0){

while (i >= 0){

sum = carry + (int) (num1.charAt(i) - '0');

carry = sum / 10;

digit = sum % 10;

sb.append(digit);

i--;

}

}else{

while (j >= 0){

sum = carry + (int) (num2.charAt(j) - '0');

carry = sum / 10;

digit = sum % 10;

sb.append(digit);

j--;

}

}

if (carry > 0){

sb.append(carry);

}

return sb.reverse().toString();

}

}

https://github.com/optimisea/Leetcode/blob/master/Java/655\_BigIntegerAddition.java

Better solution:

class Solution {

public String addStrings(String num1, String num2) {

int m = num1.length();

int n = num2.length();

StringBuilder sb = new StringBuilder();

int sum = 0;

int carry = 0;

for (int i = m -1, j = n-1; i>=0 || j >= 0; i--, j--){

sum = carry;

sum += (i >= 0) ? (int) (num1.charAt(i) - '0') : 0;

sum += (j >= 0) ? (int) (num2.charAt(j) - '0') : 0;

sb.append(sum%10);

carry = sum/10;

}

if (carry != 0){

sb.append(carry);

}

return sb.reverse().toString();

}

}

# 416\_Partition.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty array containing only positive integers, find if the array can be partitioned into

two subsets such that the sum of elements in both subsets is equal.

Note:

Each of the array element will not exceed 100.

The array size will not exceed 200.

Example 1:

Input: [1, 5, 11, 5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].

Example 2:

Input: [1, 2, 3, 5]

Output: false

Explanation: The array cannot be partitioned into equal sum subsets.

Similar to backpack problem

• State:

• f[i][S] “前i”个物品,取出一些能否组成和为S

• Function:

• a[i-1] 是第i个物品下标是i-1

• f[i][S] = f[i-1][S - a[i-1]] or f[i-1][S]

• Intialize:

• f[i][0] = true; f[0][1..target] = false

• Answer:

• f[n][target]

• O(n\*S) , 滚动数组优化

Method 1: DP

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for (int i = 0; i < nums.length; i++){

sum += nums[i];

}

if (sum % 2 != 0){

return false;

}

int target = sum / 2;

boolean[][] dp = new boolean[nums.length + 1][target+1];

for (int i = 0; i <= nums.length; i++){

dp[i][0] = true;

}

for (int i = 1; i <= nums.length; i++){

for (int j = 0; j <= target; j++){

if (j >= nums[i-1]){

dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i-1]];

}else{

dp[i][j] = dp[i-1][j];

}

}

}

return dp[nums.length][target];

}

}

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for (int i : nums){

sum += i;

}

if (sum%2 != 0){

return false;

}

int target =sum / 2;

int n = nums.length;

boolean[][] dp = new boolean[n+1][target+1];

dp[0][0] = true;

for (int i = 1; i <= n; i++){

for (int j = 0; j <= target; j++){

if (j >= nums[i-1]){

dp[i][j] = dp[i-1][j-nums[i-1]] || dp[i-1][j];

}else{

dp[i][j] = dp[i-1][j];

}

}

}

return dp[n][target];

}

}

Method 2: 滚动数组优化

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for (int i = 0; i < nums.length; i++){

sum += nums[i];

}

if (sum % 2 != 0){

return false;

}

int target = sum / 2;

boolean[][] dp = new boolean[2][target+1];

for (int i = 0; i <= nums.length; i++){

dp[i%2][0] = true;

}

for (int i = 1; i <= nums.length; i++){

for (int j = 0; j <= target; j++){

if (j >= nums[i-1]){

dp[i%2][j] = dp[(i-1)%2][j] || dp[(i-1)%2][j-nums[i-1]];

}else{

dp[i%2][j] = dp[(i-1)%2][j];

}

}

}

return dp[nums.length%2][target];

}

}

Best solution: Reduce space complexity by scan from end to start

if scan from start to end, need additional dimension because end data relies on previous start data

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for (int i : nums){

sum += i;

}

if (sum%2 != 0){

return false;

}

int target =sum / 2;

int n = nums.length;

boolean[] dp = new boolean[target+1];

dp[0] = true;

for (int i = 0; i < nums.length; i++){

for (int j = target; j >= 0; j--){

if (j >= nums[i]){

dp[j] = dp[j] || dp[j - nums[i]];

}

}

}

return dp[target];

}

}

Method 2: backtracking

generalization refer to Leetcode 698

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for (int num : nums){

sum += num;

}

if (sum % 2 != 0){

return false;

}

int target = sum / 2;

int[] sums = new int[2];

Arrays.sort(nums);

return backtrack(nums, target, sums, nums.length - 1);

}

private boolean backtrack(int[] nums, int target, int[] sums,int pos){

if (pos < 0){

for (int i = 0; i < 2; i++){

if (sums[i] != target){

return false;

}

}

return true;

}

for (int i = 0; i < 2; i++){

if (sums[i] + nums[pos] > target){

continue;

}

sums[i] += nums[pos];

if (backtrack(nums, target, sums, pos - 1)){

return true;

}

sums[i] -= nums[pos];

}

return false;

}

}

# 417\_PacificAtlanticOcean.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an m x n matrix of non-negative integers representing the height of each unit cell in a continent, the "Pacific ocean" touches

the left and top edges of the matrix and the "Atlantic ocean" touches the right and bottom edges.

Water can only flow in four directions (up, down, left, or right) from a cell to another one with height equal or lower.

Find the list of grid coordinates where water can flow to both the Pacific and Atlantic ocean.

Note:

The order of returned grid coordinates does not matter.

Both m and n are less than 150.

Example:

Given the following 5x5 matrix:

Pacific ~ ~ ~ ~ ~

~ 1 2 2 3 (5) \*

~ 3 2 3 (4) (4) \*

~ 2 4 (5) 3 1 \*

~ (6) (7) 1 4 5 \*

~ (5) 1 1 2 4 \*

\* \* \* \* \* Atlantic

Return:

[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (positions with parentheses in above matrix).

Method 1: Forward Travel

Time complexity: O(N^2 \* N^2)

class Solution {

public List<int[]> pacificAtlantic(int[][] matrix) {

List<int[]> result = new ArrayList<>();

if (matrix == null || matrix.length == 0){

return result;

}

int m = matrix.length;

int n = matrix[0].length;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (bfs(matrix, i, j)){

result.add(new int[]{i, j});

}

}

}

return result;

}

private boolean bfs(int[][] matrix, int i, int j){

int m = matrix.length;

int n = matrix[0].length;

boolean[][] visited = new boolean[m][n];

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy= new LinkedList<>();

qx.offer(i);

qy.offer(j);

boolean pacific = false;

boolean atlantic = false;

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

for (int k = 0; k < dx.length; k++){

int x = cx + dx[k];

int y = cy + dy[k];

if (x == m || y == n){

atlantic = true;

}else if (x < 0 || y < 0){

pacific = true;

}else if (!visited[x][y] && matrix[cx][cy] >= matrix[x][y]){

visited[x][y] = true;

qx.offer(x);

qy.offer(y);

}

}

}

if (pacific && atlantic){

return true;

}

return false;

}

}

class Solution {

public List<int[]> pacificAtlantic(int[][] matrix) {

List<int[]> res = new ArrayList<>();

if (matrix == null || matrix.length == 0){

return res;

}

int m = matrix.length;

int n = matrix[0].length;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (bfs(matrix, i, j)){

res.add(new int[]{i, j});

}

}

}

return res;

}

private boolean bfs(int[][] matrix, int x, int y){

int m = matrix.length;

int n = matrix[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

boolean[][] visited = new boolean[m][n];

boolean atlantic = false;

boolean pacific = false;

queue.offer(new int[]{x, y});

visited[x][y] = true;

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx < 0 || ny < 0){

pacific = true;

}else if (nx == m || ny == n){

atlantic = true;

}else if (!visited[nx][ny] && matrix[nx][ny] <= matrix[curr[0]][curr[1]]){

visited[nx][ny] = true;

queue.offer(new int[]{nx, ny});

}

if (pacific && atlantic){

return true;

}

}

}

return false;

}

}

Method 2: Best solution: Backward travel

Time complexity: O(N^2 + N^2)

class Solution {

public List<int[]> pacificAtlantic(int[][] matrix) {

List<int[]> res = new ArrayList<>();

if (matrix == null || matrix.length == 0){

return res;

}

int m = matrix.length;

int n = matrix[0].length;

boolean[][] pacific = new boolean[m][n];

boolean[][] atlantic = new boolean[m][n];

Queue<int[]> paQueue = new LinkedList<>();

Queue<int[]> atQueue = new LinkedList<>();

for (int i = 0; i < m; i++){

pacific[i][0] = true;

paQueue.offer(new int[]{i, 0});

atlantic[i][n-1] = true;

atQueue.offer(new int[]{i, n-1});

}

for (int j = 0; j < n; j++){

pacific[0][j] = true;

paQueue.offer(new int[]{0, j});

atlantic[m-1][j] = true;

atQueue.offer(new int[]{m-1, j});

}

bfs(matrix, paQueue, pacific);

bfs(matrix, atQueue, atlantic);

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (pacific[i][j] && atlantic[i][j]){

res.add(new int[]{i, j});

}

}

}

return res;

}

private void bfs(int[][] matrix, Queue<int[]> queue, boolean[][] visited){

int m = matrix.length;

int n = matrix[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >=0 && ny < n && !visited[nx][ny] && matrix[nx][ny] >= matrix[curr[0]][curr[1]]){

//note that here is >= , not < because we are doing backward travel not like method 1 which is forward travel

queue.offer(new int[]{nx, ny});

visited[nx][ny] = true;

}

}

}

}

}

# 418\_IntegertoRoman.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer, convert it to a roman numeral.

The number is guaranteed to be within the range from 1 to 3999.

Have you met this question in a real interview? Yes

Clarification

What is Roman Numeral?

https://en.wikipedia.org/wiki/Roman\_numerals

https://zh.wikipedia.org/wiki/%E7%BD%97%E9%A9%AC%E6%95%B0%E5%AD%97

http://baike.baidu.com/view/42061.htm

Example

4 -> IV

12 -> XII

21 -> XXI

99 -> XCIX

more examples at: http://literacy.kent.edu/Minigrants/Cinci/romanchart.htm

• 如何数位分离? %10 /10

• (扩展)如何将一个数转成k进制? % k / k

Method 1: best solution

public class Solution {

/\*

\* @param n: The integer

\* @return: Roman representation

\*/

public String intToRoman(int n) {

if (n <= 0){

return "";

}

int[] nums = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};

String[] symbols = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};

StringBuilder sb = new StringBuilder();

int digit = 0;

while (n > 0){

int times = n / nums[digit];

for (int i = 0; i < times; i++){

sb.append(symbols[digit]);

}

n = n % nums[digit];

digit++;

}

return sb.toString();

}

}

Method 2:

public class Solution {

/\*\*

\* @param n The integer

\* @return Roman representation

\*/

public String intToRoman(int n) {

// Write your code here

String M[] = {"", "M", "MM", "MMM"};

String C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};

String X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};

String I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};

return M[n / 1000] + C[(n / 100) % 10] + X[(n / 10) % 10] + I[n % 10];

}

}

# 418\_Sentence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a rows x cols screen and a sentence represented by a list of non-empty words, find how many times the given sentence can be fitted on the screen.

Note:

A word cannot be split into two lines.

The order of words in the sentence must remain unchanged.

Two consecutive words in a line must be separated by a single space.

Total words in the sentence won't exceed 100.

Length of each word is greater than 0 and won't exceed 10.

1 ≤ rows, cols ≤ 20,000.

Example 1:

Input:

rows = 2, cols = 8, sentence = ["hello", "world"]

Output:

1

Explanation:

hello---

world---

The character '-' signifies an empty space on the screen.

Example 2:

Input:

rows = 3, cols = 6, sentence = ["a", "bcd", "e"]

Output:

2

Explanation:

a-bcd-

e-a---

bcd-e-

The character '-' signifies an empty space on the screen.

Example 3:

Input:

rows = 4, cols = 5, sentence = ["I", "had", "apple", "pie"]

Output:

1

Explanation:

I-had

apple

pie-I

had--

The character '-' signifies an empty space on the screen.

Method 1:

class Solution {

public int wordsTyping(String[] sentence, int rows, int cols) {

String str = String.join(" ", sentence) + " ";

int start = 0;

int len = str.length();

for (int i = 0; i < rows; i++){

start += cols;

if (str.charAt(start % len) == ' '){

start++;

}else{

while (start > 0 && str.charAt((start - 1) % len) != ' '){

start--;

}

}

}

return start / len;

}

}

Better solution:

Method 2:

class Solution {

public int wordsTyping(String[] sentence, int rows, int cols) {

String str = String.join(" ", sentence) + " ";

int start = 0; //used to track the length that has traveled

int len = str.length();

for (int i = 0; i < rows; i++){

start += cols;

while (start >= 0 && str.charAt(start % len) != ' '){

start--;

}

start++;

}

return start / len;

}

}

# 419\_RomantoInteger.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a roman numeral, convert it to an integer.

The answer is guaranteed to be within the range from 1 to 3999.

Have you met this question in a real interview? Yes

Clarification

What is Roman Numeral?

https://en.wikipedia.org/wiki/Roman\_numerals

https://zh.wikipedia.org/wiki/%E7%BD%97%E9%A9%AC%E6%95%B0%E5%AD%97

http://baike.baidu.com/view/42061.htm

思路:

• 从左往右加起来,比如XVII=10+5+1+1=17

• 那么像IV=4 IX=9 XL=40 XC=90 这样的怎么处理呢?

• 没有4 9 40 90 这种的,字母代表的数字从左往右是从大到小的

• 发现左边的如果小于右边的,就把左边的减去,比如CDXXI

public class Solution {

/\*

\* @param s: Roman representation

\* @return: an integer

\*/

public int romanToInt(String s) {

if (s == null || s.length() == 0){

return 0;

}

int ans = 0;

for (int i = 0; i < s.length() - 1; i++){

int temp = toInt(s.charAt(i));

if (temp < toInt(s.charAt(i+1))){

ans -= temp;

}else{

ans += temp;

}

}

return ans + toInt(s.charAt(s.length()-1));

}

private int toInt(char c){

switch(c){

case 'I':

return 1;

case 'V':

return 5;

case 'X':

return 10;

case 'L':

return 50;

case 'C':

return 100;

case 'D':

return 500;

case 'M':

return 1000;

}

return 0;

}

}

Better version

class Solution {

public int romanToInt(String s) {

Map<Character, Integer> map = new HashMap<>();

map.put('I', 1);

map.put('V', 5);

map.put('X', 10);

map.put('L', 50);

map.put('C', 100);

map.put('D', 500);

map.put('M', 1000);

int res = 0;

for (int i = 0; i < s.length() - 1; i++){

int curr = map.get(s.charAt(i));

int next = map.get(s.charAt(i+1));

if (next > curr){

res -= curr;

}else{

res += curr;

}

}

return res + map.get(s.charAt(s.length() - 1));

}

}

# 421\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty array of numbers, a0, a1, a2, … , an-1, where 0 ≤ ai < 231.

Find the maximum result of ai XOR aj, where 0 ≤ i, j < n.

Could you do this in O(n) runtime?

Example:

Input: [3, 10, 5, 25, 2, 8]

Output: 28

Explanation: The maximum result is 5 ^ 25 = 28.

Method 1: Trie

https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/discuss/130427/()-Beats-92

https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/discuss/91059/Java-O(n)-solution-using-Trie

In order to get O(n), we have to find way to ensure "search" is O(1) which is what Trie does.

Time complexity: O(n)

Space complexity: O(n)

class Solution {

class TrieNode {

private TrieNode[] children;

public TrieNode (){

children = new TrieNode[2]; //TrieNode[0] contains 0; TrieNode[1] contains 1

}

}

public int findMaximumXOR(int[] nums) {

//build Trie

TrieNode root = new TrieNode();

for (int i : nums){

TrieNode node = root;

for (int j = 31; j >= 0; j--){

int bit = (i >> j) & 1;

if (node.children[bit] == null){

node.children[bit] = new TrieNode();

}

node = node.children[bit];

}

}

//find the max of each number and get the global max

int max = Integer.MIN\_VALUE;

for (int i : nums){

TrieNode node = root;

int localMax = 0;

for (int j = 31; j >= 0; j--){

int bit = (i >> j) & 1;

if (node.children[bit ^ 1] != null){

localMax |= (1 << j);

node = node.children[bit^1];

}else{

node = node.children[bit];

}

}

max = Math.max(max, localMax);

}

return max;

}

}

Method 2:

https://leetcode.com/problems/maximum-xor-of-two-numbers-in-an-array/discuss/91049/Java-O(n)-solution-using-bit-manipulation-and-HashMap

# 422\_Valid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sequence of words, check whether it forms a valid word square.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where 0 ≤ k < max(numRows, numColumns).

Note:

The number of words given is at least 1 and does not exceed 500.

Word length will be at least 1 and does not exceed 500.

Each word contains only lowercase English alphabet a-z.

Input:

[

"abcd",

"bnrt",

"crm",

"dt"

]

Output:

true

Explanation:

The first row and first column both read "abcd".

The second row and second column both read "bnrt".

The third row and third column both read "crm".

The fourth row and fourth column both read "dt".

Therefore, it is a valid word square.

class Solution {

public boolean validWordSquare(List<String> words) {

if (words == null){

return true;

}

for (int i = 0; i < words.size(); i++){

for (int j = 0; j < words.get(i).length(); j++){

if (j >= words.size() || i >= words.get(j).length() || words.get(i).charAt(j) != words.get(j).charAt(i)){

return false;

}

}

}

return true;

}

}

# 424\_ Longest Repeating Character Replacement.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string that consists of only uppercase English letters, you can replace any letter in the string

with another letter at most k times. Find the length of a longest substring containing all repeating letters

you can get after performing the above operations.

Note:

Both the string's length and k will not exceed 104.

Example 1:

Input:

s = "ABAB", k = 2

Output:

4

Explanation:

Replace the two 'A's with two 'B's or vice versa.

Example 2:

Input:

s = "AABABBA", k = 1

Output:

4

Explanation:

Replace the one 'A' in the middle with 'B' and form "AABBBBA".

The substring "BBBB" has the longest repeating letters, which is 4.

class Solution {

public int characterReplacement(String s, int k) {

int start = 0;

int end = 0;

int ans = 0;

int maxCount = 0;

int[] hash = new int[26];

while (end < s.length()){

hash[s.charAt(end) - 'A']++;

maxCount = Math.max(maxCount, hash[s.charAt(end) - 'A']);

while (end - start + 1 - maxCount > k){

hash[s.charAt(start) - 'A']--;

maxCount = Math.max(maxCount, hash[s.charAt(start) - 'A']);

start++;

}

ans = Math.max(ans, end - start + 1);

end++;

}

return ans;

}

}

Better version:

class Solution {

public int characterReplacement(String s, int k) {

int res = 0;

int start = 0;

int end = 0;

int maxCount = 0; // tracking the max number value of one character to replace all others

Map<Character, Integer> map = new HashMap<>();

while (end < s.length()){

char cEnd = s.charAt(end);

map.put(cEnd, map.getOrDefault(cEnd, 0) + 1);

maxCount = Math.max(maxCount, map.get(cEnd));

end++;

while (end - start - maxCount > k){

char cStart = s.charAt(start);

map.put(cStart, map.get(cStart) - 1);

maxCount = Math.max(maxCount, map.get(cStart));

start++;

}

res = Math.max(res, end - start);

}

return res;

}

}

# 425\_LetterCombinationsofPhoneNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a digit string excluded 01, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.

Cellphone

Notice

Although the above answer is in lexicographical order, your answer could be in any order you want.

Have you met this question in a real interview? Yes

Example

Given "23"

Return ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

思路:

• 基础枚举型DFS

– 输入数字串长度为n,做n个阶段的选择,DFS n层 – 每一阶段枚举该位的数字对应的一个字母

– 直到所有情况都枚举完

• 怎样写DFS比较好想,比较快?

￼– 先写扩展,再写退出情况

最简单、不易出错:全部放在DFS函数的参数中

中间不改变的东西:可以放在全局变量中

public class Solution {

/\*

\* @param digits: A digital string

\* @return: all posible letter combinations

\*/

public List<String> letterCombinations(String digits) {

List<String> result = new ArrayList<>();

if (digits == null || digits.length() == 0){

return result;

}

String[] phone = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

dfs(result, 0, "", digits, phone);

return result;

}

private void dfs(List<String> result, int x, String str, String digits, String[] phone){

if (x == digits.length()){

result.add(str);

return;

}

int num = digits.charAt(x) - '0';

String s = phone[num];

for (int i = 0; i < s.length() ; i++){

dfs(result, x + 1, str + s.charAt(i), digits, phone);

// no need to remove the last char because string uses deep copy by default

}

}

}

backtracking using StringBuilder

class Solution {

public List<String> letterCombinations(String digits) {

List<String> res = new ArrayList<>();

if (digits == null || digits.length() == 0){

return res;

}

String[] letters = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

backtrack(res, digits, new StringBuilder(), 0, letters);

return res;

}

private void backtrack(List<String> res, String digits, StringBuilder sb, int pos, String[] letters){

if (pos == digits.length()){

res.add(sb.toString());

return;

}

int num = (int)(digits.charAt(pos) - '0');

String letter = letters[num];

for (int i = 0; i < letter.length(); i++){

sb.append(letter.charAt(i));

backtrack(res, digits, sb, pos + 1, letters);

sb.deleteCharAt(sb.length() - 1);

}

}

}

# 427\_Construct.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We want to use quad trees to store an N x N boolean grid. Each cell in the grid can only be true or false. The root node represents the whole grid. For each node, it will be subdivided into four children nodes until the values in the region it represents are all the same.

Each node has another two boolean attributes : isLeaf and val. isLeaf is true if and only if the node is a leaf node. The val attribute for a leaf node contains the value of the region it represents.

Your task is to use a quad tree to represent a given grid. The following example may help you understand the problem better:

Given the 8 x 8 grid below, we want to construct the corresponding quad tree:

It can be divided according to the definition above:

The corresponding quad tree should be as following, where each node is represented as a (isLeaf, val) pair.

For the non-leaf nodes, val can be arbitrary, so it is represented as \*.

Note:

N is less than 1000 and guaranteened to be a power of 2.

If you want to know more about the quad tree, you can refer to its wiki.

/\*

// Definition for a QuadTree node.

class Node {

public boolean val;

public boolean isLeaf;

public Node topLeft;

public Node topRight;

public Node bottomLeft;

public Node bottomRight;

public Node() {}

public Node(boolean \_val,boolean \_isLeaf,Node \_topLeft,Node \_topRight,Node \_bottomLeft,Node \_bottomRight) {

val = \_val;

isLeaf = \_isLeaf;

topLeft = \_topLeft;

topRight = \_topRight;

bottomLeft = \_bottomLeft;

bottomRight = \_bottomRight;

}

};

\*/

class Solution {

public Node construct(int[][] grid) {

int n = grid.length;

return build(grid, 0, 0, n-1, n-1);

}

private Node build(int[][] grid, int r1, int c1, int r2, int c2){

if (c1 > c2 || r1 > r2){

return null;

}

boolean isLeaf = true;

int val = grid[r1][c1];

for (int i = r1; i <= r2; i++){

for (int j = c1; j <= c2; j++){

if (grid[i][j] != val){

isLeaf = false;

}

}

}

if (isLeaf){

return new Node(val==1, isLeaf, null, null, null, null);

}

int rowMid = (r1 + r2)/2;

int colMid = (c1 + c2)/2;

Node topLeft = build(grid, r1, c1, rowMid, colMid);

Node topRight = build(grid, r1, colMid + 1, rowMid, c2);

Node bottomLeft = build(grid, rowMid + 1, c1, r2, colMid);

Node bootmRight = build(grid, rowMid + 1, colMid + 1, r2, c2);

return new Node(false, false, topLeft, topRight, bottomLeft, bootmRight);

}

}

# 428\_Pow.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

implement pow(x, n).

Notice

You don't need to care about the precision of your answer, it's acceptable if the expected answer and your answer 's

difference is smaller than 1e-3.

Have you met this question in a real interview? Yes

Example

Pow(2.1, 3) = 9.261

Pow(0, 1) = 0

Pow(1, 0) = 1

Challenge: O(logn) time

思路:

• 普通求幂的时间复杂度 O(n)

• 怎样更快的求幂?

￼• x^1=x(1)=x1

• x^2=x(10)=x2

• x^3=x(11)=x2\*x1

• x^4=x(100)=x4

• x^5=x(101)=x4\*x1

• x^6=x(110)=x4\*x2

• x1=x1 •(x1)2=x2

• (x2)2= x^4

• (x4)2= x^8

问:7可以分解成哪 几个数的和?

十进制转二进制 %2 /2 时间复杂度O(logn)

class Solution {

public double myPow(double x, int n) {

if (n < 0){

if (n == Integer.MIN\_VALUE ){

if (Math.abs(x) - 1.0 < 0.001){

return 1.0;

}

return 0.0;

}

x = 1/x;

n = -n;

}

double res = 1.0;

double tmp = x;

while (n > 0){

if (n % 2 == 1){

res \*= tmp;

}

tmp \*= tmp;

n /= 2;

}

return res;

}

}

# 428\_Serialize and Deserialize N-ary Tree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or

memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize an N-ary tree. An N-ary tree is a rooted tree in which each node has no more than

N children. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that

an N-ary tree can be serialized to a string and this string can be deserialized to the original tree structure.

For example, you may serialize the following 3-ary tree

as [1 [3[5 6] 2 4]]. You do not necessarily need to follow this format, so please be creative and come up with different approaches

yourself.

Note:

N is in the range of [1, 1000]

Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> children;

public Node() {}

public Node(int \_val,List<Node> \_children) {

val = \_val;

children = \_children;

}

};

\*/

class Codec {

// Encodes a tree to a single string.

public String serialize(Node root) {

List<String> list = new ArrayList<>();

serializePreOrder(root, list);

return String.join(",", list);

}

private void serializePreOrder(Node root, List<String> list){

if (root == null){

return;

}

list.add(String.valueOf(root.val));

list.add(String.valueOf(root.children.size()));

for (Node node : root.children){

serializePreOrder(node, list);

}

}

// Decodes your encoded data to tree.

public Node deserialize(String data) {

if (data.length() == 0){

return null;

}

String[] strs = data.split(",");

Queue<String> queue = new LinkedList<>(Arrays.asList(strs));

return deserializePreOrder(queue);

}

private Node deserializePreOrder(Queue<String> queue){

Node root = new Node();

root.val = Integer.parseInt(queue.poll());

int size = Integer.parseInt(queue.poll());

root.children = new ArrayList<Node>(size);

for (int i = 0; i < size; i++){

root.children.add(deserializePreOrder(queue));

}

return root;

}

}

// Your Codec object will be instantiated and called as such:

// Codec codec = new Codec();

// codec.deserialize(codec.serialize(root));

Better version:

class Codec {

// Encodes a tree to a single string.

public String serialize(Node root) {

StringBuilder sb = new StringBuilder();

serializePreOrder(root, sb);

return sb.toString();

}

private void serializePreOrder(Node root, StringBuilder sb){

if (root == null){

return;

}

sb.append(root.val + ",");

sb.append(root.children.size(), ",");

for (Node node : root.children){

serializePreOrder(node, sb);

}

}

// Decodes your encoded data to tree.

public Node deserialize(String data) {

String[] strs = data.split(",");

int[] pt = new int[1];

pt[0] = 0;

return deserializePreOrder(strs, pt);

}

private Node deserializePreOrder(String[] strs, int[] pt){

int val = Integer.parseInt(strs[pt[0]]);

pt[0]++;

int size = Integer.parseInt(strs[pt[0]]);

pt[0]++;

Node root = new Node(val, new ArrayList<Node>(size));

for (int i = 0; i < size; i++){

root.children.add(deserializePreOrder(strs, pt));

}

return root;

}

}

# 429\_N-ary Tree Level Order Traversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an n-ary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example, given a 3-ary tree:

/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> children;

public Node() {}

public Node(int \_val,List<Node> \_children) {

val = \_val;

children = \_children;

}

};

\*/

class Solution {

public List<List<Integer>> levelOrder(Node root) {

List<List<Integer>> res = new ArrayList<>();

if (root == null){

return res;

}

Queue<Node> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

int size = queue.size();

List<Integer> list = new ArrayList<>();

for (int i = 0; i < size; i++){

Node node = queue.poll();

list.add(node.val);

for (Node n : node.children){

if (n != null){

queue.offer(n);

}

}

}

res.add(list);

}

return res;

}

}

# 43\_MultiplyStrings.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2.

Note:

The length of both num1 and num2 is < 110.

Both num1 and num2 contains only digits 0-9.

Both num1 and num2 does not contain any leading zero.

You must not use any built-in BigInteger library or convert the inputs to integer directly.

class Solution {

public String multiply(String num1, String num2) {

int l1 = num1.length();

int l2 = num2.length();

int[] ans = new int[l1+l2+1];

for (int i = 0; i < l1; i++){

for (int j = 0; j < l2; j++){

ans[i+j] += (num1.charAt(l1 - 1 - i) - '0') \* (num2.charAt(l2 - 1 - j) - '0');

}

}

for (int i = 0; i < ans.length - 1; i++){

ans[i+1] += ans[i]/ 10;

ans[i] = ans[i] % 10;

}

StringBuilder sb = new StringBuilder();

int i = l1+l2;

while (ans[i] == 0 && i > 0){

i--;

}

while (i >= 0){

sb.append(ans[i]);

i--;

}

return sb.toString();

}

}

Best solution:

The same as Big Integer Multiplication

https://github.com/optimisea/Leetcode/blob/master/Java/656\_BigIntegerMultiplication.java

class Solution {

public String multiply(String num1, String num2) {

int l1 = num1.length();

int l2 = num2.length();

int[] res = new int[l1+l2];

for (int i = 0; i < l1; i++){

for (int j = 0; j < l2; j++){

res[i+j] += (int)(num1.charAt(l1-i-1) - '0') \* (int)(num2.charAt(l2-j-1) - '0');

}

}

for (int i = 0; i < res.length - 2; i++){

res[i+1] += res[i] / 10;

res[i] %= 10;

}

int i = l1+l2-1;

while (i > 0 && res[i] == 0){

i--;

}

StringBuilder sb = new StringBuilder();

while (i >= 0){

sb.append(res[i]);

i--;

}

return sb.toString();

}

}

Better version:

class Solution {

public String multiply(String num1, String num2) {

if (num1.equals("0") || num2.equals("0")){

return "0";

}

StringBuilder sb = new StringBuilder();

int carry = 0;

int i = 0;

int j = 0;

int prev = 0;

int len1 = num1.length();

int len2 = num2.length();

for (i = 0; i < len1; i++){

int c1 = (int)(num1.charAt(len1-1-i) - '0');

int digits = 0;

int num = 0;

carry = 0;

for (j = 0; j < len2; j++){

int c2 = (int)(num2.charAt(len2-1-j) - '0');

if (sb.length() == i+j){

prev = 0;

}else{

prev = (int)(sb.charAt(i+j) - '0');

}

digits = c1 \* c2 + carry + prev;

num = digits%10;

carry = digits/10;

if (sb.length() == i+j){

sb.append(num);

}else{

sb.setCharAt(i+j, (char)(num + '0'));

}

}

if (carry != 0){

sb.append(carry);

}

}

return sb.reverse().toString();

}

}

# 430\_ Flatten a Multilevel Doubly Linked List.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a doubly linked list which in addition to the next and previous pointers, it could have a child pointer, which may or may not point to a separate doubly linked list. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure, as shown in the example below.

Flatten the list so that all the nodes appear in a single-level, doubly linked list. You are given the head of the first level of the list.

Example:

Input:

1---2---3---4---5---6--NULL

|

7---8---9---10--NULL

|

11--12--NULL

Output:

1-2-3-7-8-11-12-9-10-4-5-6-NULL

Method 1:

/\*

// Definition for a Node.

class Node {

public int val;

public Node prev;

public Node next;

public Node child;

public Node() {}

public Node(int \_val,Node \_prev,Node \_next,Node \_child) {

val = \_val;

prev = \_prev;

next = \_next;

child = \_child;

}

};

\*/

class Solution {

public Node flatten(Node head) {

if (head == null){

return head;

}

dfs(head);//return tail;

return head;

}

private Node dfs(Node head){

Node node = head;

Node tail = head;

while (node != null){

while (node.next != null && node.child == null){

node = node.next;

}

if (node.next == null && node.child == null){

tail = node;

break;

}

//node.child != null && (node.next == null || node.next != null)

Node temp = node.next;

node.next = node.child;

node.child.prev = node;

Node childTail = dfs(node.child);//return the last of the child

childTail.next = temp;

node.child = null;

if (temp != null){

temp.prev = childTail;

}else{

return childTail;

}

node = temp;

}

return tail;

}

}

Method 2:

https://leetcode.com/problems/flatten-a-multilevel-doubly-linked-list/discuss/150321/Easy-Understanding-Java-beat-95.7-with-Explanation

/\*

// Definition for a Node.

class Node {

public int val;

public Node prev;

public Node next;

public Node child;

public Node() {}

public Node(int \_val,Node \_prev,Node \_next,Node \_child) {

val = \_val;

prev = \_prev;

next = \_next;

child = \_child;

}

};

\*/

class Solution {

public Node flatten(Node head) {

if (head == null){

return head;

}

Node curr = head;

while (curr != null){

if (curr.child == null){

curr = curr.next;

continue;

}

Node temp = curr.child;

while (temp.next != null){

temp = temp.next;

}

temp.next = curr.next;

curr.next = curr.child;

curr.child.prev = curr;

curr.child = null;

if (temp.next != null){

temp.next.prev = temp;

}

}

return head;

}

}

Best solution:

class Solution {

public Node flatten(Node head) {

if (head == null){

return head;

}

Node curr = head;

while (curr != null){

if (curr.child == null){

curr = curr.next;

continue;

}

Node origin = curr;

Node temp = curr.next;

curr.next = curr.child;

curr.child.prev = curr;

curr.child = null;

while (curr.next != null){

curr = curr.next;

}

curr.next = temp;

if (temp != null){

temp.prev = curr;

}

curr = origin.next;

}

return head;

}

}

# 433\_ Minimum Genetic Mutation.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A gene string can be represented by an 8-character long string, with choices from "A", "C", "G", "T".

Suppose we need to investigate about a mutation (mutation from "start" to "end"), where ONE mutation is defined as ONE single character changed in the gene string.

For example, "AACCGGTT" -> "AACCGGTA" is 1 mutation.

Also, there is a given gene "bank", which records all the valid gene mutations. A gene must be in the bank to make it a valid gene string.

Now, given 3 things - start, end, bank, your task is to determine what is the minimum number of mutations needed to mutate from "start" to "end". If there is no such a mutation, return -1.

Note:

Starting point is assumed to be valid, so it might not be included in the bank.

If multiple mutations are needed, all mutations during in the sequence must be valid.

You may assume start and end string is not the same.

Example 1:

start: "AACCGGTT"

end: "AACCGGTA"

bank: ["AACCGGTA"]

return: 1

Example 2:

start: "AACCGGTT"

end: "AAACGGTA"

bank: ["AACCGGTA", "AACCGCTA", "AAACGGTA"]

return: 2

Example 3:

start: "AAAAACCC"

end: "AACCCCCC"

bank: ["AAAACCCC", "AAACCCCC", "AACCCCCC"]

return: 3

The same as word ladder

class Solution {

public int minMutation(String start, String end, String[] bank) {

if (bank == null || bank.length == 0){

return -1;

}

if (start.equals(end)){

return 0;

}

Set<String> bankSet = new HashSet<>();

for (String str : bank){

bankSet.add(str);

}

if (!bankSet.contains(end)){

return -1;

}

int level = 0;

Queue<String> queue = new LinkedList<>();

Set<String> set = new HashSet<>();

queue.offer(start);

set.add(start);

char[] genes = {'A', 'C', 'G', 'T'};

while(!queue.isEmpty()){

int size = queue.size();

level++;

for (int i = 0; i < size; i++){

String str = queue.poll();

for (int j = 0; j < str.length(); j++){

char[] strChar = str.toCharArray();

for (char ch : genes){

if (str.charAt(j) != ch){

strChar[j] = ch;

String s = new String(strChar);

if (end.equals(s)){

return level;

}

if (!set.contains(s) && bankSet.contains(s)){

queue.offer(s);

set.add(s);

}

}

}

}

}

}

return -1;

}

}

class Solution {

public int minMutation(String start, String end, String[] bank) {

Set<String> set = new HashSet<>();

for (String s : bank){

set.add(s);

}

Queue<String> queue = new LinkedList<>();

Set<String> visited = new HashSet<>();

queue.offer(start);

visited.add(start);

char[] dir = {'A', 'C', 'G', 'T'};

int res = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

String curr = queue.poll();

if (curr.equals(end)){

return res;

}

char[] charArr = curr.toCharArray();

for (int j = 0; j < charArr.length; j++){

char c = charArr[j];

for (int k = 0; k < dir.length; k++){

char ch = dir[k];

if (c != ch){

charArr[j] = ch;

String str = String.valueOf(charArr);

if (set.contains(str) && !visited.contains(str)){

visited.add(str);

queue.offer(str);

}

}

}

charArr[j] = c;

}

}

res++;

}

return -1;

}

}

# 200\_NumberofIslands.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a boolean 2D matrix, 0 is represented as the sea, 1 is represented as the island. If two 1 is adjacent,

we consider them in the same island. We only consider up/down/left/right adjacent.

Find the number of islands.

Have you met this question in a real interview? Yes

Example

Given graph:

[

[1, 1, 0, 0, 0],

[0, 1, 0, 0, 1],

[0, 0, 0, 1, 1],

[0, 0, 0, 0, 0],

[0, 0, 0, 0, 1]

]

Method 1:

Time complexity; O(mn)

public class Solution {

/\*

\* @param grid: a boolean 2D matrix

\* @return: an integer

\*/

public int numIslands(boolean[][] grid) {

if (grid == null || grid.length == 0 || grid[0].length == 0){

return 0;

}

int count = 0;

for (int i = 0; i < grid.length; i++){

for (int j = 0; j < grid[0].length; j++){

if (grid[i][j]){

bfs(grid, i, j);

count++;

}

}

}

return count;

}

private void bfs(boolean[][] grid, int i, int j){

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

int[] dx = {0, 1, 0, -1};

int[] dy = {1, 0, -1, 0};

qx.offer(i);

qy.offer(j);

grid[i][j] = false;

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

for (int k = 0; k < 4; k++){

int nx = cx + dx[k];

int ny = cy + dy[k];

if (isBound(grid, nx, ny) && grid[nx][ny]){

qx.offer(nx);

qy.offer(ny);

grid[nx][ny] = false;

}

}

}

}

private boolean isBound(boolean[][] grid, int x, int y){

return x >= 0 && x < grid.length && y >= 0 && y < grid[0].length;

}

}

Method 2: create a new class

public class Solution {

/\*

\* @param grid: a boolean 2D matrix

\* @return: an integer

\*/

private class Coordinate{

int x;

int y;

public Coordinate(int x, int y){

this.x = x;

this.y = y;

}

}

public int numIslands(boolean[][] grid) {

if (grid == null || grid.length == 0 || grid[0].length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

int isIsland = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j]){

markByBFS(grid, i, j);

isIsland++;

}

}

}

return isIsland;

}

private void markByBFS(boolean[][] grid, int x, int y){

int[] directionX = {0, 1, -1, 0};

int[] directionY = {-1, 0, 0, 1};

Queue<Coordinate> queue = new LinkedList<>();

queue.offer(new Coordinate(x,y));

grid[x][y] = false;

while (!queue.isEmpty()){

Coordinate coor = queue.poll();

for (int i = 0; i < 4; i++){

Coordinate adj = new Coordinate(coor.x + directionX[i], coor.y + directionY[i]);

if (!inBound(adj, grid)){

continue;

}

if (grid[adj.x][adj.y]){

queue.offer(adj);

grid[adj.x][adj.y] = false;

}

}

}

}

private boolean inBound(Coordinate adj, boolean[][] grid){

int m = grid.length;

int n = grid[0].length;

return adj.x >= 0 && adj.y >= 0 && adj.x < m && adj.y < n;

}

}

Change the value of grid:

class Solution {

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

int count = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == '1'){

bfs(grid, i, j);

count++;

}

}

}

return count;

}

private void bfs(char[][] grid, int x, int y){

int m = grid.length;

int n = grid[0].length;

Queue<int[]> queue = new LinkedList<>();

int[][] dirs = {{1,0}, {0, 1}, {-1, 0}, {0, -1}};

// boolean[][] visited = new boolean[m][n];

queue.offer(new int[]{x, y});

grid[x][y] = '0';

// visited[x][y] = true;

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] == '1'){

queue.offer(new int[]{nx, ny});

grid[nx][ny] = '0';

}

}

}

}

}

Best solution without changing value of grid

class Solution {

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

int count = 0;

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == '1' && !visited[i][j]){

bfs(grid, i, j, visited);

count++;

}

}

}

return count;

}

private void bfs(char[][] grid, int x, int y, boolean[][] visited){

int m = grid.length;

int n = grid[0].length;

Queue<int[]> queue = new LinkedList<>();

int[][] dirs = {{1,0}, {0, 1}, {-1, 0}, {0, -1}};

queue.offer(new int[]{x, y});

visited[x][y] = true;

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] == '1' && !visited[nx][ny]){

queue.offer(new int[]{nx, ny});

visited[nx][ny] = true;

}

}

}

}

}

Method 3: UF

class Solution {

class UF {

int[] parent;

public UF (int N){

parent = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

}

}

public int find(int x){

if (x == parent[x]){

return x;

}

return parent[x] = find(parent[x]);//path compression

}

public void union (int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

}

}

}

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

UF uf = new UF(m\*n);

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

int count = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == '1'){

count++;

int num1 = i \* n + j;

for (int[] dir : dirs){

int x = i + dir[0];

int y = j + dir[1];

if (x >= 0 && x < m && y >= 0 && y < n && grid[x][y] == '1'){

int num2 = x \* n + y;

if (uf.find(num1) != uf.find(num2)){

uf.union(num1, num2);

count--;

}

}

}

}

}

}

return count;

}

}

class Solution {

class UF {

int[] parent;

int[] size;

int count;

public UF (int N){

parent = new int[N];

size = new int[N];

count = N;

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (x == parent[x]){

return x;

}

return parent[x] = find(parent[x]);

}

public void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

count--;

}

}

public int getSize(){

return count;

}

}

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

UF uf = new UF(m\*n);

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

int countZero = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == '1'){

for (int[] dir : dirs){

int nx = i + dir[0];

int ny = j + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] == '1'){

int curr = converter(i, j, n);

int next = converter(nx, ny, n);

uf.union(curr, next);

}

}

}else{

countZero++;

}

}

}

return uf.getSize() - countZero;

}

private int converter(int x, int y, int n){

return x \* n + y;

}

}

# 434\_NumberofIslandsII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a n,m which means the row and column of the 2D matrix and an array of pair A( size k).

Originally, the 2D matrix is all 0 which means there is only sea in the matrix. The list pair has k operator

and each operator has two integer A[i].x, A[i].y means that you can change the grid matrix[A[i].x][A[i].y]

from sea to island. Return how many island are there in the matrix after each operator.

Notice

0 is represented as the sea, 1 is represented as the island. If two 1 is adjacent, we consider them in the same island.

We only consider up/down/left/right adjacent.

Have you met this question in a real interview?

Example

Given n = 3, m = 3, array of pair A = [(0,0),(0,1),(2,2),(2,1)].

return [1,1,2,2].

/\*\*

\* Definition for a point.

\* class Point {

\* int x;

\* int y;

\* Point() { x = 0; y = 0; }

\* Point(int a, int b) { x = a; y = b; }

\* }

\*/

Time complexity of Union Find with path compression: O(k + m\*n) m\*n -- union find inialization

Time complexity of BFS: O(k \* m \* n)

public class Solution {

/\*

\* @param n: An integer

\* @param m: An integer

\* @param operators: an array of point

\* @return: an integer array

\*/

class UF{

int[] parent;

int[] size;

int count;

public UF(int N){

parent = new int[N];

size = new int[N];

count = N;

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int a){

if (parent[a] == a){

return a;

}

return parent[a] = find(parent[a]);

}

public void union(int a, int b){

int rootA = find(a);

int rootB = find(b);

if (rootA != rootB){

parent[rootA] = rootB;

size[rootB] += size[rootA];

count--;

}

}

public int size(){

return count;

}

}

public List<Integer> numIslands2(int n, int m, Point[] operators) {

List<Integer> result = new ArrayList<>();

if (operators == null || operators.length == 0){

return result;

}

UF uf = new UF(n\*m);

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

int[][] island = new int[n][m];

int count = 0;

for (int i = 0; i < operators.length; i++){

int x = operators[i].x;

int y = operators[i].y;

if (island[x][y] != 1){

count++;

island[x][y] = 1;

for (int j = 0; j < dx.length; j++){

int cx = x + dx[j];

int cy = y + dy[j];

if (cx >= 0 && cx < n && cy >= 0 && cy < m && island[cx][cy] == 1){

int a = convert2dto1d(x, y, m);

int b = convert2dto1d(cx, cy, m);

if (uf.find(a) != uf.find(b)){

uf.union(a, b);

count--;

}

}

}

}

result.add(count);

}

return result;

}

private int convert2dto1d(int x, int y, int m){

return x \* m + y;

}

}

# 434\_ Number of Segments in a String.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Count the number of segments in a string, where a segment is defined to be a contiguous sequence of non-space characters.

Please note that the string does not contain any non-printable characters.

Example:

Input: "Hello, my name is John"

Output: 5

https://leetcode.com/articles/number-of-segments-in-a-string/

Method 1:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int countSegments(String s) {

if (s == null){

return 0;

}

String trim = s.trim();

if (trim.equals("")){

return 0;

}

return trim.split("\\s+").length;

}

}

Method 2: Good solution

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public int countSegments(String s) {

int count = 0;

for (int i = 0; i < s.length(); i++){

if ((i == 0 || s.charAt(i-1) == ' ') && s.charAt(i) != ' '){

count++;

}

}

return count;

}

}

# 437\_CopyBooks.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n books and the ith book has A[i] pages. You are given k people to copy the n books.

n books list in a row and each person can claim a continous range of the n books.

For example one copier can copy the books from ith to jth continously, but he can not copy the 1st book,

2nd book and 4th book (without 3rd book).

They start copying books at the same time and they all cost 1 minute to copy 1 page of a book.

What's the best strategy to assign books so that the slowest copier can finish at earliest time?

Have you met this question in a real interview?

Example

Given array A = [3,2,4], k = 2.

Return 5( First person spends 5 minutes to copy book 1 and book 2 and second person spends 4 minutes to copy book 3. )

Method: Binary Search

O(n log m) where n is the number of books and m is the sum of the pages

public class Solution {

/\*\*

\* @param pages: an array of integers

\* @param k: An integer

\* @return: an integer

\*/

public int copyBooks(int[] pages, int k) {

if (pages == null || pages.length == 0){

return 0;

}

int total = 0;

int max = 0;

for (int i = 0; i < pages.length; i++){

total += pages[i];

if (max < pages[i]) {

max = pages[i];

}

}

int start = max;

int end = total;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (countCopies(pages, mid) > k){

start = mid;

}else{

end = mid;

}

}

if (countCopies(pages, start) <= k){

return start;

}

return end;

}

private int countCopies(int[] pages, int time){

int copies = 1;

int sum = 0;

for (int i = 0; i < pages.length; i++){

if (sum + pages[i] > time){

copies++;

sum = 0;

}

sum += pages[i];

}

return copies;

}

}

# 437\_Path.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a binary tree in which each node contains an integer value.

Find the number of paths that sum to a given value.

The path does not need to start or end at the root or a leaf, but it must go downwards

(traveling only from parent nodes to child nodes).

The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000.

Example:

root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

10

/ \

5 -3

/ \ \

3 2 11

/ \ \

3 -2 1

Return 3. The paths that sum to 8 are:

1. 5 -> 3

2. 5 -> 2 -> 1

3. -3 -> 11

Method 1:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int pathSum(TreeNode root, int sum) {

if (root == null){

return 0;

}

int self = pathSumIncludeRoot(root, sum); // must include root in the path

int left = pathSum(root.left, sum); //must not include root in the path, may include root.left or not

int right = pathSum(root.right, sum); //must not include root in the path, may include root.left or not

return self + left + right;

}

private int pathSumIncludeRoot(TreeNode root, int sum){

if (root == null){

return 0;

}

int count = 0;

if (sum == root.val){

count++;

}

return count + pathSumIncludeRoot(root.left, sum - root.val) + pathSumIncludeRoot(root.right, sum - root.val);

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int pathSum(TreeNode root, int sum) {

if (root == null){

return 0;

}

return pathSumIncludeRoot(root, sum) + pathSum(root.left, sum) + pathSum(root.right, sum);

}

private int pathSumIncludeRoot(TreeNode root, int sum){

if (root == null){

return 0;

}

int count = 0;

if (sum == root.val){

count++;

}

return count + pathSumIncludeRoot(root.left, sum - root.val) + pathSumIncludeRoot(root.right, sum - root.val);

}

}

# 439\_Ternary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string representing arbitrarily nested ternary expressions, calculate the result of the expression. You can always assume

that the given expression is valid and only consists of digits 0-9, ?, :, T and F (T and F represent True and False respectively).

Note:

The length of the given string is ≤ 10000.

Each number will contain only one digit.

The conditional expressions group right-to-left (as usual in most languages).

The condition will always be either T or F. That is, the condition will never be a digit.

The result of the expression will always evaluate to either a digit 0-9, T or F.

Example 1:

Input: "T?2:3"

Output: "2"

Explanation: If true, then result is 2; otherwise result is 3.

Example 2:

Input: "F?1:T?4:5"

Output: "4"

Explanation: The conditional expressions group right-to-left. Using parenthesis, it is read/evaluated as:

"(F ? 1 : (T ? 4 : 5))" "(F ? 1 : (T ? 4 : 5))"

-> "(F ? 1 : 4)" or -> "(T ? 4 : 5)"

-> "4" -> "4"

Example 3:

Input: "T?T?F:5:3"

Output: "F"

Explanation: The conditional expressions group right-to-left. Using parenthesis, it is read/evaluated as:

"(T ? (T ? F : 5) : 3)" "(T ? (T ? F : 5) : 3)"

-> "(T ? F : 3)" or -> "(T ? F : 5)"

-> "F" -> "F"

class Solution {

public String parseTernary(String expression) {

Stack<Character> stack = new Stack<>();

for (int i = expression.length() - 1; i >= 0; i--){

char c = expression.charAt(i);

if (stack.isEmpty() || stack.peek() != '?'){

stack.push(c);

}else{

stack.pop();

char first = stack.pop();

stack.pop();

char second = stack.pop();

if (c == 'T'){

stack.push(first);

}else{

stack.push(second);

}

}

}

return String.valueOf(stack.peek());

}

}

public String parseTernary(String expression){

Stack<Character> stack = new Stack<>();

for (int i = expression.length() - 1; i >= 0; i--){

char c = expression.charAt(i);

if (c == ':'){

continue;

}else if (!stack.isEmpty() && stack.peek() == '?'){// key note: let the "?" to be pushed in the stack.

stack.pop();

int first = stack.pop();

int second = stack.pop();

if (c == 'T'){

stack.push(first);

}else{

stack.push(second);

}

}else{

stack.push(c);

}

return String.valueOf(stack.pop());

}

}

# 44\_MinimumSubarray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers, find the subarray with smallest sum.

Return the sum of the subarray.

Notice

The subarray should contain one integer at least.

Have you met this question in a real interview? Yes

Example

For [1, -1, -2, 1], return -3.

Method: greedy

public class Solution {

/\*

\* @param nums: a list of integers

\* @return: A integer indicate the sum of minimum subarray

\*/

public int minSubArray(List<Integer> nums) {

if (nums == null || nums.size() == 0){

return 0;

}

int local = nums.get(0);

int global = nums.get(0);

for (int i = 1; i < nums.size(); i++){

local = Math.min(nums.get(i), nums.get(i) + local);

global = Math.min(global, local);

}

return global;

}

}

# 44\_WildcardMatching.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement wildcard pattern matching with support for '?' and '\*'.

'?' Matches any single character.

'\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

isMatch("aaa","aa") → false

isMatch("aa", "\*") → true

isMatch("aa", "a\*") → true

isMatch("ab", "?\*") → true

isMatch("aab", "c\*a\*b") → false

Method: 2D Dynamic programming

The most confusing part for me is how to deal with ‘\*’. At first I couldn’t figure out why

the condition would be (dp[i-1][j] == true || dp[i][j-1] == true). Hope detailed DP description below helps!

dp[i][j]: true if the first i char in String s matches the first j chars in String p

Base case:

origin: dp[0][0]: they do match, so dp[0][0] = true

first row: dp[0][j]: except for String p starts with \*, otherwise all false

first col: dp[i][0]: can't match when p is empty. All false.

Recursion:

Iterate through every dp[i][j]

dp[i][j] = true:

if (s[ith] == p[jth] || p[jth] == '?') && dp[i-1][j-1] == true

elif p[jth] == '\*' && (dp[i-1][j] == true || dp[i][j-1] == true)

-for dp[i-1][j], means that \* acts like an empty sequence. eg: ab, ab\*

-for dp[i][j-1], means that \* acts like any sequences. eg: abcd, ab\*

Start from 0 to len Output put should be dp[s.len][p.len], referring to the whole s matches the whole p

Be careful about the difference of index i,j in String (0 to len-1) and the index i, j in dp (0 to len)!

class Solution {

public boolean isMatch(String s, String p) {

if (s == null || p == null ){

return false;

}

int m = s.length();

int n = p.length();

boolean[][] dp = new boolean[m+1][n+1];

dp[0][0] = true;

char[] sc = s.toCharArray();

char[] pc = p.toCharArray();

for (int i = 1; i <= m; i++){

dp[i][0] = false;

}

for (int j = 1; j <= n; j++){

if (pc[j-1] == '\*'){

dp[0][j] = dp[0][j-1];

}

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (dp[i-1][j-1] && (sc[i-1] == pc[j-1] || pc[j-1] == '?')){

dp[i][j] = true;

}else if (pc[j-1] == '\*' && (dp[i-1][j] || dp[i][j-1])){

dp[i][j] = true;

}

}

}

return dp[m][n];

}

}

'\*' has three cases:

1) represents as empty: dp[i][j] = dp[i][j-1]

2) represents one char: dp[i][j] = dp[i-1][j-1]

3) represetns multiple chars: dp[i][j] = dp[[i-1][j]

Better version:

class Solution {

public boolean isMatch(String s, String p) {

if (s == null || p == null){

return false;

}

char[] sc = s.toCharArray();

char[] pc = p.toCharArray();

int m = sc.length;

int n = pc.length;

boolean[][] dp = new boolean[m+1][n+1];

dp[0][0] = true;

for (int j = 1; j <= n; j++){

if (pc[j-1] == '\*'){

dp[0][j] = dp[0][j-1];

}

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (sc[i-1] == pc[j-1] || pc[j-1] == '?'){

dp[i][j] = dp[i-1][j-1];

}else if (pc[j-1] == '\*'){

dp[i][j] = dp[i][j-1] || dp[i-1][j-1] || dp[i-1][j];

}

}

}

return dp[m][n];

}

}

# 440\_K-th.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given integers n and k, find the lexicographically k-th smallest integer in the range from 1 to n.

Note: 1 ≤ k ≤ n ≤ 109.

Example:

Input:

n: 13 k: 2

Output:

10

Explanation:

The lexicographical order is [1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9], so the second smallest number is 10.

https://leetcode.com/problems/k-th-smallest-in-lexicographical-order/discuss/92242/ConciseEasy-to-understand-Java-5ms-solution-with-Explaination

class Solution {

public int findKthNumber(int n, int k) {

int curr = 1;

k = k -1;

while (k > 0){

int steps = calStep(n, curr, curr+1);

if (steps <= k){

curr += 1;

k -= steps;

}else{

curr \*= 10;

k--;

}

}

return curr;

}

private int calStep(int n, long n1, long n2){

int steps = 0;

while (n1 <= n){

steps += Math.min(n2 - n1, n - n1 + 1);

n1 \*= 10;

n2 \*= 10;

}

return steps;

}

}

# 441\_Arranging.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have a total of n coins that you want to form in a staircase shape, where every k-th row must have exactly k coins.

Given n, find the total number of full staircase rows that can be formed.

n is a non-negative integer and fits within the range of a 32-bit signed integer.

Example 1:

n = 5

The coins can form the following rows:

¤

¤ ¤

¤ ¤

Because the 3rd row is incomplete, we return 2.

Example 2:

n = 8

The coins can form the following rows:

¤

¤ ¤

¤ ¤ ¤

¤ ¤

Because the 4th row is incomplete, we return 3.

class Solution {

public int arrangeCoins(int n) {

long start = 1;

long end = n;

while (start + 1 < end){

long mid = start + (end - start) / 2;

long candid = mid \* (mid+1) / 2;

if (candid <= n){

start = mid;

}else{

end = mid;

}

}

if (end\*(end+1) / 2 <= n){

return (int) end;

}

return (int) start;

}

}

Better:

class Solution {

public int arrangeCoins(int n) {

long start = 1;

long end = n;

while (start + 1 < end){

long mid = start + (end - start) / 2;

long res = (1 + mid) \* mid / 2;

if (res == n){

return (int)mid;

}else if (res < n){

start = mid;

}else{

end = mid;

}

}

if ((start+1)\*start/2 <= n){

return (int)start;

}

return (int)end;

}

}

# 442\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers, 1 ≤ a[i] ≤ n (n = size of array), some elements appear twice and others appear once.

Find all the elements that appear twice in this array.

Could you do it without extra space and in O(n) runtime?

Example:

Input:

[4,3,2,7,8,2,3,1]

Output:

[2,3]

https://leetcode.com/problems/find-all-duplicates-in-an-array/discuss/92411/Java-O(1)-space-O(n)-time-solution-with-swapping

class Solution {

public List<Integer> findDuplicates(int[] nums) {

List<Integer> result = new ArrayList<>();

for (int i = 0; i < nums.length; i++){

int index = Math.abs(nums[i]) - 1;

if (nums[index] < 0){

result.add(Math.abs(nums[i]));

}

nums[index] = - nums[index];

}

return result;

}

}

Best solution:

The same as Leetcode 41 the first missing positive https://github.com/optimisea/Leetcode/blob/master/Java/41\_FirstMissingPositive.java

The same solution as Leetcode 448 Find All Numbers Disappeared in an Array

https://github.com/optimisea/Leetcode/blob/master/Java/448\_Find.java

class Solution {

public List<Integer> findDuplicates(int[] nums) {

List<Integer> res = new ArrayList<>();

for (int i = 0; i < nums.length; i++){

while (nums[i] > 0 && nums[i] <= nums.length && i != nums[i] - 1 && nums[i] != nums[nums[i]-1]){

swap(nums, i, nums[i]-1);

}

}

for (int i = 0; i < nums.length; i++){

if (nums[i] - 1 != i){

res.add(nums[i]);

}

}

return res;

}

private void swap(int[] nums, int i, int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

# 443\_StringCompression.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of characters, compress it in-place.

The length after compression must always be smaller than or equal to the original array.

Every element of the array should be a character (not int) of length 1.

After you are done modifying the input array in-place, return the new length of the array.

Follow up:

Could you solve it using only O(1) extra space?

Example 1:

Input:

["a","a","b","b","c","c","c"]

Output:

Return 6, and the first 6 characters of the input array should be: ["a","2","b","2","c","3"]

Explanation:

"aa" is replaced by "a2". "bb" is replaced by "b2". "ccc" is replaced by "c3".

Example 2:

Input:

["a"]

Output:

Return 1, and the first 1 characters of the input array should be: ["a"]

Explanation:

Nothing is replaced.

Example 3:

Input:

["a","b","b","b","b","b","b","b","b","b","b","b","b"]

Output:

Return 4, and the first 4 characters of the input array should be: ["a","b","1","2"].

Explanation:

Since the character "a" does not repeat, it is not compressed. "bbbbbbbbbbbb" is replaced by "b12".

Notice each digit has it's own entry in the array.

Note:

All characters have an ASCII value in [35, 126].

1 <= len(chars) <= 1000.

Best solution: in place

class Solution {

public int compress(char[] chars) {

int index = 0;

int i = 0;

while (i < chars.length){

char curr = chars[i];

int count = 0;

while (i < chars.length && chars[i] == curr){

count++;

i++;

}

chars[index++] = curr;

if (count > 1){

String str = String.valueOf(count);

for (char c : str.toCharArray()){

chars[index++] = c;

}

}

}

return index;

}

}

Method 1:

Space complexity: O(1)

class Solution {

public int compress(char[] chars) {

if (chars == null || chars.length == 0){

return 0;

}

int ans = 0;

char prev = chars[0];

int count = 1;

for (int i = 1; i < chars.length; i++){

if (chars[i] == prev){

count++;

}else{

chars[ans++] = prev;

if (count > 1){

int tmp = count;

int n = 1;

while (tmp >= 10){

n \*= 10;

tmp /= 10;

}

while (count > 0){

int digit = count / n;

chars[ans++] = (char)(digit + '0');

count %= n;

n /= 10;

}

}

count = 1;

}

prev = chars[i];

}

if (count > 1){

chars[ans++] = prev;

if (count > 1){

int tmp = count;

int n = 1;

while (tmp >= 10){

n \*= 10;

tmp /= 10;

}

while (n > 0){

int digit = count / n;

chars[ans++] = (char)(digit + '0');

count %= n;

n /= 10;

}

}

return ans;

}

chars[ans++] = prev;

return ans;

}

}

Method 2: better

class Solution {

public int compress(char[] chars) {

if (chars == null || chars.length == 0){

return 0;

}

int ans = 0;

int i = 0;

while (i < chars.length){

char curr = chars[i];

int count = 0;

while (i < chars.length && chars[i] == curr){

count++;

i++;

}

chars[ans++] = curr;

if (count > 1){

int tmp = count;

int n = 1;

while (tmp >= 10){

n \*= 10;

tmp /= 10;

}

while (n > 0){

int digit = count / n;

chars[ans++] = (char)(digit + '0');

count %= n;

n /= 10;

}

}

}

return ans;

}

}

Method 3: Best use Integer.toString(count).toCharArray()

class Solution {

public int compress(char[] chars) {

if (chars == null || chars.length == 0){

return 0;

}

int ans = 0;

int i = 0;

while (i < chars.length){

char curr = chars[i];

int count = 0;

while (i < chars.length && chars[i] == curr){

count++;

i++;

}

chars[ans++] = curr;

if (count > 1){

for (char c : Integer.toString(count).toCharArray()){

chars[ans++] = c;

}

}

}

return ans;

}

}

Method 4:

class Solution {

public int compress(char[] chars) {

boolean newchar = true;

int index = 0;

int i = 0;

while (i < chars.length){

if (newchar){

chars[index] = chars[i];

if (i < chars.length - 1 && chars[i+1] == chars[i]){

newchar = false;

}

index++;

i++;

}else{

int count = 1;

while(i < chars.length && chars[i] == chars[i-1]){

count++;

i++;

}

String str = String.valueOf(count);

for (int j = 0; j < str.length(); j++){

chars[index++] = str.charAt(j);

}

newchar = true;

}

}

return index;

}

}

# 444\_Sequence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Check whether the original sequence org can be uniquely reconstructed from the sequences in seqs. The org sequence is a permutation

of the integers from 1 to n, with 1 ≤ n ≤ 104. Reconstruction means building a shortest common supersequence of the sequences in

seqs (i.e., a shortest sequence so that all sequences in seqs are subsequences of it). Determine whether there is only one sequence

that can be reconstructed from seqs and it is the org sequence.

Example 1:

Input:

org: [1,2,3], seqs: [[1,2],[1,3]]

Output:

false

Explanation:

[1,2,3] is not the only one sequence that can be reconstructed, because [1,3,2] is also a valid sequence that can be reconstructed.

Example 2:

Input:

org: [1,2,3], seqs: [[1,2]]

Output:

false

Explanation:

The reconstructed sequence can only be [1,2].

Example 3:

Input:

org: [1,2,3], seqs: [[1,2],[1,3],[2,3]]

Output:

true

Explanation:

The sequences [1,2], [1,3], and [2,3] can uniquely reconstruct the original sequence [1,2,3].

Example 4:

Input:

org: [4,1,5,2,6,3], seqs: [[5,2,6,3],[4,1,5,2]]

Output:

true

The same as Alien Dictionary, build indegree map and priority map

https://github.com/optimisea/Leetcode/blob/master/Java/269\_AlienDictionary.java

indegree: Map<Integer, Integer> indegree, which stores the count of numbers that have higher level of the key

priority map: Map<Integer, Set<Integer>> map, which stores the numbers that have lower level of the key

class Solution {

public boolean sequenceReconstruction(int[] org, List<List<Integer>> seqs) {

int n = org.length;

Map<Integer, Integer> indegree = new HashMap<>(); //store the number of upper level

Map<Integer, Set<Integer>> map = new HashMap<>();//store the value of lower level

for (List<Integer> seq : seqs){

for (int i = 0; i < seq.size() - 1; i++){

int curr = seq.get(i);

int next = seq.get(i+1);

if (!map.containsKey(curr)){

map.put(curr, new HashSet<>());

}

Set<Integer> set = map.get(curr);

if (!set.contains(next)){

set.add(next);

indegree.put(next, indegree.getOrDefault(next, 0) + 1);

}

if (!indegree.containsKey(curr)){

indegree.put(curr, 0);

}

}

}

Queue<Integer> queue = new LinkedList<>();

for (int key : indegree.keySet()){

if (indegree.get(key) == 0){

queue.offer(key);

}

}

if (queue.size() > 1){

return false;

}

int index = 1;

while (!queue.isEmpty()){

int curr = queue.poll();

Set<Integer> set = map.get(curr);

if (set == null){

continue;

}

boolean hasOne= false;

for (int next : set){

indegree.put(next, indegree.get(next) - 1);

if (indegree.get(next) == 0){

if (!hasOne){

queue.offer(next);

index++;

hasOne = true;

}else{

return false;

}

}

}

}

return index == n;

}

}

# 445\_Add.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given two non-empty linked lists representing two non-negative integers.

The most significant digit comes first and each of their nodes contain a single digit.

Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Follow up:

What if you cannot modify the input lists? In other words, reversing the lists is not allowed.

Example:

Input: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)

Output: 7 -> 8 -> 0 -> 7

Method 1: reverse

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

l1 = reverse(l1);

l2 = reverse(l2);

ListNode dummy = new ListNode(-1);

ListNode head = dummy;

ListNode n1= l1;

ListNode n2 = l2;

int carry = 0;

int sum = 0;

while (n1 != null || n2 != null){

sum = carry;

sum += (n1 != null) ? n1.val : 0;

sum += (n2 != null) ? n2.val : 0;

carry = sum / 10;

ListNode node = new ListNode(sum % 10);

head.next = node;

head = node;

n1 = (n1 != null) ? n1.next : n1;

n2 = (n2 != null) ? n2.next : n2;

}

if (carry != 0){

head.next = new ListNode(carry);

}

return reverse(dummy.next);

}

private ListNode reverse(ListNode head){

ListNode prev = null;

while (head != null){

ListNode temp = head.next;

head.next = prev;

prev = head;

head = temp;

}

return prev;

}

}

Method 2: no revserse

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

Stack<ListNode> s1 = new Stack<>();

Stack<ListNode> s2 = new Stack<>();

while (l1 != null){

s1.push(l1);

l1 = l1.next;

}

while (l2 != null){

s2.push(l2);

l2 = l2.next;

}

int sum = 0;

int carry = 0;

ListNode node = new ListNode(-1);

while (!s1.isEmpty() || !s2.isEmpty()){

sum = carry;

if (!s1.isEmpty()){

sum += s1.pop().val;

}

if (!s2.isEmpty()){

sum += s2.pop().val;

}

carry = sum / 10;

node.val = sum % 10;

ListNode head = new ListNode(carry);

head.next = node;

node = head;

}

return node.val != 0 ? node : node.next;

}

}

Better:

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

Stack<Integer> s1 = new Stack<>();

Stack<Integer> s2 = new Stack<>();

ListNode n1 = l1;

while (n1 != null){

s1.push(n1.val);

n1 = n1.next;

}

ListNode n2 = l2;

while (n2 != null){

s2.push(n2.val);

n2 = n2.next;

}

int carry = 0;

int sum = 0;

ListNode dummy = new ListNode(-1);

ListNode head = dummy;

while (!s1.isEmpty() && !s2.isEmpty()){

sum = carry + s1.pop() + s2.pop();

ListNode node = new ListNode(sum%10);

carry = sum/10;

node.next = head.next;

head.next = node;

}

while (!s1.isEmpty()){

sum = carry + s1.pop();

carry = sum / 10;

ListNode node = new ListNode(sum % 10);

node.next = head.next;

head.next = node;

}

while (!s2.isEmpty()){

sum = carry + s2.pop();

carry = sum / 10;

ListNode node = new ListNode(sum % 10);

node.next = head.next;

head.next = node;

}

if (carry == 0){

return dummy.next;

}

ListNode node = new ListNode(carry);

node.next = head.next;

head.next = node;

return dummy.next;

}

}

# 446\_Arithmetic.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A sequence of numbers is called arithmetic if it consists of at least three elements and if the difference between any

two consecutive elements is the same.

For example, these are arithmetic sequences:

1, 3, 5, 7, 9

7, 7, 7, 7

3, -1, -5, -9

The following sequence is not arithmetic.

1, 1, 2, 5, 7

A zero-indexed array A consisting of N numbers is given. A subsequence slice of that array is any sequence of integers

(P0, P1, ..., Pk) such that 0 ≤ P0 < P1 < ... < Pk < N.

A subsequence slice (P0, P1, ..., Pk) of array A is called arithmetic if the sequence A[P0], A[P1], ..., A[Pk-1], A[Pk] is

arithmetic. In particular, this means that k ≥ 2.

The function should return the number of arithmetic subsequence slices in the array A.

The input contains N integers. Every integer is in the range of -231 and 231-1 and 0 ≤ N ≤ 1000. The output is guaranteed

to be less than 231-1.

Example:

Input: [2, 4, 6, 8, 10]

Output: 7

Explanation:

All arithmetic subsequence slices are:

[2,4,6]

[4,6,8]

[6,8,10]

[2,4,6,8]

[4,6,8,10]

[2,4,6,8,10]

[2,6,10]

2D dp with hashmap array

Similar to Arithmetics Slice DP but add additional dimension

does not need to sort, like LIS

The similar as Leetcode 1027

https://github.com/optimisea/Leetcode/blob/master/Java/1027\_Longest.java

class Solution {

public int numberOfArithmeticSlices(int[] A) {

int n = A.length;

int res = 0;

Map<Integer, Integer>[] maps = new Map[A.length];

for (int i = 0; i < n; i++){

maps[i] = new HashMap<>();

for (int j = 0; j < i; j++){

long diff = (long)A[i] - A[j];

if (diff >= Integer.MAX\_VALUE || diff <= Integer.MIN\_VALUE){

continue;

}

int d = (int)diff;

int c1 = maps[j].getOrDefault(d, 0);//the count at ending with index j and difference of d

int c2 = maps[i].getOrDefault(d, 0);// the count at ending with index i and difference between A[i] and A[j] is d

res += c1;

maps[i].put(d, c2 + c1 + 1);

}

}

return res;

}

}

# 447\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n points in the plane that are all pairwise distinct, a "boomerang" is a tuple of points (i, j, k) such

that the distance between i and j equals the distance between i and k (the order of the tuple matters).

Find the number of boomerangs. You may assume that n will be at most 500 and coordinates of points are all in

the range [-10000, 10000] (inclusive).

Example:

Input:

[[0,0],[1,0],[2,0]]

Output:

2

Explanation:

The two boomerangs are [[1,0],[0,0],[2,0]] and [[1,0],[2,0],[0,0]]

class Solution {

public int numberOfBoomerangs(int[][] points) {

int count = 0;

for (int i = 0; i < points.length; i++){

Map<Integer, Integer> map = new HashMap<>();

for (int j = 0; j < points.length; j++){

if (i == j){

continue;

}

int dx = points[i][0] - points[j][0];

int dy = points[i][1] - points[j][1];

int distance = dx \* dx + dy \* dy;

map.put(distance, map.getOrDefault(distance, 0) + 1);

}

for (int val : map.values()){

count += val \* (val - 1); //permutation

}

}

return count;

}

}

class Solution {

public int numberOfBoomerangs(int[][] points) {

int count = 0;

for (int[] p : points){

Map<Integer, Integer> map = new HashMap<>();

for (int[] q : points){

if (p[0] == q[0] && p[1] == q[1]){

continue;

}

int dist = (p[0] - q[0]) \* (p[0] - q[0]) + (p[1] - q[1]) \* (p[1] - q[1]);

map.put(dist, map.getOrDefault(dist, 0) + 1);

}

for (int val : map.values()){

count += val \* (val - 1);

}

}

return count;

}

}

# 448\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers where 1 ≤ a[i] ≤ n (n = size of array), some elements appear twice and others appear once.

Find all the elements of [1, n] inclusive that do not appear in this array.

Could you do it without extra space and in O(n) runtime? You may assume the returned list does not count as extra space.

Example:

Input:

[4,3,2,7,8,2,3,1]

Output:

[5,6]

The same as Leetcode 442

https://github.com/optimisea/Leetcode/blob/master/Java/442\_Find.java

Method 1: without extra space Best solution

class Solution {

public List<Integer> findDisappearedNumbers(int[] nums) {

List<Integer> result = new ArrayList<>();

for (int i = 0; i < nums.length; i++){

while (i != nums[i] - 1 && nums[i] != nums[nums[i] - 1]){

swap(nums, i, nums[i] - 1);

}

}

for (int i = 0; i < nums.length; i++){

if (i != nums[i] - 1){

result.add(i+1);

}

}

return result;

}

private void swap(int[] nums, int i, int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

class Solution {

public List<Integer> findDisappearedNumbers(int[] nums) {

List<Integer> res = new ArrayList<>();

int i = 0;

while (i < nums.length){

if (nums[i] == i+1){

i++;

}else{

swap(nums, i, nums[i]-1);

if (nums[i] == nums[nums[i] - 1]){

i++;

}

}

}

for (int j = 0; j < nums.length; j++){

if (nums[j] != j + 1){

res.add(j+1);

}

}

return res;

}

private void swap(int[] nums, int i, int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

Method 2: with extra space

class Solution {

public List<Integer> findDisappearedNumbers(int[] nums) {

List<Integer> result = new ArrayList<>();

int[] hash = new int[nums.length];

for (int i = 0; i < nums.length; i++){

hash[nums[i]-1]++;

}

for (int i = 0; i < nums.length; i++){

if (hash[i] == 0){

result.add(i+1);

}

}

return result;

}

}

# 450\_Delete.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

Search for a node to remove.

If the node is found, delete the node.

Note: Time complexity should be O(height of tree).

Example:

root = [5,3,6,2,4,null,7]

key = 3

5

/ \

3 6

/ \ \

2 4 7

Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the following BST.

5

/ \

4 6

/ \

2 7

Another valid answer is [5,2,6,null,4,null,7].

5

/ \

2 6

\ \

4 7

Method:

https://leetcode.com/problems/delete-node-in-a-bst/discuss/93296/Recursive-Easy-to-Understand-Java-Solution

Divde and Conquer

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode deleteNode(TreeNode root, int key) {

if (root == null){

return null;

}

if (key < root.val){

root.left = deleteNode(root.left, key);

}else if (key > root.val){

root.right = deleteNode(root.right, key);

}else{

if (root.left == null){

return root.right;

}else if (root.right == null){

return root.left;

}

TreeNode rightSuccessor = findMin(root.right);

root.val = rightSuccessor.val;

root.right = deleteNode(root.right, rightSuccessor.val);

}

return root;

}

private TreeNode findMin(TreeNode root){

while (root.left != null){

root = root.left;

}

return root;

}

}

Method 2: Use inorderSuccessor

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode deleteNode(TreeNode root, int key) {

if (root == null){

return null;

}

if (key < root.val){

root.left = deleteNode(root.left, key);

}else if (key > root.val){

root.right = deleteNode(root.right, key);

}else{

if (root.left == null){

return root.right;

}else if (root.right == null){

return root.left;

}

TreeNode rightSuccessor = inorderSuccessor(root, root.val); //not root.right

root.val = rightSuccessor.val;

root.right = deleteNode(root.right, rightSuccessor.val);

}

return root;

}

private TreeNode inorderSuccessor(TreeNode root, int val){

if (root == null){

return root;

}

if (val >= root.val){

return inorderSuccessor(root.right, val);

}else{

TreeNode left = inorderSuccessor(root.left, val);

return left != null ? left : root;

}

}

}

Better Version:

class Solution {

public TreeNode deleteNode(TreeNode root, int key) {

TreeNode dummy = new TreeNode(-1);

TreeNode parent = dummy;

dummy.left = root;

delete(parent, root, key);

return dummy.left;

}

private void delete(TreeNode parent, TreeNode root, int key){

if (root == null){

return;

}

if (key == root.val){

if (root.right == null){

if (root == parent.left){

parent.left = root.left;

}else{

parent.right = root.left;

}

}else{

if (root == parent.left){

parent.left = root.right;

}else{

parent.right = root.right;

}

TreeNode node = root.right;

while (node.left != null){

node = node.left;

}

node.left = root.left;

}

root.left = null;

root.right = null;

}else if (key > root.val){

delete(root, root.right, key);

}else{

delete(root, root.left, key);

}

}

}

class Solution {

public TreeNode deleteNode(TreeNode root, int key) {

TreeNode dummy = new TreeNode(-1);

dummy.left = root;

delete(root, dummy, key);

return dummy.left;

}

private void delete(TreeNode root, TreeNode parent, int key){

if (root == null){

return;

}

if (root.val == key){

if (root.left == null){

if (root == parent.left){

parent.left = root.right;

}else{

parent.right = root.right;

}

}else{

if (root == parent.left){

parent.left = root.left;

}else{

parent.right = root.left;

}

TreeNode node = root.left;

while (node.right != null){

node = node.right;

}

node.right = root.right;

}

}else if (root.val > key){

delete(root.left, root, key);

}else{

delete(root.right, root, key);

}

}

}

# 450\_ReverseNodesk-Group.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k

then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example,

Given this linked list: 1->2->3->4->5

For k = 2, you should return: 2->1->4->3->5

For k = 3, you should return: 3->2->1->4->5

best solution

Method 1:

class Solution {

public ListNode reverseKGroup(ListNode head, int k) {

if (head == null){

return head;

}

ListNode node = head;

int len = 0;

while (node != null){

node = node.next;

len++;

}

int n = len/k;

if (n == 0){

return head;

}

ListNode dummy = new ListNode(-1);

dummy.next = head;

ListNode dummyNode = dummy;

for (int i = 0; i < n; i++){

ListNode prev = null;

node = head;

for (int j = 0; j < k; j++){

ListNode temp = node.next;

node.next = prev;

prev = node;

node = temp;

}

dummyNode.next = prev;

head.next = node;

dummyNode = head;

head = node;

}

return dummy.next;

}

}

Method 2:

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

/\*

\* @param head: a ListNode

\* @param k: An integer

\* @return: a ListNode

\*/

public ListNode reverseKGroup(ListNode head, int k) {

ListNode dummy = new ListNode(0);

dummy.next = head;

head = dummy;

while (true){

head = reverseK(head, k);

if (head == null){

break;

}

}

return dummy.next;

}

private ListNode reverseK(ListNode head, int k){

ListNode nk = head;

for (int i = 0; i < k; i++){

if (nk == null){

return null;

}

nk = nk.next;

}

if (nk == null){

return null;

}

ListNode nkplus = nk.next;

ListNode n1 = head.next;

//reverse link

ListNode prev = null;

ListNode cur = n1;

while (cur != nkplus){

ListNode temp = cur.next;

cur.next = prev;

prev = cur;

cur = temp;

}

head.next = nk;

n1.next = nkplus;

return n1;

}

}

# 451\_SortCharactersByFrequency.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, sort it in decreasing order based on the frequency of characters.

Example 1:

Input:

"tree"

Output:

"eert"

Explanation:

'e' appears twice while 'r' and 't' both appear once.

So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

Example 2:

Input:

"cccaaa"

Output:

"cccaaa"

Explanation:

Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.

Note that "cacaca" is incorrect, as the same characters must be together.

Example 3:

Input:

"Aabb"

Output:

"bbAa"

Explanation:

"bbaA" is also a valid answer, but "Aabb" is incorrect.

Note that 'A' and 'a' are treated as two different characters.

class Solution {

public String frequencySort(String s) {

if (s == null || s.length() == 0){

return s;

}

int[] hash = new int[256];

for (int i = 0; i < s.length(); i++){

hash[s.charAt(i)]++;

}

TreeMap<Integer, List<Character>> map = new TreeMap<Integer, List<Character>>();

for (int i = 0 ; i < hash.length; i++){

if (hash[i] > 0){

if (!map.containsKey(hash[i])){

map.put(hash[i], new ArrayList<Character>());

}

map.get(hash[i]).add((char)i);

}

}

StringBuilder sb = new StringBuilder();

while (map.size() > 0){

Map.Entry<Integer, List<Character>> entry = map.pollLastEntry();

for (Character c : entry.getValue()){

for (int i = 0; i < entry.getKey(); i++){

sb.append(c);

}

}

}

return sb.toString();

}

}

Method 2: PQ

class Solution {

class Pair{

char c;

int freq;

public Pair(char c, int freq){

this.c = c;

this.freq = freq;

}

}

public String frequencySort(String s) {

StringBuilder sb = new StringBuilder();

if (s == null || s.length() == 0){

return "";

}

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

map.put(c, map.getOrDefault(c, 0) + 1);

}

Queue<Pair> queue = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare(Pair a, Pair b){

return b.freq - a.freq;

}

});

for (Character ch : map.keySet()){

queue.offer(new Pair(ch, map.get(ch)));

}

while (!queue.isEmpty()){

Pair p = queue.poll();

int count = p.freq;

char c = p.c;

while (count > 0){

sb.append(c);

count--;

}

}

return sb.toString();

}

}

# 452\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are a number of spherical balloons spread in two-dimensional space. For each balloon, provided input is

the start and end coordinates of the horizontal diameter. Since it's horizontal, y-coordinates don't matter

and hence the x-coordinates of start and end of the diameter suffice. Start is always smaller than end.

There will be at most 104 balloons.

An arrow can be shot up exactly vertically from different points along the x-axis. A balloon with xstart

and xend bursts by an arrow shot at x if xstart ≤ x ≤ xend. There is no limit to the number of arrows that

can be shot. An arrow once shot keeps travelling up infinitely. The problem is to find the minimum number

of arrows that must be shot to burst all balloons.

Example:

Input:

[[10,16], [2,8], [1,6], [7,12]]

Output:

2

Explanation:

One way is to shoot one arrow for example at x = 6 (bursting the balloons [2,8] and [1,6]) and

another arrow at x = 11 (bursting the other two balloons).

The same as non-overlappping interval

Very close to merge interval, the only difference is to use Math.min instead of Math.max

class Solution {

public int findMinArrowShots(int[][] points) {

if (points == null || points.length == 0){

return 0;

}

Arrays.sort(points, new Comparator<int[]>(){

public int compare(int[] a, int[] b){

if (a[0] == b[0]){

return a[1] - b[1];

}

return a[0] - b[0];

}

});

int lastEnd = points[0][1];

int count = 1;

for (int i = 1; i < points.length; i++){

if (points[i][0] > lastEnd){

count++;

lastEnd = points[i][1];

}else{

lastEnd = Math.min(lastEnd, points[i][1]);

}

}

return count;

}

}

class Solution {

public int findMinArrowShots(int[][] points) {

int count = 0;

Arrays.sort(points, new Comparator<int[]>(){

public int compare (int[] a, int[] b){

if (a[0] == b[0]){

return a[1] - b[1];

}

return a[0] - b[0];

}

});

int[] last = null;

for (int[] p : points){

if (last == null || last[1] < p[0]){

count++;

last = p;

}else{

last[1] = Math.min(last[1], p[1]);

}

}

return count;

}

}

# 453\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty integer array of size n, find the minimum number of moves required to make

all array elements equal, where a move is incrementing n - 1 elements by 1.

Example:

Input:

[1,2,3]

Output:

3

Explanation:

Only three moves are needed (remember each move increments two elements):

[1,2,3] => [2,3,3] => [3,4,3] => [4,4,4]

class Solution {

public int minMoves(int[] nums) {

int sum = 0;

int n = nums.length;

int min = Integer.MAX\_VALUE;

for (int i = 0; i < nums.length; i++){

sum += nums[i];

min = Math.min(min, nums[i]);

}

return sum - n \* min;

}

}

# 454\_4SumII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given four lists A, B, C, D of integer values, compute how many tuples (i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l]

is zero.

To make problem a bit easier, all A, B, C, D have same length of N where 0 ≤ N ≤ 500.

All integers are in the range of -228 to 228 - 1 and the result is guaranteed to be at most 231 - 1.

Example:

Input:

A = [ 1, 2]

B = [-2,-1]

C = [-1, 2]

D = [ 0, 2]

Output:

2

Explanation:

The two tuples are:

1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0

2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0

Method 1:

Time complexity: O(n^2)

Space complexity: O(n^2)

class Solution {

public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {

int ans = 0;

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < C.length; i++){

for (int j = 0; j < D.length; j++){

map.put(C[i] + D[j], map.getOrDefault(C[i] + D[j], 0) + 1);

}

}

for (int i = 0; i < A.length; i++){

for (int j = 0; j < B.length; j++){

int target = -(A[i] + B[j]);

if (map.containsKey(target)){

ans += map.get(target);

}

}

}

return ans;

}

}

Method 2:

Time complexity: O(n^3) TLE

Space complexity: O(n)

class Solution {

public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {

int ans = 0;

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < D.length; i++){

map.put(D[i], map.getOrDefault(D[i], 0) + 1);

}

for (int i = 0; i < A.length; i++){

for (int j = 0; j < B.length; j++){

for (int k = 0; k < C.length; k++){

int target = -(A[i] + B[j] + C[k]);

if (map.containsKey(target)){

ans += map.get(target);

}

}

}

}

return ans;

}

}

Method 3:

Time complexity: O(n^3longn) TLE

Space complexity: O(1)

class Solution {

public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {

int ans = 0;

Arrays.sort(D);

for (int i = 0; i < A.length; i++){

for (int j = 0; j < B.length; j++){

for (int k = 0; k < C.length; k++){

int target = -(A[i] + B[j] + C[k]);

int len = binarySearch(D, target);

if(len != -1){

ans += len;

}

}

}

}

return ans;

}

private int binarySearch(int[] A, int target){

int first = 0;

int last = 0;

int start = 0;

int end = A.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (A[mid] >= target){

end = mid;

}else if (A[mid] < target){

start = mid;

}

}

if (A[start] == target){

first = start;

}else if (A[end] == target){

first = end;

}else{

return -1;

}

start = 0;

end = A.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (A[mid] <= target){

start = mid;

}else if (A[mid] > target){

end = mid;

}

}

if (A[end] == target){

last = end;

}else if (A[start] == target){

last = start;

}else{

return -1;

}

return last - first + 1;

}

}

# 455\_Assign.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Assume you are an awesome parent and want to give your children some cookies. But,

you should give each child at most one cookie. Each child i has a greed factor gi,

which is the minimum size of a cookie that the child will be content with; and each cookie j

has a size sj. If sj >= gi, we can assign the cookie j to the child i, and the child i will be content.

Your goal is to maximize the number of your content children and output the maximum number.

Note:

You may assume the greed factor is always positive.

You cannot assign more than one cookie to one child.

Example 1:

Input: [1,2,3], [1,1]

Output: 1

Explanation: You have 3 children and 2 cookies. The greed factors of 3 children are 1, 2, 3.

And even though you have 2 cookies, since their size is both 1, you could only make the child whose

greed factor is 1 content.

You need to output 1.

Example 2:

Input: [1,2], [1,2,3]

Output: 2

Explanation: You have 2 children and 3 cookies. The greed factors of 2 children are 1, 2.

You have 3 cookies and their sizes are big enough to gratify all of the children,

You need to output 2.

class Solution {

public int findContentChildren(int[] g, int[] s) {

Arrays.sort(g);

Arrays.sort(s);

int i = 0;

for (int j = 0; j < s.length && i < g.length; j++){

if (s[j] >= g[i]){

i++;

}

}

return i;

}

}

# 456\_132Pattern.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sequence of n integers a1, a2, ..., an, a 132 pattern is a subsequence ai, aj, ak such that i < j < k and ai < ak < aj. Design an algorithm that takes a list of n numbers as input and checks whether there is a 132 pattern in the list.

Note: n will be less than 15,000.

Example 1:

Input: [1, 2, 3, 4]

Output: False

Explanation: There is no 132 pattern in the sequence.

Example 2:

Input: [3, 1, 4, 2]

Output: True

Explanation: There is a 132 pattern in the sequence: [1, 4, 2].

Example 3:

Input: [-1, 3, 2, 0]

Output: True

Explanation: There are three 132 patterns in the sequence: [-1, 3, 2], [-1, 3, 0] and [-1, 2, 0].

Method 1: Better Brute Force

Time complexity: O(n^2)

Space complexity: O(1)

class Solution {

public boolean find132pattern(int[] nums) {

int min\_i = Integer.MAX\_VALUE;

for (int j = 0; j < nums.length-1; j++){

min\_i = Math.min(min\_i, nums[j]);

for (int k = j + 1; k < nums.length; k++){

if (nums[k] > min\_i && nums[k] < nums[j]){

return true;

}

}

}

return false;

}

}

https://leetcode.com/articles/132-pattern/

Method 2: Monotonic Decreasing Stack

To keep a track of these potential nums[k]nums[k] values for a particular nums[i],nums[j]nums[i],nums[j] considered currently,

we maintain a stackstack on which these potential nums[k]nums[k]'s satisfying the 132 criteria lie in a descending

order(minimum element on the top).We need not sort these elements on the stackstack,

but they'll be sorted automatically as we'll discuss along with the process.

we keep on popping the elements from the top of the stackstack till we find an element, stack[top]stack[top]

such that, stack[top] > min[j]stack[top]>min[j](or stack[top] > nums[i]stack[top]>nums[i]).

In summary:

Step 1: use min array to store the potential nums[i]

Step 2: use montonic array to store the potential nums[k]

Step 3: loop j backward to find nums[i] < nums[k] < nums[j]

class Solution {

public boolean find132pattern(int[] nums) {

if (nums.length < 3){

return false;

}

int[] minArray = new int[nums.length];

minArray[0] = nums[0];

for (int i = 1; i < nums.length; i++){

minArray[i] = Math.min(minArray[i-1], nums[i]);

}

Stack<Integer> stack = new Stack<>();

for (int j = nums.length - 1; j >= 0; j--){

if (nums[j] > minArray[j]){

while (!stack.isEmpty() && stack.peek() <= minArray[j]){//invariant: all the stack values are greater than minArray[j]

stack.pop();

}

if (!stack.isEmpty() && stack.peek() < nums[j]){

return true;

}

stack.push(nums[j]);

}

}

return false;

}

}

# 458\_Poor.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are 1000 buckets, one and only one of them contains poison, the rest are filled with water. They all look the same. If a pig drinks that poison it will die within 15 minutes. What is the minimum amount of pigs you need to figure out which bucket contains the poison within one hour.

Answer this question, and write an algorithm for the follow-up general case.

Follow-up:

If there are n buckets and a pig drinking poison will die within m minutes, how many pigs (x) you need to figure out the "poison" bucket within p minutes? There is exact one bucket with poison.

class Solution {

public int poorPigs(int buckets, int minutesToDie, int minutesToTest) {

int attempts = minutesToTest / minutesToDie + 1;

return (int) Math.ceil(Math.log(buckets) / Math.log(attempts));

}

}

https://leetcode.com/problems/poor-pigs/discuss/94273/Solution-with-detailed-explanation

# 46\_NumbersPermutations.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a collection of distinct numbers, return all possible permutations.

For example,

[1,2,3] have the following permutations:

[

[1,2,3],

[1,3,2],

[2,1,3],

[2,3,1],

[3,1,2],

[3,2,1]

]

class Solution {

public List<List<Integer>> permute(int[] nums) {

List<List<Integer>> result = new ArrayList<>();

if (nums == null || nums.length == 0){

return result;

}

dfs(result, new ArrayList<Integer>(), nums);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int[] nums){

if (item.size() == nums.length){

result.add(new ArrayList<Integer>(item));

return;

}

for (int i = 0; i < nums.length ; i++){

if (item.contains(nums[i])){ //arraylist contains time complexity O(n)

continue;

}

item.add(nums[i]);

dfs(result, item, nums);

item.remove(item.size() - 1);

}

}

}

Better

class Solution {

public List<List<Integer>> permute(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

Set<Integer> seen = new HashSet<>();

dfs(res, new ArrayList<Integer>(), seen, nums);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, Set<Integer> seen, int[] nums){

if(item.size() == nums.length){

res.add(new ArrayList<Integer>(item));

return;

}

for (int i = 0; i < nums.length; i++){

if (seen.contains(nums[i])){

continue;

}

item.add(nums[i]);

seen.add(nums[i]);

dfs(res, item, seen, nums);

seen.remove(nums[i]);

item.remove(item.size() - 1);

}

}

}

class Solution {

public List<List<Integer>> permute(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

boolean[] visited = new boolean[nums.length];

dfs(res, new ArrayList<Integer>(), visited, nums);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, boolean[] visited, int[] nums){

if(item.size() == nums.length){

res.add(new ArrayList<Integer>(item));

return;

}

for (int i = 0; i < nums.length; i++){

if (visited[i]){

continue;

}

item.add(nums[i]);

visited[i] = true;

dfs(res, item, visited, nums);

visited[i] = false;

item.remove(item.size() - 1);

}

}

}

# 461\_Hamming.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Given two integers x and y, calculate the Hamming distance.

Note:

0 ≤ x, y < 231.

Example:

Input: x = 1, y = 4

Output: 2

Explanation:

1 (0 0 0 1)

4 (0 1 0 0)

↑ ↑

The above arrows point to positions where the corresponding bits are different.

class Solution {

public int hammingDistance(int x, int y) {

int n = x ^ y;

int count = 0;

while (n != 0){

n = n & (n-1);

count++;

}

return count;

}

}

# 462\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty integer array, find the minimum number of moves required to make all array elements equal,

where a move is incrementing a selected element by 1 or decrementing a selected element by 1.

You may assume the array's length is at most 10,000.

Example:

Input:

[1,2,3]

Output:

2

Explanation:

Only two moves are needed (remember each move increments or decrements one element):

[1,2,3] => [2,2,3] => [2,2,2]

Note that The target number is the median number. So basically it is the problem to find median.

class Solution {

public int minMoves2(int[] nums) {

Arrays.sort(nums);

int count = 0;

int i = 0;

int j = nums.length - 1;

while (i < j){

count += nums[j] - nums[i];

i++;

j--;

}

return count;

}

}

# 463\_Island.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a map in form of a two-dimensional integer grid where 1 represents

land and 0 represents water. Grid cells are connected horizontally/vertically (not diagonally).

The grid is completely surrounded by water, and there is exactly one island (i.e., one or more

connected land cells). The island doesn't have "lakes" (water inside that isn't connected to the

water around the island). One cell is a square with side length 1. The grid is rectangular, width

and height don't exceed 100. Determine the perimeter of the island.

class Solution {

public int islandPerimeter(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int island = 0;

int neighbor = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1){

island++;

if (i+1 < m && grid[i+1][j] == 1){//count down

neighbor++;

}

if (j+1 < n && grid[i][j+1] == 1){//count right

neighbor++;

}

}

}

}

return island \* 4 - neighbor \* 2;

}

}

# 464\_Can.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In the "100 game," two players take turns adding, to a running total, any integer from 1..10. The player who first causes the running

total to reach or exceed 100 wins.

What if we change the game so that players cannot re-use integers?

For example, two players might take turns drawing from a common pool of numbers of 1..15 without replacement until they reach a

total >= 100.

Given an integer maxChoosableInteger and another integer desiredTotal, determine if the first player to move can force a win,

assuming both players play optimally.

You can always assume that maxChoosableInteger will not be larger than 20 and desiredTotal will not be larger than 300.

Example

Input:

maxChoosableInteger = 10

desiredTotal = 11

Output:

false

Explanation:

No matter which integer the first player choose, the first player will lose.

The first player can choose an integer from 1 up to 10.

If the first player choose 1, the second player can only choose integers from 2 up to 10.

The second player will win by choosing 10 and get a total = 11, which is >= desiredTotal.

Same with other integers chosen by the first player, the second player will always win.

Compare to Leetcode 486, this one has to use backtrack + memo because number is chosen in any order.

https://github.com/optimisea/Leetcode/blob/master/Java/486\_Predict.java

class Solution {

public boolean canIWin(int maxChoosableInteger, int desiredTotal) {

int sum = maxChoosableInteger \* (maxChoosableInteger + 1) / 2;

if (sum < desiredTotal){

return false;

}

if (desiredTotal <= 0){

return true;

}

boolean[] used = new boolean[maxChoosableInteger+1];

Map<Integer, Boolean> map = new HashMap<>();

return dfs(map, used, desiredTotal);//return if the first play can win

}

private boolean dfs(Map<Integer, Boolean> map, boolean[] used, int desiredTotal){

int key = format(used);

if (map.containsKey(key)){

return map.get(key);

}

if (desiredTotal <= 0){

return false;

}

for (int i = 1; i < used.length; i++){

if (!used[i]){

used[i] = true;

if (!dfs(map, used, desiredTotal - i)){//if the second player can't win

map.put(key, true);

used[i] = false; // need backtrack

return true;

}

used[i] = false;

}

}

map.put(key, false);

return false;

}

private int format(boolean[] used){

int num = 0;

for (int i = 0; i < used.length; i++){

num <<= 1;

if (used[i]){

num |= 1;

}

}

return num;

}

}

Best solution

Backtrack + meomo

class Solution {

public boolean canIWin(int maxChoosableInteger, int desiredTotal) {

int sum = maxChoosableInteger \* (maxChoosableInteger + 1) / 2;

if (sum < desiredTotal){

return false;

}

if (desiredTotal <= 0){

return true;

}

if (maxChoosableInteger >= desiredTotal){

return true;

}

Map<String, Boolean> map = new HashMap<>();

boolean[] visited = new boolean[maxChoosableInteger+1];

int[] total = new int[1];

return firstWin(maxChoosableInteger, visited, desiredTotal, total, map);

}

private boolean firstWin(int max, boolean[] visited, int target, int[] total, Map<String, Boolean> map){

String key = format(visited);

if (map.containsKey(key)){

return map.get(key);

}

if (total[0] >= target){

return false;

}

for (int i = 1; i <= max; i++){

if (visited[i]){

continue;

}

visited[i] = true;

total[0] += i;

if (!firstWin(max, visited, target, total, map)){

map.put(key, true);

visited[i] = false; // need backtrack

total[0] -= i; // need backtrack

return true;

}

total[0] -= i;

visited[i] = false;

}

map.put(key, false);

return false;

}

private String format(boolean[] visited){

StringBuilder sb = new StringBuilder();

for (int i = 0; i < visited.length; i++){

if (visited[i]){

sb.append(1);

}else{

sb.append(0);

}

}

return sb.toString();

}

}

# 465\_Optimal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A group of friends went on holiday and sometimes lent each other money. For example, Alice paid for Bill's lunch for $10. Then later

Chris gave Alice $5 for a taxi ride. We can model each transaction as a tuple (x, y, z) which means person x gave person y $z.

Assuming Alice, Bill, and Chris are person 0, 1, and 2 respectively (0, 1, 2 are the person's ID), the transactions can be represented

as [[0, 1, 10], [2, 0, 5]].

Given a list of transactions between a group of people, return the minimum number of transactions required to settle the debt.

Note:

A transaction will be given as a tuple (x, y, z). Note that x ≠ y and z > 0.

Person's IDs may not be linear, e.g. we could have the persons 0, 1, 2 or we could also have the persons 0, 2, 6.

Example 1:

Input:

[[0,1,10], [2,0,5]]

Output:

2

Explanation:

Person #0 gave person #1 $10.

Person #2 gave person #0 $5.

Two transactions are needed. One way to settle the debt is person #1 pays person #0 and #2 $5 each.

Example 2:

Input:

[[0,1,10], [1,0,1], [1,2,5], [2,0,5]]

Output:

1

Explanation:

Person #0 gave person #1 $10.

Person #1 gave person #0 $1.

Person #1 gave person #2 $5.

Person #2 gave person #0 $5.

Therefore, person #1 only need to give person #0 $4, and all debt is settled.

backtracking

class Solution {

public int minTransfers(int[][] transactions) {

Map<Integer, Integer> map = new HashMap<>();

for (int[] t : transactions) {

map.put(t[0], map.getOrDefault(t[0], 0) - t[2]);

map.put(t[1], map.getOrDefault(t[1], 0) + t[2]);

}

List<Integer> list = new ArrayList<>();

for (int val : map.values()){

list.add(val);

}

return settle(0, list);

}

private int settle(int start, List<Integer> debt) {

while (start < debt.size() && debt.get(start) == 0){

start++;

}

if (start == debt.size())

return 0;

int res = Integer.MAX\_VALUE;

for (int i = start; i < debt.size(); i++){

if (debt.get(start) \* debt.get(i) < 0) {

debt.set(i, debt.get(i) + debt.get(start));

res = Math.min(res, 1 + settle(start + 1, debt));

debt.set(i, debt.get(i) - debt.get(start));

}

}

return res;

}

}

# 467\_Unique.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Consider the string s to be the infinite wraparound string of "abcdefghijklmnopqrstuvwxyz", so s will look like this:

"...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".

Now we have another string p. Your job is to find out how many unique non-empty substrings of p are present in s. In particular,

your input is the string p and you need to output the number of different non-empty substrings of p in the string s.

Note: p consists of only lowercase English letters and the size of p might be over 10000.

Example 1:

Input: "a"

Output: 1

Explanation: Only the substring "a" of string "a" is in the string s.

Example 2:

Input: "cac"

Output: 2

Explanation: There are two substrings "a", "c" of string "cac" in the string s.

Example 3:

Input: "zab"

Output: 6

Explanation: There are six substrings "z", "a", "b", "za", "ab", "zab" of string "zab" in the string s.

Method 1: DP

class Solution {

public int findSubstringInWraproundString(String p) {

int[] dp = new int[26]; //ending with char

int maxCount = 0;

for (int i = 0; i < p.length(); i++){

if (i > 0 && (p.charAt(i) - p.charAt(i-1) == 1 || p.charAt(i-1) - p.charAt(i) == 25)){

maxCount++;

}else{

maxCount = 1;

}

int index = p.charAt(i) - 'a';

dp[index] = Math.max(dp[index], maxCount);

}

int sum = 0;

for (int i = 0; i < 26; i++){

sum += dp[i];

}

return sum;

}

}

https://leetcode.com/problems/unique-substrings-in-wraparound-string/discuss/95439/Concise-Java-solution-using-DP

# 468\_Validate.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a function to check whether an input string is a valid IPv4 address or IPv6 address or neither.

IPv4 addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255,

separated by dots ("."), e.g.,172.16.254.1;

Besides, leading zeros in the IPv4 is invalid. For example, the address 172.16.254.01 is invalid.

IPv6 addresses are represented as eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by

colons (":"). For example, the address 2001:0db8:85a3:0000:0000:8a2e:0370:7334 is a valid one. Also, we could omit some leading zeros

among four hexadecimal digits and some low-case characters in the address to upper-case ones, so 2001:db8:85a3:0:0:8A2E:0370:7334 is

also a valid IPv6 address(Omit leading zeros and using upper cases).

However, we don't replace a consecutive group of zero value with a single empty group using two consecutive colons (::) to pursue

simplicity. For example, 2001:0db8:85a3::8A2E:0370:7334 is an invalid IPv6 address.

Besides, extra leading zeros in the IPv6 is also invalid. For example, the address 02001:0db8:85a3:0000:0000:8a2e:0370:7334 is invalid.

Note: You may assume there is no extra space or special characters in the input string.

Example 1:

Input: "172.16.254.1"

Output: "IPv4"

Explanation: This is a valid IPv4 address, return "IPv4".

Example 2:

Input: "2001:0db8:85a3:0:0:8A2E:0370:7334"

Output: "IPv6"

Explanation: This is a valid IPv6 address, return "IPv6".

Example 3:

Input: "256.256.256.256"

Output: "Neither"

Explanation: This is neither a IPv4 address nor a IPv6 address.

class Solution {

public String validIPAddress(String IP) {

if (isIPv4(IP)) {

return "IPv4";

}

if (isIPv6(IP.toUpperCase())) {

return "IPv6";

}

return "Neither";

}

private boolean isIPv4(String IP) {

int cnt = 0;

for (char ch : IP.toCharArray()) {

if (ch == '.') {

cnt++;

}

}

if (cnt != 3) {

return false;

}

String[] fields = IP.split("\\.");

if (fields.length != 4) {

return false;

}

for (String field : fields) {

if (field.isEmpty() || field.length() > 3) {

return false;

}

int sz = field.length();

for (int i = 0; i < sz; ++i) {

if (!Character.isDigit(field.charAt(i))) {

return false;

}

}

int num = Integer.valueOf(field);

if (!String.valueOf(num).equals(field) || num < 0 || num > 255) {

return false;

}

}

return true;

}

private boolean isIPv6(String IP) {

int cnt = 0;

for (char ch : IP.toCharArray()) {

if (ch == ':') {

cnt++;

}

}

if (cnt != 7) {

return false;

}

String[] fields = IP.split(":");

if (fields.length != 8) {

return false;

}

for (String field : fields) {

if (field.isEmpty() || field.length() > 4) {

return false;

}

int sz = field.length();

for (int i = 0; i < sz; ++i) {

if (!Character.isDigit(field.charAt(i)) && (field.charAt(i) < 'A' || field.charAt(i) > 'F')) {

return false;

}

}

}

return true;

}

}

class Solution {

public String validIPAddress(String IP) {

if (isIPv4(IP)){

return "IPv4";

}

if (isIPv6(IP)){

return "IPv6";

}

return "Neither";

}

private boolean isIPv4(String IP){

int count = 0;

for (char c : IP.toCharArray()){

if (c == '.'){

count++;

}

}

if (count != 3){

return false;

}

String[] strs = IP.split("\\.");

if (strs.length != 4){

return false;

}

for (String str : strs){

if (str.length() == 0 || str.length() > 3){

return false;

}

for (char c : str.toCharArray()){

if (!Character.isDigit(c)){

return false;

}

}

int num = Integer.parseInt(str);

if (!str.equals(String.valueOf(num)) || num < 0 || num > 255){

return false;

}

}

return true;

}

private boolean isIPv6(String IP){

int count = 0;

for (char c : IP.toCharArray()){

if (c == ':'){

count++;

}

}

if (count != 7){

return false;

}

String[] strs = IP.split(":");

if (strs.length != 8){

return false;

}

for (String str : strs){

if (str.length() == 0 || str.length() > 4){

return false;

}

for (char c : str.toCharArray()){

if (Character.isLetter(c)){

if (Character.isUpperCase(c)){

if (c < 'A' || c > 'F'){

return false;

}

}else{

if (c < 'a' || c > 'f'){

return false;

}

}

}else if (!Character.isDigit(c)){

return false;

}

}

}

return true;

}

}

# 47\_PermutationsII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,

[1,1,2] have the following unique permutations:

[

[1,1,2],

[1,2,1],

[2,1,1]

]

class Solution {

public List<List<Integer>> permuteUnique(int[] nums) {

List<List<Integer>> result = new ArrayList<>();

if (nums == null || nums.length == 0){

return result;

}

Arrays.sort(nums);

boolean[] marked = new boolean[nums.length];

dfs(result, new ArrayList<Integer>(), nums, marked);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int[] nums, boolean[] marked){

if (item.size() == nums.length){

result.add(new ArrayList<Integer>(item));

return;

}

for (int i = 0; i < nums.length; i++){

if (i != 0 && nums[i] == nums[i-1] && !marked[i-1]){

continue;

}

if (!marked[i]){

item.add(nums[i]);

marked[i] = true;

dfs(result, item, nums, marked);

item.remove(item.size() - 1);

marked[i] = false;

}

}

}

}

Better version

class Solution {

public List<List<Integer>> permuteUnique(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

Arrays.sort(nums);

boolean[] visited = new boolean[nums.length];

dfs(res, new ArrayList<Integer>(), visited, nums);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, boolean[] visited, int[] nums){

if (item.size() == nums.length){

res.add(new ArrayList<Integer>(item));

return;

}

for (int i = 0; i < nums.length; i++){

if (visited[i] || i > 0 && nums[i] == nums[i-1] && !visited[i-1]){

continue;

}

visited[i] = true;

item.add(nums[i]);

dfs(res, item, visited, nums);

item.remove(item.size() - 1);

visited[i] = false;

}

}

}

Check the similar question: 996. Number of Squareful Arrays

https://github.com/optimisea/Leetcode/blob/master/Java/996\_Number.java

# 470\_Implement.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a function rand7 which generates a uniform random integer in the range 1 to 7, write a function rand10 which generates a uniform random integer in the range 1 to 10.

Do NOT use system's Math.random().

Example 1:

Input: 1

Output: [7]

Example 2:

Input: 2

Output: [8,4]

Example 3:

Input: 3

Output: [8,1,10]

Note:

rand7 is predefined.

Each testcase has one argument: n, the number of times that rand10 is called.

Follow up:

What is the expected value for the number of calls to rand7() function?

Could you minimize the number of calls to rand7()?

/\*\*

\* The rand7() API is already defined in the parent class SolBase.

\* public int rand7();

\* @return a random integer in the range 1 to 7

\*/

class Solution extends SolBase {

public int rand10() {

int row, col, idx;

do {

row = rand7();

col = rand7();

idx = col + (row - 1) \* 7;

} while (idx > 40);

return 1 + (idx - 1) % 10;

}

}

https://leetcode.com/problems/implement-rand10-using-rand7/solution/

471\_Encode.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty string, encode the string such that its encoded length is the shortest.

The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets is being repeated exactly k times.

Note:

k will be a positive integer and encoded string will not be empty or have extra space.

You may assume that the input string contains only lowercase English letters. The string's length is at most 160.

If an encoding process does not make the string shorter, then do not encode it. If there are several solutions,

return any of them is fine.

Example 1:

Input: "aaa"

Output: "aaa"

Explanation: There is no way to encode it such that it is shorter than the input string, so we do not encode it.

Example 2:

Input: "aaaaa"

Output: "5[a]"

Explanation: "5[a]" is shorter than "aaaaa" by 1 character.

Example 3:

Input: "aaaaaaaaaa"

Output: "10[a]"

Explanation: "a9[a]" or "9[a]a" are also valid solutions, both of them have the same length = 5, which is the same as "10[a]".

Example 4:

Input: "aabcaabcd"

Output: "2[aabc]d"

Explanation: "aabc" occurs twice, so one answer can be "2[aabc]d".

Example 5:

Input: "abbbabbbcabbbabbbc"

Output: "2[2[abbb]c]"

Explanation: "abbbabbbc" occurs twice, but "abbbabbbc" can also be encoded to "2[abbb]c", so one answer can be "2[2[abbb]c]".

Similar Questions

Decode StringNumber of Atoms

Method: Dynamic Programming:

Time complexity: O(N^4) since replaceAll takes O(N) time

dp[i][j] = string from index i to index j in encoded form

diff means the length diff between i and j

diff is used because we only need to fill the top left half dp[i][j] matrix and return dp[0][n-1]

dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j]) or if we can find some pattern in string from i to j which will

result in more less length.

class Solution {

public String encode(String s) {

int n = s.length();

String[][] dp = new String[n][n];//dp[i][j] = string from index i to index j in encoded form

for (int diff = 0; diff < n; diff++){//diff means the length diff between i and j

for (int i = 0; i < n - diff; i++){

int j = i + diff;

String substr = s.substring(i, j+1);

if (diff <= 3){

dp[i][j] = substr;

}else{

dp[i][j] = substr;

//Loop for trying all results that we get after dividing the strings into 2 and combine the results of 2 substrings

for (int k = i; k < j; k++){

if ((dp[i][k] + dp[k+1][j]).length() < dp[i][j].length()){

dp[i][j] = dp[i][k] + dp[k+1][j];

}

}

// Loop for checking if string can itself found some pattern in it which could be repeated.

int subLen = substr.length();

for (int k = 0; k < subLen; k++){

String strK = substr.substring(0, k+1);

int kLen = strK.length();

if (subLen % kLen == 0 && substr.replaceAll(strK, "").length() == 0){

String str = subLen/kLen + "[" + dp[i][i+k] + "]";//note that we need use dp[i][i+k] instead of strK

if (str.length() < dp[i][j].length()){

dp[i][j] = str;

}

}

}

}

}

}

return dp[0][n-1];

}

}

472\_Concatenated.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of words (without duplicates), please write a program that returns all concatenated words in the given list of words.

A concatenated word is defined as a string that is comprised entirely of at least two shorter words in the given array.

Example:

Input: ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]

Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]

Explanation: "catsdogcats" can be concatenated by "cats", "dog" and "cats";

"dogcatsdog" can be concatenated by "dog", "cats" and "dog";

"ratcatdogcat" can be concatenated by "rat", "cat", "dog" and "cat".

Note:

The number of elements of the given array will not exceed 10,000

The length sum of elements in the given array will not exceed 600,000.

All the input string will only include lower case letters.

The returned elements order does not matter.

Method 1: DP solution 284ms

use the code from word break I

Time complexity: O(m^2 \* nlogn)

class Solution {

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> res = new ArrayList<>();

Set<String> set = new HashSet<>();

Arrays.sort(words, new Comparator<String>(){

public int compare (String s1, String s2){

return s1.length() - s2.length();

}

});

for (String word : words){

if (word.length() != 0 && canBreak(word, set)){

res.add(word);

}

set.add(word);

}

return res;

}

private boolean canBreak(String word, Set<String> set){

int len = word.length();

boolean[] dp = new boolean[len+1];

dp[0] = true;

for (int i = 1; i <= len; i++){

for (int j = 0; j < i; j++){

if (dp[j] && set.contains(word.substring(j, i))){

dp[i] = true;

break;

}

}

}

return dp[len];

}

}

Time complexity: O(m^2 \* n)

class Solution {

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> res = new ArrayList<>();

Set<String> set = new HashSet<>();

for (String word : words){

if (word.length() != 0){

set.add(word);

}

}

for (String word : words){

if (word.length() != 0 && canBreak(word, set)){

res.add(word);

}

}

return res;

}

private boolean canBreak(String word, Set<String> set){

int len = word.length();

boolean[] dp = new boolean[len+1];

dp[0] = true;

for (int i = 1; i <= len; i++){

for (int j = 0; j < i; j++){

if (dp[j] && set.contains(word.substring(j, i)) && i - j != len){

dp[i] = true;

break;

}

}

}

return dp[len];

}

}

Method 2: DFS + Trie 54ms

https://leetcode.com/problems/concatenated-words/discuss/95644/102ms-java-Trie-+-DFS-solution.-With-explanation-easy-to-understand.

class Solution {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode() {

children = new TrieNode[26];

isEnd = false;

}

}

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> res = new ArrayList<>();

//build trie

TrieNode root = new TrieNode();

for (String word : words){

if (word.length() == 0){

continue;

}

TrieNode node = root;

for (char c : word.toCharArray()){

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

//check

for (String word : words){

if (word.length() == 0){

continue;

}

if (isValid(root, word, 0, 0)){

res.add(word);

}

}

return res;

}

private boolean isValid(TrieNode root, String word, int start, int count){

TrieNode node = root;

for (int i = start; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

if (node.children[c - 'a'].isEnd){

if (i == word.length() - 1){

return count >= 1;

}

if (isValid(root, word, i+1, count+1)){

return true;

}

}

node = node.children[c - 'a'];

}

return false;

}

}

Better solution: O(mn)

class Solution {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode(){

children = new TrieNode[26];

isEnd = false;

}

}

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> res = new ArrayList<>();

//build Trie

TrieNode root = new TrieNode();

for (String word : words){

if (word.length() == 0){

continue;

}

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

//search

for (String word : words){

if (word.length() == 0){

continue;

}

if (isValid(root, word, 0, 0)){

res.add(word);

}

}

return res;

}

private boolean isValid(TrieNode root, String word, int start, int count){

if (start == word.length()){

return count > 1;

}

TrieNode node = root;

for (int i = start; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c- 'a'];

if (node.isEnd){

if (isValid(root, word, i+1, count+1)){

return true;

}

}

}

return false;

}

}

473\_Matchsticks.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Remember the story of Little Match Girl? By now, you know exactly what matchsticks the little match girl has, please find out a way

you can make one square by using up all those matchsticks. You should not break any stick, but you can link them up, and each

matchstick must be used exactly one time.

Your input will be several matchsticks the girl has, represented with their stick length. Your output will either be true or false,

to represent whether you could make one square using all the matchsticks the little match girl has.

Example 1:

Input: [1,1,2,2,2]

Output: true

Explanation: You can form a square with length 2, one side of the square came two sticks with length 1.

Example 2:

Input: [3,3,3,3,4]

Output: false

Explanation: You cannot find a way to form a square with all the matchsticks.

Note:

The length sum of the given matchsticks is in the range of 0 to 10^9.

The length of the given matchstick array will not exceed 15.

https://leetcode.com/problems/matchsticks-to-square/discuss/95752/Java-DFS-solution-with-various-optimizations-(sorting-sequential-partition-DP)

https://leetcode.com/problems/matchsticks-to-square/discuss/95729/Java-DFS-Solution-with-Explanation

The idea is to assign one number into different sums and try different combinations.

numIndex is increased by one to ensure only one visit.

Method 1: without sorting... 423ms

class Solution {

public boolean makesquare(int[] nums) {

if (nums == null || nums.length == 0){

return false;

}

int sum = 0;

int max = 0;

for (int i : nums){

sum += i;

max = Math.max(max, i);

}

if (sum % 4 != 0 || max > sum / 4){

return false;

}

int side = sum / 4;

int[] sums = new int[4];

return makeSqure(nums, nums.length - 1, sums, side);

}

private boolean makeSqure(int[] nums, int numsIndex, int[] sums, int target){

if (numsIndex < 0){

if (sums[0] == target && sums[1] == target && sums[2] == target && sums[3] == target){

return true;

}

return false;

}

for (int i = 0; i < 4; i++){

if (sums[i] + nums[numsIndex] > target){

continue;

}

sums[i] += nums[numsIndex];

if (makeSqure(nums, numsIndex-1, sums, target)){

return true;

}

sums[i] -= nums[numsIndex];

}

return false;

}

}

Method 2: with sorting 40ms refer to Leetcoe 698 for the genelized format for rationale

class Solution {

public boolean makesquare(int[] nums) {

if (nums == null || nums.length == 0){

return false;

}

int sum = 0;

int max = 0;

for (int i : nums){

sum += i;

max = Math.max(max, i);

}

if (sum % 4 != 0 || max > sum / 4){

return false;

}

int side = sum / 4;

int[] sums = new int[4];

Arrays.sort(nums);

return makeSqure(nums, nums.length - 1, sums, side);

}

private boolean makeSqure(int[] nums, int numsIndex, int[] sums, int target){

if (numsIndex < 0){

if (sums[0] == target && sums[1] == target && sums[2] == target && sums[3] == target){

return true;

}

return false;

}

for (int i = 0; i < 4; i++){

if (sums[i] + nums[numsIndex] > target){

continue;

}

sums[i] += nums[numsIndex];

if (makeSqure(nums, numsIndex-1, sums, target)){

return true;

}

sums[i] -= nums[numsIndex];

}

return false;

}

}

Backtracking:

refer to Leetcoe 698 for the genelized format in which k = 4

class Solution {

public boolean makesquare(int[] nums) {

if (nums == null || nums.length == 0){

return false;

}

int sum = 0;

for (int i : nums){

sum += i;

}

if (sum % 4 != 0){

return false;

}

int target = sum / 4;

int[] sums = new int[4];

return canPartition(nums, sums, target, nums.length - 1);

}

private boolean canPartition(int[] nums, int[] sums, int target, int pos){

if (pos < 0){

for (int i = 0; i < 4; i++){

if (sums[i] != target){

return false;

}

}

return true;

}

for (int i = 0; i < 4; i++){

if (sums[i] + nums[pos] > target){

continue;

}

sums[i] += nums[pos];

if (canPartition(nums, sums, target, pos - 1)){

return true;

}

sums[i] -= nums[pos];

}

return false;

}

}

474\_Ones.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In the computer world, use restricted resource you have to generate maximum benefit is what we always want to pursue.

For now, suppose you are a dominator of m 0s and n 1s respectively. On the other hand, there is an array with strings consisting

of only 0s and 1s.

Now your task is to find the maximum number of strings that you can form with given m 0s and n 1s. Each 0 and 1 can be used

at most once.

Note:

The given numbers of 0s and 1s will both not exceed 100

The size of given string array won't exceed 600.

Example 1:

Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3

Output: 4

Explanation: This are totally 4 strings can be formed by the using of 5 0s and 3 1s, which are “10,”0001”,”1”,”0”

Example 2:

Input: Array = {"10", "0", "1"}, m = 1, n = 1

Output: 2

Explanation: You could form "10", but then you'd have nothing left. Better form "0" and "1".

Method 1: recursion OLE, so need use DP to save solved problem to remove duplicate calculation

class Solution {

public int findMaxForm(String[] strs, int m, int n) {

return maxForm(strs, m, n, 0);

}

private int maxForm(String[] strs, int m , int n, int start){

if (start == strs.length || m+n == 0){

return 0;

}

int ones = 0;

int zeros = 0;

String s = strs[start];

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (c == '0'){

zeros++;

}else if (c == '1'){

ones++;

}

}

int maxWithCurrStr= 0;

if (m >= zeros && n >= ones){

maxWithCurrStr = maxForm(strs, m - zeros, n - ones, start + 1) + 1;

}

int maxWithoutCurrStr = maxForm(strs, m, n, start + 1);

return Math.max(maxWithCurrStr, maxWithoutCurrStr);

}

}

https://leetcode.com/problems/ones-and-zeroes/discuss/95845/Easy-to-understand-Recursive-Solutions-in-Java-with-Explanation

Method 2:

class Solution {

private int[][][] hash;

public int findMaxForm(String[] strs, int m, int n) {

hash = new int[m+1][n+1][strs.length];

return maxForm(strs, m, n, 0);

}

private int maxForm(String[] strs, int m , int n, int start){

if (start == strs.length || m+n == 0){

return 0;

}

if (hash[m][n][start] != 0){

return hash[m][n][start];

}

int ones = 0;

int zeros = 0;

String s = strs[start];

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (c == '0'){

zeros++;

}else if (c == '1'){

ones++;

}

}

int maxWithCurrStr= 0;

if (m >= zeros && n >= ones){

maxWithCurrStr = maxForm(strs, m - zeros, n - ones, start + 1) + 1;

}

int maxWithoutCurrStr = maxForm(strs, m, n, start + 1);

hash[m][n][start] = Math.max(maxWithCurrStr, maxWithoutCurrStr);

return hash[m][n][start];

}

}

https://leetcode.com/problems/ones-and-zeroes/discuss/95807/0-1-knapsack-detailed-explanation.

Method 3: Best,similar to a backpack(Partition Equal Subset Sum) problem

class Solution {

public int findMaxForm(String[] strs, int m, int n) {

int l = strs.length;

int[][][] dp = new int[l+1][m+1][n+1];

for (int i = 1; i <= l; i++){

int zeros = 0;

int ones = 0;

String s = strs[i-1];

for (int j = 0; j < s.length(); j++){

if (s.charAt(j) == '0'){

zeros++;

}else{

ones++;

}

}

for (int j = 0; j <= m; j++){

for (int k = 0; k <= n; k++){

if (j >= zeros && k >= ones){

dp[i][j][k] = Math.max(dp[i-1][j-zeros][k-ones] + 1, dp[i-1][j][k]);

}else{

dp[i][j][k] = dp[i-1][j][k];

}

}

}

}

return dp[l][m][n];

}

}

Method 4: reduce space complexity by using rolling array

class Solution {

public int findMaxForm(String[] strs, int m, int n) {

int l = strs.length;

int[][][] dp = new int[2][m+1][n+1];

for (int i = 1; i <= l; i++){

int zeros = 0;

int ones = 0;

String s = strs[i-1];

for (int j = 0; j < s.length(); j++){

if (s.charAt(j) == '0'){

zeros++;

}else{

ones++;

}

}

for (int j = 0; j <= m; j++){

for (int k = 0; k <= n; k++){

if (j >= zeros && k >= ones){

dp[i%2][j][k] = Math.max(dp[(i-1)%2][j-zeros][k-ones] + 1, dp[(i-1)%2][j][k]);

}else{

dp[i%2][j][k] = dp[(i-1)%2][j][k];

}

}

}

}

return dp[l%2][m][n];

}

}

Method 5:

class Solution {

public int findMaxForm(String[] strs, int m, int n) {

int l = strs.length;

int[][] dp = new int[m+1][n+1];

for (int i = 0; i < l; i++){

int zeros = 0;

int ones = 0;

String s = strs[i];

for (int j = 0; j < s.length(); j++){

if (s.charAt(j) == '0'){

zeros++;

}else{

ones++;

}

}

for (int j = m; j >=0 ; j--){

for (int k = n; k >= 0; k--){

if (j >= zeros && k >= ones){

dp[j][k] = Math.max(dp[j-zeros][k-ones] + 1, dp[j][k]);

}else{

dp[j][k] = dp[j][k];

}

}

}

}

return dp[m][n];

}

}

475\_Heaters.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Winter is coming! Your first job during the contest is to design a standard heater with fixed warm radius to warm all the houses.

Now, you are given positions of houses and heaters on a horizontal line, find out minimum radius of heaters so that all houses could be covered by those heaters.

So, your input will be the positions of houses and heaters seperately, and your expected output will be the minimum radius standard of heaters.

Note:

Numbers of houses and heaters you are given are non-negative and will not exceed 25000.

Positions of houses and heaters you are given are non-negative and will not exceed 10^9.

As long as a house is in the heaters' warm radius range, it can be warmed.

All the heaters follow your radius standard and the warm radius will the same.

Example 1:

Input: [1,2,3],[2]

Output: 1

Explanation: The only heater was placed in the position 2, and if we use the radius 1 standard, then all the houses can be warmed.

Example 2:

Input: [1,2,3,4],[1,4]

Output: 1

Explanation: The two heater was placed in the position 1 and 4. We need to use radius 1 standard, then all the houses can be warmed.

Method 1: O(mn + Math.max(nlogn, mlogm)) TLE

class Solution {

public int findRadius(int[] houses, int[] heaters) {

Arrays.sort(houses);

Arrays.sort(heaters);

int i = 0;

int ans = 0;

while (i < houses.length){

int j = 0;

while (j < heaters.length - 1 && Math.abs(heaters[j+1] - houses[i]) <= Math.abs(heaters[j] - houses[i])){

j++;

}

ans = Math.max(ans, Math.abs(heaters[j] - houses[i]));

i++;

}

return ans;

}

}

https://leetcode.com/problems/heaters/discuss/95881/Simple-Java-Solution-with-2-Pointers

Based on 2 pointers, the idea is to find the nearest heater for each house, by comparing the next heater with the current heater.

Method 2: O(m+n + Math.max(nlogn, mlogm)), there is no need to reset j to be 0 every loop of house.

class Solution {

public int findRadius(int[] houses, int[] heaters) {

Arrays.sort(houses);

Arrays.sort(heaters);

int i = 0;

int ans = 0;

int j = 0;

while (i < houses.length){

while (j < heaters.length - 1 && Math.abs(heaters[j+1] - houses[i]) <= Math.abs(heaters[j] - houses[i])){

j++;

}

ans = Math.max(ans, Math.abs(heaters[j] - houses[i]));

i++;

}

return ans;

}

}

476\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer, output its complement number. The complement strategy is to flip the bits of

its binary representation.

Note:

The given integer is guaranteed to fit within the range of a 32-bit signed integer.

You could assume no leading zero bit in the integer’s binary representation.

Example 1:

Input: 5

Output: 2

Explanation: The binary representation of 5 is 101 (no leading zero bits), and its complement is 010.

So you need to output 2.

Example 2:

Input: 1

Output: 0

Explanation: The binary representation of 1 is 1 (no leading zero bits), and its complement is 0. So you need to output 0.

https://leetcode.com/problems/number-complement/discuss/95992/Java-1-line-bit-manipulation-solution

class Solution {

public int findComplement(int num) {

int mask = (Integer.highestOneBit(num) << 1) - 1;

num = ~num;

return num & mask;

}

}

Create a bit mask which has N bits of 1 from RIGHTMOST. In above example, the mask is 111.

And we can use the decent Java built-in function Integer.highestOneBit to get the LEFTMOST bit of 1, left shift one,

and then minus one.

Please remember this wonderful trick to create bit masks with N ones at RIGHTMOST, you will be able to use it later.

477\_SurroundedRegions.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O''s into 'X''s in that surrounded region.

Have you met this question in a real interview? Yes

Example

X X X X

X O O X

X X O X

X O X X

After capture all regions surrounded by 'X', the board should be:

X X X X

X X X X

X X X X

X O X X

• 找到所有被 ‘X’ 围绕的区域 == 找到被平原围绕的盆地

– 反向思维:找到没有被平原围绕的盆地

public class Solution {

/\*

\* @param board: board a 2D board containing 'X' and 'O'

\* @return: nothing

\*/

int n, m;

public void surroundedRegions(char[][] board) {

n = board.length;

if (n == 0){

return;

}

m = board[0].length;

for (int i = 0; i < n; i++){

bfs(board, i, 0);

bfs(board, i, m - 1);

}

for (int j = 0; j < m; j++){

bfs(board, 0, j);

bfs(board, n - 1, j);

}

for (int i = 0; i < n; i++){

for (int j = 0; j < m; j++){

if (board[i][j] == 'W'){

board[i][j] = 'O';

}else{

board[i][j] = 'X';

}

}

}

}

private void bfs(char[][] board, int x, int y){

if (board[x][y] != 'O'){

return;

}

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

int[] dx = {0, 1, 0 , -1};

int[] dy = {1, 0, -1, 0};

board[x][y] = 'W';

qx.offer(x);

qy.offer(y);

while (!qx.isEmpty()){

int nx = qx.poll();

int ny = qy.poll();

for (int i = 0; i < 4; i++){

int newx = nx + dx[i];

int newy = ny + dy[i];

if (newx >= 0 && newx < n && newy >= 0 && newy < m

&& board[newx][newy] == 'O'){

board[newx][newy] = 'W';

qx.offer(newx);

qy.offer(newy);

}

}

}

}

}

Method 2: one bfs

class Solution {

public void solve(char[][] board) {

if (board == null || board.length == 0 || board[0].length == 0){

return;

}

int n = board.length;

int m = board[0].length;

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

int[] dx = {1, 0 , -1 ,0};

int[] dy = {0, 1, 0, -1};

for (int i = 0; i < m; i++){

if (board[0][i] == 'O'){

qx.offer(0);

qy.offer(i);

board[0][i] = 'W';

}

if (board[n-1][i] == 'O'){

qx.offer(n-1);

qy.offer(i);

board[n-1][i] = 'W';

}

}

for (int i = 0; i < n; i++){

if (board[i][0] == 'O'){

qx.offer(i);

qy.offer(0);

board[i][0] = 'W';

}

if (board[i][m-1] == 'O'){

qx.offer(i);

qy.offer(m-1);

board[i][m-1] = 'W';

}

}

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

for (int i = 0; i < 4; i++){

int nx = cx + dx[i];

int ny = cy + dy[i];

if (inBound(n, m, nx, ny) && board[nx][ny] == 'O'){

qx.offer(nx);

qy.offer(ny);

board[nx][ny] = 'W';

}

}

}

for (int i = 0; i < n; i++){

for (int j = 0; j < m; j++){

if (board[i][j] == 'W'){

board[i][j] = 'O';

}else{

board[i][j] = 'X';

}

}

}

}

private boolean inBound(int n, int m, int nx, int ny){

return nx >= 0 && nx < n && ny >= 0 && ny < m;

}

}

Better version:

class Solution {

public void solve(char[][] board) {

if (board == null || board.length == 0){

return;

}

int m = board.length;

int n = board[0].length;

for (int i = 0; i < m; i++){

if (board[i][0] == 'O'){

bfs(board, i, 0);

}

if (board[i][n-1] == 'O'){

bfs(board, i, n-1);

}

}

for (int j = 0; j < n; j++){

if (board[0][j] == 'O'){

bfs(board, 0, j);

}

if (board[m-1][j] == 'O'){

bfs(board, m-1, j);

}

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (board[i][j] == 'O'){

board[i][j] = 'X';

}else if (board[i][j] == 'W'){

board[i][j] = 'O';

}

}

}

}

private void bfs(char[][] board, int x, int y){

int m = board.length;

int n = board[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

queue.offer(new int[]{x, y});

board[x][y] = 'W';

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && board[nx][ny] == 'O'){

queue.offer(new int[]{nx, ny});

board[nx][ny] = 'W';

}

}

}

}

}

477\_Total.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Now your job is to find the total Hamming distance between all pairs of the given numbers.

Example:

Input: 4, 14, 2

Output: 6

Explanation: In binary representation, the 4 is 0100, 14 is 1110, and 2 is 0010 (just

showing the four bits relevant in this case). So the answer will be:

HammingDistance(4, 14) + HammingDistance(4, 2) + HammingDistance(14, 2) = 2 + 2 + 2 = 6.

Note:

Elements of the given array are in the range of 0 to 10^9

Length of the array will not exceed 10^4.

Method 1: Brute Force (TLE)

class Solution {

public int totalHammingDistance(int[] nums) {

int count = 0;

for (int i = 0; i < nums.length; i++){

for (int j = i+1; j < nums.length; j++){

count += hammingDistance(nums[i], nums[j]);

}

}

return count;

}

private int hammingDistance(int x, int y){

int n = x ^ y;

int count = 0;

while (n != 0){

n = n & (n-1);

count++;

}

return count;

}

}

Method 2: Best solution

O(n)

https://leetcode.com/problems/total-hamming-distance/discuss/96243/Share-my-O(n)-C++-bitwise-solution-with-thinking-process-and-explanation

https://leetcode.com/problems/total-hamming-distance/discuss/96226/Java-O(n)-time-O(1)-Space

class Solution {

public int totalHammingDistance(int[] nums) {

int total = 0;

int n = nums.length;

for (int i = 0; i < 32; i++){

int bitCount = 0;

for (int j = 0; j < n; j++){

bitCount += (nums[j] >>> i) & 1;

}

total += bitCount \* (n-bitCount);

}

return total;

}

}

class Solution {

public int totalHammingDistance(int[] nums) {

int res = 0;

for (int i = 0; i < 32; i++){

int zeros = 0;

int ones = 0;

for (int j = 0 ;j < nums.length; j++){

if (((nums[j] >> i) & 1) == 1){

ones++;

}else{

zeros++;

}

}

res += zeros \* ones;

}

return res;

}

}

The total Hamming distance is constructed bit by bit in this approach.

Let's take a series of number: a1, a2, a3,..., an

Just think about all the Least Significant Bit (LSB) of a(k) (1 ≤ k ≤ n).

How many Hamming distance will they bring to the total?

If a pair of number has same LSB, the total distance will get 0.

If a pair of number has different LSB, the total distance will get 1.

For all number a1, a2, a3,..., a(n), if there are p numbers have 0 as LSB (put in set M), and q numbers have 1 for LSB (put in set N).

There are 2 situations:

Situation 1. If the 2 number in a pair both comes from M (or N), the total will get 0.

Situation 2. If the 1 number in a pair comes from M, the other comes from N, the total will get 1.

Since Situation 1 will add NOTHING to the total, we only need to think about Situation 2

How many pairs are there in Situation 2?

We choose 1 number from M (p possibilities), and 1 number from N (q possibilities).

The total possibilities is p × q = pq, which means

The total Hamming distance will get pq from LSB.

If we remove the LSB of all numbers (right logical shift), the same idea can be used again and again until all numbers becomes zero

479\_Largest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the largest palindrome made from the product of two n-digit numbers.

Since the result could be very large, you should return the largest palindrome mod 1337.

Example:

Input: 2

Output: 987

Explanation: 99 x 91 = 9009, 9009 % 1337 = 987

Note:

The range of n is [1,8].

The idea of this question is greedy:

find the max number that product of two number

try first largest palindrome, then second ...

verify if the palindrome can be formed by production of two i digit number.

repeat this until the palindrome has been found.

class Solution {

public int largestPalindrome(int n) {

if (n == 1){

return 9;

}

long maxNum = (long) Math.pow(10, n) - 1;

long minNum = (long) Math.pow(10, n-1);

long maxProd = maxNum \* maxNum;

long firstHalf = maxProd / (long) Math.pow(10, n);

while (firstHalf > minNum){

long pa = createPa(firstHalf);

for (long i = maxNum; i >= minNum && pa / i <= maxNum; i--){

if (pa % i == 0){

return (int) (pa % 1337);

}

}

firstHalf--;

}

return -1;

}

private long createPa(long num){

String s = num + new StringBuilder().append(num).reverse().toString();

return Long.parseLong(s);

}

}

48\_RotateImage.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Note:

You have to rotate the image in-place, which means you have to modify the input 2D matrix directly.

DO NOT allocate another 2D matrix and do the rotation.

Example 1:

Given input matrix =

[

[1,2,3],

[4,5,6],

[7,8,9]

],

rotate the input matrix in-place such that it becomes:

[

[7,4,1],

[8,5,2],

[9,6,3]

]

Example 2:

Given input matrix =

[

[ 5, 1, 9,11],

[ 2, 4, 8,10],

[13, 3, 6, 7],

[15,14,12,16]

],

rotate the input matrix in-place such that it becomes:

[

[15,13, 2, 5],

[14, 3, 4, 1],

[12, 6, 8, 9],

[16, 7,10,11]

]

Method 1:

calculate all the four coordiantes first

class Solution {

public void rotate(int[][] matrix) {

int n = matrix.length;

for (int i = 0; i < n / 2; i++){

for (int j = 0; j < (n+1) / 2; j++){

int temp = matrix[i][j];

matrix[i][j] = matrix[n-1-j][i];

matrix[n-1-j][i] = matrix[n-1-i][n-1-j];

matrix[n-1-i][n-1-j] = matrix[j][n-1-i];

matrix[j][n-1-i] = temp;

}

}

}

}

Method 2:

a. flip along x-axis

b. transpose

class Solution {

public void rotate(int[][] matrix) {

int n = matrix.length;

for (int i = 0; i < n / 2; i++){

for (int j = 0; j < n ; j++){

int temp = matrix[i][j];

matrix[i][j] = matrix[n-1-i][j];

matrix[n-1-i][j] = temp;

}

}

for (int i = 0; i < n; i++){

for (int j = i; j < n; j++){

int temp = matrix[i][j];

matrix[i][j] = matrix[j][i];

matrix[j][i] = temp;

}

}

}

}

480\_Sliding.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:

[2,3,4] , the median is 3

[2,3], the median is (2 + 3) / 2 = 2.5

Given an array nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Your job is to output the median array for each window in the original array.

For example,

Given nums = [1,3,-1,-3,5,3,6,7], and k = 3.

Window position Median

--------------- -----

[1 3 -1] -3 5 3 6 7 1

1 [3 -1 -3] 5 3 6 7 -1

1 3 [-1 -3 5] 3 6 7 -1

1 3 -1 [-3 5 3] 6 7 3

1 3 -1 -3 [5 3 6] 7 5

1 3 -1 -3 5 [3 6 7] 6

Therefore, return the median sliding window as [1,-1,-1,3,5,6].

The similar as 295. Find Median from Data Stream

Use two PQ

Time complexity: O(nk) as add takes O(logk) but remove takes O(k)

Space complexity: O(n)

class Solution {

public double[] medianSlidingWindow(int[] nums, int k) {

Queue<Integer> minPQ = new PriorityQueue<>();//store the larger half, if even number, put into minPQ

Queue<Integer> maxPQ = new PriorityQueue<>(new Comparator<Integer>(){//store the smaller half

public int compare (Integer i1, Integer i2){

return i2.compareTo(i1);

}

});

int n = nums.length;

if (n == 0){

return new double[0];

}

double[] res = new double[n-k+1];

for (int i = 0; i < n; i++){

add(minPQ, maxPQ, nums[i]);

if (i >= k - 1){

res[i-k+1] = getMedian(minPQ, maxPQ);

remove(minPQ, maxPQ, nums[i-k+1]);

}

}

return res;

}

private void add(Queue<Integer> minPQ, Queue<Integer> maxPQ, int num){

if (num < getMedian(minPQ, maxPQ)){

maxPQ.offer(num);

}else{

minPQ.offer(num);

}

if (maxPQ.size() > minPQ.size()){

minPQ.offer(maxPQ.poll());

}else if (minPQ.size() - maxPQ.size() > 1){

maxPQ.offer(minPQ.poll());

}

}

private void remove (Queue<Integer> minPQ, Queue<Integer> maxPQ, int num){

if (num < getMedian(minPQ, maxPQ)){

maxPQ.remove(num);

}else{

minPQ.remove(num);

}

if (maxPQ.size() > minPQ.size()){

minPQ.offer(maxPQ.poll());

}else if (minPQ.size() - maxPQ.size() > 1){

maxPQ.offer(minPQ.poll());

}

}

private double getMedian(Queue<Integer> minPQ, Queue<Integer> maxPQ){

if (minPQ.size() == 0 && maxPQ.size() == 0){

return 0.0;

}

if (minPQ.size() == maxPQ.size()){

return ((double)minPQ.peek() + (double)maxPQ.peek()) / 2.0;

}else{

return (double) minPQ.peek();

}

}

}

class Solution {

public double[] medianSlidingWindow(int[] nums, int k) {

Queue<Integer> minPQ = new PriorityQueue<>();//store the larger half, if even number, put into minPQ

Queue<Integer> maxPQ = new PriorityQueue<>(new Comparator<Integer>(){//store the smaller half

public int compare (Integer i1, Integer i2){

return i2.compareTo(i1);

}

});

int n = nums.length;

if (n == 0){

return new double[0];

}

double[] res = new double[n-k+1];

for (int i = 0; i < n; i++){

add(minPQ, maxPQ, nums[i]);

if (i >= k - 1){

res[i-k+1] = getMedian(minPQ, maxPQ);

remove(minPQ, maxPQ, nums[i-k+1]);

}

}

return res;

}

private void add(Queue<Integer> minPQ, Queue<Integer> maxPQ, int num){

if (minPQ.isEmpty()){

minPQ.offer(num);

}else{

if (num < minPQ.peek()){

maxPQ.offer(num);

if (maxPQ.size() > minPQ.size()){

minPQ.offer(maxPQ.poll());

}

}else{

minPQ.offer(num);

if(minPQ.size() > maxPQ.size() + 1){

maxPQ.offer(minPQ.poll());

}

}

}

}

private void remove(Queue<Integer> minPQ, Queue<Integer> maxPQ, int num){

if (num < minPQ.peek()){

maxPQ.remove(num);

}else{

minPQ.remove(num);

}

if (maxPQ.size() > minPQ.size()){

minPQ.offer(maxPQ.poll());

}else if (minPQ.size() > maxPQ.size() + 1){

maxPQ.offer(minPQ.poll());

}

}

private double getMedian(Queue<Integer> minPQ, Queue<Integer> maxPQ){

if (minPQ.size() == 0 && maxPQ.size() == 0){

return 0.0;

}

if (minPQ.size() == maxPQ.size()){

return ((double)minPQ.peek() + (double)maxPQ.peek()) / 2.0;

}else{

return (double) minPQ.peek();

}

}

}

482\_License.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a license key represented as a string S which consists only alphanumeric character and dashes. The string is separated into N+1 groups by N dashes.

Given a number K, we would want to reformat the strings such that each group contains exactly K characters, except for the first group which could be shorter than K, but still must contain at least one character. Furthermore, there must be a dash inserted between two groups and all lowercase letters should be converted to uppercase.

Given a non-empty string S and a number K, format the string according to the rules described above.

Example 1:

Input: S = "5F3Z-2e-9-w", K = 4

Output: "5F3Z-2E9W"

Explanation: The string S has been split into two parts, each part has 4 characters.

Note that the two extra dashes are not needed and can be removed.

Example 2:

Input: S = "2-5g-3-J", K = 2

Output: "2-5G-3J"

Explanation: The string S has been split into three parts, each part has 2 characters except the first part as it could be shorter as mentioned above.

Note:

The length of string S will not exceed 12,000, and K is a positive integer.

String S consists only of alphanumerical characters (a-z and/or A-Z and/or 0-9) and dashes(-).

String S is non-empty.

class Solution {

public String licenseKeyFormatting(String S, int K) {

StringBuilder sb = new StringBuilder();

int count = 0;

for (int i = S.length() - 1; i >= 0; i--){

if (S.charAt(i) != '-'){

sb.append(S.charAt(i));

count++;

if (count == K){

sb.append('-');

count = 0;

}

}

}

if (sb.length() > 0 && sb.charAt(sb.length() - 1) == '-'){

sb.deleteCharAt(sb.length() - 1);

}

return sb.reverse().toString().toUpperCase();

}

}

class Solution {

public String licenseKeyFormatting(String S, int K) {

StringBuilder sb = new StringBuilder();

int count = 0;

for (int i = S.length() - 1; i >= 0; i--){

char c = S.charAt(i);

if (c != '-'){

if (count == K){

sb.append('-');

count = 0;

}

sb.append(Character.toUpperCase(c));

count++;

}

}

return sb.reverse().toString();

}

}

Smarter way:

public String licenseKeyFormatting(String s, int k) {

StringBuilder sb = new StringBuilder();

for (int i = s.length() - 1; i >= 0; i--)

if (s.charAt(i) != '-')

sb.append(sb.length() % (k + 1) == k ? '-' : "").append(s.charAt(i));

return sb.reverse().toString().toUpperCase();

}

484\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

By now, you are given a secret signature consisting of character 'D' and 'I'. 'D' represents a decreasing relationship between

two numbers, 'I' represents an increasing relationship between two numbers. And our secret signature was constructed by a special

integer array, which contains uniquely all the different number from 1 to n (n is the length of the secret signature plus 1).

For example, the secret signature "DI" can be constructed by array [2,1,3] or [3,1,2], but won't be constructed by array [3,2,4]

or [2,1,3,4], which are both illegal constructing special string that can't represent the "DI" secret signature.

On the other hand, now your job is to find the lexicographically smallest permutation of [1, 2, ... n] could refer to

the given secret signature in the input.

Example 1:

Input: "I"

Output: [1,2]

Explanation: [1,2] is the only legal initial spectial string can construct secret signature "I", where the number 1 and 2

construct an increasing relationship.

Example 2:

Input: "DI"

Output: [2,1,3]

Explanation: Both [2,1,3] and [3,1,2] can construct the secret signature "DI",

but since we want to find the one with the smallest lexicographical permutation, you need to output [2,1,3]

Note:

The input string will only contain the character 'D' and 'I'.

The length of input string is a positive integer and will not exceed 10,000

class Solution {

public int[] findPermutation(String s) {

int[] ans = new int[s.length() + 1];

for (int i = 0; i <= s.length(); i++){

ans[i] = i + 1;

}

for (int i = 0; i < s.length() ; i++){

if (s.charAt(i) == 'D'){

int start = i;

while (i < s.length() && s.charAt(i) == 'D'){

i++;

}

reverse(ans, start, i);

i--;

}

}

return ans;

}

private void reverse(int[] ans, int i, int j){

while (i < j){

int temp = ans[i];

ans[i] = ans[j];

ans[j] = temp;

i++;

j--;

}

}

}

485\_Max.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary array, find the maximum number of consecutive 1s in this array.

Example 1:

Input: [1,1,0,1,1,1]

Output: 3

Explanation: The first two digits or the last three digits are consecutive 1s.

The maximum number of consecutive 1s is 3.

Note:

The input array will only contain 0 and 1.

The length of input array is a positive integer and will not exceed 10,000

class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

int max = 0;

int start = 0;

int i = 0;

while (i < nums.length){

while (i < nums.length && nums[i] == 0){

i++;

}

start = i;

while (i < nums.length && nums[i] == 1){

i++;

}

max = Math.max(max, i - start);

}

return max;

}

}

class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

int max = 0;

int count = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

return max;

}

}

Best solution derived from max consecutive ones II

Sliding window

class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

int max = 0;

int count = 0;

int low = 0;

int high = 0;

int k = 0;//flip most number

while (high < nums.length){

if (nums[high] == 0){

count++;

}

high++;

while (count > k){

if (nums[low] == 0){

count--;

}

low++;

}

max = Math.max(max, high - low);

}

return max;

}

}

486\_Predict.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of scores that are non-negative integers. Player 1 picks one of the numbers from either end of the array followed by

the player 2 and then player 1 and so on. Each time a player picks a number, that number will not be available for the next player.

This continues until all the scores have been chosen. The player with the maximum score wins.

Given an array of scores, predict whether player 1 is the winner. You can assume each player plays to maximize his score.

Example 1:

Input: [1, 5, 2]

Output: False

Explanation: Initially, player 1 can choose between 1 and 2.

If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 chooses 5, then player 1 will be left with 1 (or 2).

So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.

Hence, player 1 will never be the winner and you need to return False.

Example 2:

Input: [1, 5, 233, 7]

Output: True

Explanation: Player 1 first chooses 1. Then player 2 have to choose between 5 and 7. No matter which number player 2 choose, player

1 can choose 233.

Finally, player 1 has more score (234) than player 2 (12), so you need to return True representing player1 can win.

Note:

1 <= length of the array <= 20.

Any scores in the given array are non-negative integers and will not exceed 10,000,000.

If the scores of both players are equal, then player 1 is still the winner.

Compare to 464, this one has to use dp (recursion + memo) because number is chosen following some order

https://leetcode.com/problems/predict-the-winner/discuss/96838/Java-'1-Line'-Recursive-Solution-O(n2)-Time-and-O(n)-Space

class Solution {

public boolean PredictTheWinner(int[] nums) {

int n = nums.length;

int[][] cache = new int[n][n];

for (int i = 0; i < n; i++){

Arrays.fill(cache[i], -1);

}

return maxRelativeScore(nums, 0, n-1, cache) >= 0;

}

private int maxRelativeScore(int[] nums, int start, int end, int[][] cache){

if (cache[start][end] != -1){

return cache[start][end];

}

if (start == end){

return nums[start];

}

int chooseFront = nums[start] - maxRelativeScore(nums, start+1, end, cache);

int chooseEnd = nums[end] - maxRelativeScore(nums, start, end-1, cache);

cache[start][end] = Math.max(chooseFront, chooseEnd);

return cache[start][end];

}

}

Method 3: best of best

Time complexity: O(n^2)

Space complexity: O(n^2)

class Solution {

public boolean PredictTheWinner(int[] nums) {

int n = nums.length;

int[][] dp = new int[n][n];

for (int i = n - 2; i >= 0; i--){

for (int j = i + 1; j < n; j++){

dp[i][j] = Math.max(nums[i] - dp[i+1][j], nums[j] - dp[i][j-1]);

}

}

return dp[0][n-1] >= 0;

}

}

Method 1:

Time complexity: O(2^n)

Space complexity: O(n)

public class Solution {

public boolean PredictTheWinner(int[] nums) {

return winner(nums, 0, nums.length - 1, 1) >= 0;

}

public int winner(int[] nums, int s, int e, int turn) {

if (s == e)

return turn \* nums[s];

int a = turn \* nums[s] + winner(nums, s + 1, e, -turn);

int b = turn \* nums[e] + winner(nums, s, e - 1, -turn);

return turn \* Math.max(turn \* a, turn \* b);

}

}

Method 2:

Time complexity: O(n^2)

Space complexity: O(n^2)

public class Solution {

public boolean PredictTheWinner(int[] nums) {

Integer[][] memo = new Integer[nums.length][nums.length];

return winner(nums, 0, nums.length - 1, memo) >= 0;

}

public int winner(int[] nums, int s, int e, Integer[][] memo) {

if (s == e)

return nums[s];

if (memo[s][e] != null)

return memo[s][e];

int a = nums[s] - winner(nums, s + 1, e, memo);

int b = nums[e] - winner(nums, s, e - 1, memo);

memo[s][e] = Math.max(a, b);

return memo[s][e];

}

}

Method 4:

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public boolean PredictTheWinner(int[] nums) {

int n = nums.length;

int[] dp = new int[n];

for (int i = n - 2; i >= 0; i--){

for (int j = i + 1; j < n; j++){

dp[j] = Math.max(nums[i] - dp[j], nums[j] - dp[j-1]);

}

}

return dp[n-1] >= 0;

}

}

The idea behind the recursive approach is simple. The two players Player 1 and Player 2 will be taking turns alternately.

For the Player 1 to be the winner, we need scorePlayer\_1≥scorePlayer\_2. Or in other terms, scorePlayer\_1−scorePlayer\_2≥0

.

Thus, for the turn of Player 1, we can add its score obtained to the total score and for Player 2's turn, we can substract its

score from the total score. At the end, we can check if the total score is greater than or equal to zero(equal score of both players),

to predict that Player 1 will be the winner.

Thus, by making use of a recursive function winner(nums,s,e,turn) which predicts the winner for the numsnumsnums array as the

score array with the elements in the range of indices [s,e][s,e][s,e] currently being considered, given a particular player's turn,

indicated by turn=1turn=1turn=1 being Player 1's turn and turn=−1turn=-1turn=−1 being the Player 2's turn, we can predict the winner of

the given problem by making the function call winner(nums,0,n-1,1). Here, nnn refers to the length of numsnumsnums array.

In every turn, we can either pick up the first(nums[s]nums[s]nums[s]) or the last(nums[e]nums[e]nums[e]) element of the current

subarray. Since both the players are assumed to be playing smartly and making the best move at every step, both will tend to maximize

their scores. Thus, we can make use of the same function winner to determine the maximum score possible for any of the players.

Now, at every step of the recursive process, we determine the maximum score possible for the current player. It will be the maximum

one possible out of the scores obtained by picking the first or the last element of the current subarray.

To obtain the score possible from the remaining subarray, we can again make use of the same winner function and add the score

corresponding to the point picked in the current function call. But, we need to take care of whether to add or subtract this score

to the total score available. If it is Player 1's turn, we add the current number's score to the total score, otherwise, we need to

subtract the same.

Thus, at every step, we need update the search space appropriately based on the element chosen and also invert the turnturnturn's

value to indicate the turn change among the players and either add or subtract the current player's score from the total score available to determine the end result.

Further, note that the value returned at every step is given by turn∗max(turn∗a,turn∗b)turn \*\text{max}(turn \* a, turn \* b)

turn∗max(turn∗a,turn∗b). This is equivalent to the statement max(a,b)max(a,b)max(a,b) for Player 1's turn and min(a,b)min(a,b)min(a,b)

for Player 2's turn.

This is done because, looking from Player 1's perspective, for any move made by Player 1, it tends to leave the remaining subarray

in a situation which minimizes the best score possible for Player 2, even if it plays in the best possible manner. But, when the turn

passes to Player 1 again, for Player 1 to win, the remaining subarray should be left in a state such that the score obtained from this

subarrray is maximum(for Player 1).

This is a general criteria for any arbitrary two player game and is commonly known as the Min-Max algorithm.

The following image shows how the scores are passed to determine the end result for a simple example.

487\_Max.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary array, find the maximum number of consecutive 1s in this array if you can flip at most one 0.

Example 1:

Input: [1,0,1,1,0]

Output: 4

Explanation: Flip the first zero will get the the maximum number of consecutive 1s.

After flipping, the maximum number of consecutive 1s is 4.

Note:

The input array will only contain 0 and 1.

The length of input array is a positive integer and will not exceed 10,000

Follow up:

What if the input numbers come in one by one as an infinite stream? In other words, you can't store all numbers coming from the stream as it's too large to hold in memory. Could you solve it efficiently?

Method 1:

Use variable index to record the position of zero, and initiate the index to -1. Once we find a zero, refresh the count and store the new index.

class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

int max = 0;

int count = 0;

int index = -1;

for (int i = 0; i < nums.length; i++){

if (nums[i] == 1){

count++;

}else{

count = i - index;

index = i;

}

max = Math.max(max, count);

}

return max;

}

}

Method 2 (better solution): sliding window to cover follow up

keep a window [l, h] that contains at most k zero

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

int max = 0;

int zeros = 0;

int k = 1;//flip most

int low = 0;

int high = 0;

while (high < nums.length){

if (nums[high] == 0){

zeros++;

}

high++;

while (zeros > k){

if (nums[low] == 0){

zeros--;

}

low++;

}

max = Math.max(max, high - low);

}

return max;

}

}

Now let's deal with follow-up, we need to store up to k indexes of zero within the window [l, h] so that we know where to move l next when the window contains more than k zero. If the input stream is infinite, then the output could be extremely large because there could be super long consecutive ones. In that case we can use BigInteger for all indexes. For simplicity, here we will use int

Time: O(n) Space: O(k)

Follow up

public int findMaxConsecutiveOnes(int[] nums) {

int max = 0, k = 1; // flip at most k zero

Queue<Integer> zeroIndex = new LinkedList<>();

for (int l = 0, h = 0; h < nums.length; h++) {

if (nums[h] == 0)

zeroIndex.offer(h);

if (zeroIndex.size() > k)

l = zeroIndex.poll() + 1;

max = Math.max(max, h - l + 1);

}

return max;

}

Note that setting k = 0 will give a solution to the earlier version Max Consecutive Ones

For k = 1 we can apply the same idea to simplify the solution. Here q stores the index of zero within the window [l, h] so its role is similar to Queue in the above solution

public int findMaxConsecutiveOnes(int[] nums) {

int max = 0, q = -1;

for (int l = 0, h = 0; h < nums.length; h++) {

if (nums[h] == 0) {

l = q + 1;

q = h;

}

max = Math.max(max, h - l + 1);

}

return max;

}

489\_Robot.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a robot cleaner in a room modeled as a grid.

Each cell in the grid can be empty or blocked.

The robot cleaner with 4 given APIs can move forward, turn left or turn right. Each turn it made is 90 degrees.

When it tries to move into a blocked cell, its bumper sensor detects the obstacle and it stays on the current cell.

Design an algorithm to clean the entire room using only the 4 given APIs shown below.

interface Robot {

// returns true if next cell is open and robot moves into the cell.

// returns false if next cell is obstacle and robot stays on the current cell.

boolean move();

// Robot will stay on the same cell after calling turnLeft/turnRight.

// Each turn will be 90 degrees.

void turnLeft();

void turnRight();

// Clean the current cell.

void clean();

}

Example:

Input:

room = [

[1,1,1,1,1,0,1,1],

[1,1,1,1,1,0,1,1],

[1,0,1,1,1,1,1,1],

[0,0,0,1,0,0,0,0],

[1,1,1,1,1,1,1,1]

],

row = 1,

col = 3

Explanation:

All grids in the room are marked by either 0 or 1.

0 means the cell is blocked, while 1 means the cell is accessible.

The robot initially starts at the position of row=1, col=3.

From the top left corner, its position is one row below and three columns right.

Notes:

The input is only given to initialize the room and the robot's position internally. You must solve this problem "blindfolded".

In other words, you must control the robot using only the mentioned 4 APIs, without knowing the room layout and the initial

robot's position.

The robot's initial position will always be in an accessible cell.

The initial direction of the robot will be facing up.

All accessible cells are connected, which means the all cells marked as 1 will be accessible by the robot.

Assume all four edges of the grid are all surrounded by wall.

/\*\*

\* // This is the robot's control interface.

\* // You should not implement it, or speculate about its implementation

\* interface Robot {

\* // Returns true if the cell in front is open and robot moves into the cell.

\* // Returns false if the cell in front is blocked and robot stays in the current cell.

\* public boolean move();

\*

\* // Robot will stay in the same cell after calling turnLeft/turnRight.

\* // Each turn will be 90 degrees.

\* public void turnLeft();

\* public void turnRight();

\*

\* // Clean the current cell.

\* public void clean();

\* }

\*/

class Solution {

public void cleanRoom(Robot robot) {

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Set<String> visited = new HashSet<>();

int[] pos = new int[]{0, 0};

backtracking(robot, visited, dirs, pos, 0);

}

private void backtracking(Robot robot, Set<String> visited, int[][] dirs, int[] pos, int direct){

String position = pos[0] + "," + pos[1];

if (visited.contains(position)){

return;

}

visited.add(position);

robot.clean();

for (int i = 0; i < dirs.length; i++){

int newDirect = (direct + i) % 4;

if (robot.move()){

int x = pos[0] + dirs[newDirect][0];

int y = pos[1] + dirs[newDirect][1];

int[] newPos = new int[]{x, y};

backtracking(robot, visited, dirs, newPos, newDirect);

robot.turnRight();

robot.turnRight();

robot.move();

robot.turnLeft();

robot.turnLeft();

}

robot.turnRight();

}

}

}

49\_GroupAnagrams.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of strings, group anagrams together.

For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],

Return:

[

["ate", "eat","tea"],

["nat","tan"],

["bat"]

]

Note: All inputs will be in lower-case.

Time complexity: O(N\*MlogM)

Space complexity: O(n + m)

class Solution {

public List<List<String>> groupAnagrams(String[] strs) {

List<List<String>> result = new ArrayList<>();

Map<String, List<String>> map = new HashMap<>();

for (int i = 0; i < strs.length; i++){

char[] sc = strs[i].toCharArray();

Arrays.sort(sc);

String s = String.valueOf(sc);

if (!map.containsKey(s)){

map.put(s, new ArrayList<>());

}

map.get(s).add(strs[i]);

}

for (Map.Entry<String, List<String>> entry : map.entrySet()){

result.add(entry.getValue());

}

return result;

}

}

class Solution {

public List<List<String>> groupAnagrams(String[] strs) {

List<List<String>> res = new ArrayList<>();

Map<String, List<String>> map = new HashMap<>();

for (String str : strs){

char[] cs = str.toCharArray();

Arrays.sort(cs);

String key = new String(cs);

if (!map.containsKey(key)){

map.put(key, new ArrayList<>());

}

map.get(key).add(str);

}

for (List<String> list : map.values()){

res.add(list);

}

return res;

}

}

49\_SortLettersbyCase.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string which contains only letters. Sort it by lower case first and upper case second.

Notice

It's NOT necessary to keep the original order of lower-case letters and upper case letters.

Have you met this question in a real interview? Yes

Example

For "abAcD", a reasonable answer is "acbAD"

public class Solution {

/\*

\* @param chars: The letter array you should sort by Case

\* @return: nothing

\*/

public void sortLetters(char[] chars) {

if (chars == null || chars.length == 0){

return;

}

int start = 0;

int end = chars.length - 1;

while (start < end){

while (start < end && Character.isLowerCase(chars[start])){

start++;

}

while (start < end && Character.isUpperCase(chars[end])){

end--;

}

if (start < end){

char temp = chars[start];

chars[start] = chars[end];

chars[end] = temp;

start++;

end--;

}

}

}

}

490\_maze.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right,

but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's start position, the destination and the maze, determine whether the ball could stop at the destination.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of

the maze are all walls. The start and destination coordinates are represented by row and column indexes.

Example 1

Input 1: a maze represented by a 2D array

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

0 0 0 0 0

Input 2: start coordinate (rowStart, colStart) = (0, 4)

Input 3: destination coordinate (rowDest, colDest) = (4, 4)

Output: true

Explanation: One possible way is : left -> down -> left -> down -> right -> down -> right.

Example 2

Input 1: a maze represented by a 2D array

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

0 0 0 0 0

Input 2: start coordinate (rowStart, colStart) = (0, 4)

Input 3: destination coordinate (rowDest, colDest) = (3, 2)

Output: false

Explanation: There is no way for the ball to stop at the destination.

Note:

There is only one ball and one destination in the maze.

Both the ball and the destination exist on an empty space, and they will not be at the same position initially.

The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.

The maze contains at least 2 empty spaces, and both the width and height of the maze won't exceed 100.

Time complexity : O(mn) Complete traversal of maze will be done in the worst case.

Space complexity : O(mn). visited array of size m∗n is used and queue size can grow upto m∗n in worst case.

Method 1: BFS

class Solution {

public boolean hasPath(int[][] maze, int[] start, int[] destination) {

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

int m = maze.length;

int n = maze[0].length;

boolean[][] visited = new boolean[m][n];

qx.offer(start[0]);

qy.offer(start[1]);

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

if (cx == destination[0] && cy == destination[1]){

return true;

}

for (int i = 0; i < dx.length; i++){

int x = cx + dx[i];

int y = cy + dy[i];

while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0){

x += dx[i];

y += dy[i];

}

if (!visited[x-dx[i]][y-dy[i]]){

qx.offer(x-dx[i]);

qy.offer(y-dy[i]);

visited[x-dx[i]][y-dy[i]] = true;

}

}

}

return false;

}

}

One queue: Best solution

class Solution {

public boolean hasPath(int[][] maze, int[] start, int[] destination) {

int[][] dir = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

int m = maze.length;

int n = maze[0].length;

boolean[][] visited = new boolean[m][n];

queue.offer(start);

while (!queue.isEmpty()){

int[] p = queue.poll();

if (p[0] == destination[0] && p[1] == destination[1]){

return true;

}

for (int i = 0; i < dir.length; i++){

int x = p[0] + dir[i][0];

int y = p[1] + dir[i][1];

while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0){

x += dir[i][0];

y += dir[i][1];

}

if (!visited[x-dir[i][0]][y-dir[i][1]]){

queue.offer(new int[]{x-dir[i][0], y-dir[i][1]});

visited[x-dir[i][0]][y-dir[i][1]] = true;

}

}

}

return false;

}

}

Method 2: DFS

class Solution {

public boolean hasPath(int[][] maze, int[] start, int[] destination) {

boolean[][] visited = new boolean[maze.length][maze[0].length];

visited[start[0]][start[1]] = true;

return dfs(maze, start, destination, visited);

}

private boolean dfs(int[][] maze, int[] start, int[] destination, boolean[][] visited){

if (start[0] == destination[0] && start[1] == destination[1]){

return true;

}

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

int m = maze.length;

int n = maze[0].length;

for (int i = 0; i < dx.length; i++){

int x = start[0] + dx[i];

int y = start[1] + dy[i];

while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0){

x += dx[i];

y += dy[i];

}

if (!visited[x-dx[i]][y-dy[i]]){

visited[x-dx[i]][y-dy[i]] = true;

if (dfs(maze, new int[]{x-dx[i], y-dy[i]}, destination, visited)){

return true;

}

visited[x-dx[i]][y-dy[i]] = false;

}

}

return false;

}

}

public class Solution {

public boolean hasPath(int[][] maze, int[] start, int[] destination) {

boolean[][] visited = new boolean[maze.length][maze[0].length];

return dfs(maze, start, destination, visited);

}

public boolean dfs(int[][] maze, int[] start, int[] destination, boolean[][] visited) {

if (visited[start[0]][start[1]])

return false;

if (start[0] == destination[0] && start[1] == destination[1])

return true;

visited[start[0]][start[1]] = true;

int r = start[1] + 1, l = start[1] - 1, u = start[0] - 1, d = start[0] + 1;

while (r < maze[0].length && maze[start[0]][r] == 0) // right

r++;

if (dfs(maze, new int[] {start[0], r - 1}, destination, visited))

return true;

while (l >= 0 && maze[start[0]][l] == 0) //left

l--;

if (dfs(maze, new int[] {start[0], l + 1}, destination, visited))

return true;

while (u >= 0 && maze[u][start[1]] == 0) //up

u--;

if (dfs(maze, new int[] {u + 1, start[1]}, destination, visited))

return true;

while (d < maze.length && maze[d][start[1]] == 0) //down

d++;

if (dfs(maze, new int[] {d - 1, start[1]}, destination, visited))

return true;

return false;

}

}

DFS:

In order to do this traversal, one of the simplest schemes is to undergo depth first search. In this case, we choose one

path at a time and try to go as deep as possible into the levels of the tree before going for the next path. In order to

implement this, we make use of a recursive function dfs(maze, start, desination, visited). This function takes the given

mazemazemaze array, the startstartstart position and the destinationdestinationdestination position as its arguments along

with a visitedvisitedvisited array. visitedvisitedvisited array is a 2-D boolean array of the same size as that of mazemazemaze.

A True value at visited[i][j]visited[i][j]visited[i][j] represents that the current position has already been reached earlier

during the path traversal. We make use of this array so as to keep track of the same paths being repeated over and over. We

mark a True at the current position in the visitedvisitedvisited array once we reach that particular positon in the mazemazemaze.

From every startstartstart position, we can move continuously in either left, right, upward or downward direction till we reach

the boundary or a wall. Thus, from the startstartstart position, we determine all the end points which can be reached by choosing

the four directions. For each of the cases, the new endpoint will now act as the new start point for the traversals. The destination,

obviously remains unchanged. Thus, now we call the same function four times for the four directions, each time with a new start point

obtained previously.

If any of the function call returns a True value, it means we can reach the desination.

491\_Increasing.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array, your task is to find all the different possible increasing subsequences of the given array, and the length of

an increasing subsequence should be at least 2 .

Example:

Input: [4, 6, 7, 7]

Output: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7,7], [4,7,7]]

Note:

The length of the given array will not exceed 15.

The range of integer in the given array is [-100,100].

The given array may contain duplicates, and two equal integers should also be considered as a special case of increasing sequence.

Method 1: TLE

class Solution {

public List<List<Integer>> findSubsequences(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

dfs(res, new ArrayList<>(), nums, 0);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, int[] nums, int start){

if (item.size() >= 2 && !res.contains(item)){

res.add(new ArrayList<Integer>(item));

}

for (int i = start; i < nums.length; i++){

if (item.isEmpty() || item.get(item.size()-1) <= nums[i]){

item.add(nums[i]);

dfs(res, item, nums, i+1);

item.remove(item.size() - 1);

}

}

}

}

Method 2:

class Solution {

public List<List<Integer>> findSubsequences(int[] nums) {

Set<List<Integer>> res = new HashSet<List<Integer>>();

dfs(res, new ArrayList<>(), nums, 0);

List<List<Integer>> result = new ArrayList<>(res);

return result;

}

private void dfs(Set<List<Integer>> res, List<Integer> item, int[] nums, int start){

if (item.size() >= 2){

res.add(new ArrayList<Integer>(item));

}

for (int i = start; i < nums.length; i++){

if (item.isEmpty() || item.get(item.size()-1) <= nums[i]){

item.add(nums[i]);

dfs(res, item, nums, i+1);

item.remove(item.size() - 1);

}

}

}

}

Method 3: Best Solution note that this is not stardard backtracking

Pretty straightforward. Maybe one thing is: while nums is not necessarily sorted but we have to skip duplicates in each recursion,

so we use a hash set to record what we have used in this particular recursion.

We don't do the same as standard backtrack to set.remove() because we have the set at each level,

not the same set used by all levels in the DFS.

class Solution {

public List<List<Integer>> findSubsequences(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

dfs(res, new ArrayList<>(), nums, 0);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, int[] nums, int start){

if (item.size() >= 2){

res.add(new ArrayList<Integer>(item));

}

Set<Integer> set = new HashSet<>(); // note that this is not stardard backtracking

for (int i = start; i < nums.length; i++){

if (set.contains(nums[i])){

continue;

}

if (item.isEmpty() || item.get(item.size()-1) <= nums[i]){

set.add(nums[i]);

item.add(nums[i]);

dfs(res, item, nums, i+1);

item.remove(item.size() - 1);

}

}

}

}

492\_Construct.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For a web developer, it is very important to know how to design a web page's size. So, given a specific rectangular web page’s area, your job by now is to design a rectangular web page, whose length L and width W satisfy the following requirements:

1. The area of the rectangular web page you designed must equal to the given target area.

2. The width W should not be larger than the length L, which means L >= W.

3. The difference between length L and width W should be as small as possible.

You need to output the length L and the width W of the web page you designed in sequence.

Example:

Input: 4

Output: [2, 2]

Explanation: The target area is 4, and all the possible ways to construct it are [1,4], [2,2], [4,1].

But according to requirement 2, [1,4] is illegal; according to requirement 3, [4,1] is not optimal compared to [2,2]. So the length L is 2, and the width W is 2.

Note:

The given area won't exceed 10,000,000 and is a positive integer

The web page's width and length you designed must be positive integers.

class Solution {

public int[] constructRectangle(int area) {

int[] res = new int[2];

for (int i = (int) Math.sqrt(area); i <= area; i++){

if (i \* (area/ i) == area){

res[0] = Math.max(i, area/i);

res[1] = Math.min(i, area/i);

break;

}

}

return res;

}

}

4ms Best

class Solution {

public int[] constructRectangle(int area) {

int[] res = new int[2];

for (int i = (int) Math.sqrt(area); i > 0; i--){

if (area % i == 0){

res[0] = area / i;

res[1] = i;

break;

}

}

return res;

}

}

493\_Reverse.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array nums, we call (i, j) an important reverse pair if i < j and nums[i] > 2\*nums[j].

You need to return the number of important reverse pairs in the given array.

Example1:

Input: [1,3,2,3,1]

Output: 2

Example2:

Input: [2,4,3,5,1]

Output: 3

Note:

The length of the given array will not exceed 50,000.

All the numbers in the input array are in the range of 32-bit integer.

Merge sort idea

The similar as Count of Smaller number after self

class Solution {

int count = 0;

public int reversePairs(int[] nums) {

int[] temp = new int[nums.length];

mergeSort(nums, temp, 0, nums.length - 1);

return count;

}

private void mergeSort(int[] nums, int[] temp, int start, int end){

if (start >= end){

return;

}

int mid = (start + end) / 2;

mergeSort(nums, temp, start, mid);

mergeSort(nums, temp, mid + 1, end);

merge(nums, temp, start, mid, end);

}

private void merge(int[] nums, int[] temp, int start, int mid, int end){

int left = start;

int right = mid + 1;

int index = start;

//this is count which is the only part different from merge sort

int subCount = 0;

int l = left;

int r = right;

while (l <= mid){

if (r > end || (long) nums[l] <= 2 \* (long) nums[r]){

count += subCount;

l++;

}else{

subCount++;

r++;

}

}

///////////////////////

while (left <= mid && right <= end){

if (nums[left] < nums[right]){

temp[index++] = nums[left++];

}else {

temp[index++] = nums[right++];

}

}

while (left <= mid){

temp[index++] = nums[left++];

}

while (right <= end){

temp[index++] = nums[right++];

}

for (int i = start; i <= end; i++){

nums[i] = temp[i];

}

}

}

494\_Target.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols + and -. For each integer,

you should choose one from + and - as its new symbol.

Find out how many ways to assign symbols to make sum of integers equal to target S.

Example 1:

Input: nums is [1, 1, 1, 1, 1], S is 3.

Output: 5

Explanation:

-1+1+1+1+1 = 3

+1-1+1+1+1 = 3

+1+1-1+1+1 = 3

+1+1+1-1+1 = 3

+1+1+1+1-1 = 3

There are 5 ways to assign symbols to make the sum of nums be target 3.

Note:

The length of the given array is positive and will not exceed 20.

The sum of elements in the given array will not exceed 1000.

Your output answer is guaranteed to be fitted in a 32-bit integer.

https://leetcode.com/problems/target-sum/solution/

Method 1: dfs

Time complexity : O(2^n) Size of recursion tree will be 2^n2

Space complexity: O(n) n refers to the size of numsnums array.

Space complexity : O(n)O(n). The depth of the recursion tree can go upto nn.

class Solution {

int count = 0;

public int findTargetSumWays(int[] nums, int S) {

List<String> result = new ArrayList<>();

dfs(nums, 0, S, 0);

return count;

}

private void dfs(int[] nums, int start, int target, int eval){

if (start == nums.length){

if (eval == target){

count++;

}

return;

}

// for (int i = start; i < nums.length; i++){

dfs(nums, start + 1, target, eval + nums[start]);

dfs(nums, start + 1, target, eval - nums[start]);

// }

}

}

class Solution {

int count = 0;

public int findTargetSumWays(int[] nums, int S) {

List<String> res = new ArrayList<>();

dfs(nums, S, 0);

return count;

}

private void dfs(int[] nums, int target, int start){

if (start == nums.length){

if (target == 0){

count++;

}

return;

}

dfs(nums, target - nums[start], start+1);

dfs(nums, target + nums[start], start+1);

}

}

Similar as Leetcode 416 Partition Equal Subset Sum

https://leetcode.com/problems/target-sum/discuss/97334/Java-(15-ms)-C++-(3-ms)-O(ns)-iterative-DP-solution-using-subset-sum-with-explanation

Best solution: Reduce space complexity by scan from end to start

if scan from start to end, need additional dimension because end data relies on previous start data

Best solution:

class Solution {

public int findTargetSumWays(int[] nums, int S) {

int sum = 0;

for (int i : nums){

sum += i;

}

if (sum < S || (sum + S) %2 != 0){

return 0;

}

//below is the problem of Partition Equal Subset Sum

int target = (sum + S) / 2;

int n = nums.length;

int[] dp = new int[target+1];

dp[0] = 1;

for (int i = 0; i < n; i++){

for (int j = target; j >= nums[i]; j--){

dp[j] += dp[j - nums[i]];

}

}

return dp[target];

}

}

class Solution {

public int findTargetSumWays(int[] nums, int S) {

int sum = 0;

for (int i : nums){

sum += i;

}

if (sum < S || (sum + S) %2 != 0){

return 0;

}

//below is the problem of Partition Equal Subset Sum

int target = (sum + S) / 2;

int n = nums.length;

int[][] dp = new int[n+1][target+1];

for (int i = 0; i <= n; i++){

dp[i][0] = 1;

}

for (int i = 1; i <= n; i++){

for (int j = 0; j <= target; j++){ // here j must start from 0, not 1 for special case [0, 0, 0, 0, 1], S = 1

dp[i][j] = dp[i-1][j];

if (j >= nums[i-1]){

dp[i][j] += dp[i-1][j - nums[i-1]];

}

}

}

return dp[n][target];

}

}

496\_Next.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given two arrays (without duplicates) nums1 and nums2 where nums1’s elements are subset of nums2.

Find all the next greater numbers for nums1's elements in the corresponding places of nums2.

The Next Greater Number of a number x in nums1 is the first greater number to its right in nums2.

If it does not exist, output -1 for this number.

Example 1:

Input: nums1 = [4,1,2], nums2 = [1,3,4,2].

Output: [-1,3,-1]

Explanation:

For number 4 in the first array, you cannot find the next greater number for it in the second array, so output -1.

For number 1 in the first array, the next greater number for it in the second array is 3.

For number 2 in the first array, there is no next greater number for it in the second array, so output -1.

Example 2:

Input: nums1 = [2,4], nums2 = [1,2,3,4].

Output: [3,-1]

Explanation:

For number 2 in the first array, the next greater number for it in the second array is 3.

For number 4 in the first array, there is no next greater number for it in the second array, so output -1.

Note:

All elements in nums1 and nums2 are unique.

The length of both nums1 and nums2 would not exceed 1000.

Method 1: brute force

Time complexity: O(n^2)

Space complexity: O(1)

class Solution {

public int[] nextGreaterElement(int[] nums1, int[] nums2) {

int[] result = new int[nums1.length];

for (int i = 0; i < nums1.length; i++){

boolean foundKey = false;

boolean foundGreater = false;

for (int j = 0; j < nums2.length; j++){

if (!foundKey && nums2[j] == nums1[i]){

foundKey = true;

continue;

}

if (foundKey){

if (nums2[j] > nums1[i]){

result[i] = nums2[j];

foundGreater = true;

break;

}

}

}

if (!foundGreater){

result[i] = -1;

}

}

return result;

}

}

Method 2: monotonic increasing stack: top element is the smallest one

https://leetcode.com/problems/next-greater-element-i/discuss/97595/Java-10-lines-linear-time-complexity-O(n)-with-explanation

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int[] nextGreaterElement(int[] nums1, int[] nums2) {

int[] result = new int[nums1.length];

Map<Integer, Integer> map = new HashMap<>();

Stack<Integer> stack = new Stack<>();//monotonic stack

for (int num : nums2){

while(!stack.isEmpty() && stack.peek() < num){

map.put(stack.pop(), num);

}

stack.push(num);

}

for (int i = 0; i < nums1.length; i++){

result[i] = map.getOrDefault(nums1[i], -1);

}

return result;

}

}

498\_Diagonal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a matrix of M x N elements (M rows, N columns), return all elements of the matrix in diagonal order as shown in the below image.

Example:

Input:

[

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

]

Output: [1,2,4,7,5,3,6,8,9]

Explanation:

class Solution {

public int[] findDiagonalOrder(int[][] matrix) {

if (matrix == null || matrix.length == 0){

return new int[0];

}

int m = matrix.length;

int n = matrix[0].length;

int[] res = new int[m\*n];

Map<Integer, List<Integer>> map = new HashMap<>();

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (!map.containsKey(i+j)){

map.put(i+j, new ArrayList<Integer>());

}

map.get(i+j).add(matrix[i][j]);

}

}

int index = 0;

for (int key : map.keySet()){

List<Integer> temp = map.get(key);

int size = temp.size();

if (key % 2 == 0){

for (int i = size - 1; i >= 0; i--){

res[index++] = map.get(key).get(i);

}

}else{

for (int i = 0; i < size; i++){

res[index++] = map.get(key).get(i);

}

}

}

return res;

}

}

499\_MazeIII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up (u), down (d), left (l)

or right (r), but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction. There is

also a hole in this maze. The ball will drop into the hole if it rolls on to the hole.

Given the ball position, the hole position and the maze, find out how the ball could drop into the hole by moving the shortest distance.

The distance is defined by the number of empty spaces traveled by the ball from the start position (excluded) to the hole (included).

Output the moving directions by using 'u', 'd', 'l' and 'r'. Since there could be several different shortest ways, you should output

the lexicographically smallest way. If the ball cannot reach the hole, output "impossible".

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze

are all walls. The ball and the hole coordinates are represented by row and column indexes.

Example 1

Input 1: a maze represented by a 2D array

0 0 0 0 0

1 1 0 0 1

0 0 0 0 0

0 1 0 0 1

0 1 0 0 0

Input 2: ball coordinate (rowBall, colBall) = (4, 3)

Input 3: hole coordinate (rowHole, colHole) = (0, 1)

Output: "lul"

Explanation: There are two shortest ways for the ball to drop into the hole.

The first way is left -> up -> left, represented by "lul".

The second way is up -> left, represented by 'ul'.

Both ways have shortest distance 6, but the first way is lexicographically smaller because 'l' < 'u'. So the output is "lul".

Example 2

Input 1: a maze represented by a 2D array

0 0 0 0 0

1 1 0 0 1

0 0 0 0 0

0 1 0 0 1

0 1 0 0 0

Input 2: ball coordinate (rowBall, colBall) = (4, 3)

Input 3: hole coordinate (rowHole, colHole) = (3, 0)

Output: "impossible"

Explanation: The ball cannot reach the hole.

Note:

There is only one ball and one hole in the maze.

Both the ball and hole exist on an empty space, and they will not be at the same position initially.

The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.

The maze contains at least 2 empty spaces, and the width and the height of the maze won't exceed 30.

class Solution {

class Point{

int x;

int y;

int dist;

String s;

public Point(int x, int y, int dist, String s){

this.x = x;

this.y = y;

this.dist = dist;

this.s = s;

}

}

public String findShortestWay(int[][] maze, int[] ball, int[] hole) {

int m = maze.length;

int n = maze[0].length;

Point[][] points = new Point[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (i == ball[0] && j == ball[1]){

points[i][j] = new Point(i, j, 0, "");

}else{

points[i][j] = new Point(i, j, Integer.MAX\_VALUE, "");

}

}

}

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

String[] str = {"d", "r", "u", "l"};

Queue<Point> queue= new LinkedList<>();

queue.offer(new Point(ball[0], ball[1], 0, ""));

while (!queue.isEmpty()){

Point p = queue.poll();

for (int i = 0; i < dx.length; i++){

int x = p.x;

int y = p.y;

int count = 0;

String currStr = p.s + str[i];

while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0 && (x != hole[0] || y != hole[1])){

x += dx[i];

y += dy[i];

count++;

}

if (x != hole[0] || y != hole[1]){

x -= dx[i];

y -= dy[i];

count--;

}

if (p.dist + count < points[x][y].dist || p.dist + count == points[x][y].dist && currStr.compareTo(points[x][y].s) < 0){

points[x][y].dist = p.dist + count;

points[x][y].s = currStr;

queue.offer(new Point(x, y, p.dist + count, currStr));

}

}

}

return points[hole[0]][hole[1]].dist != Integer.MAX\_VALUE ? points[hole[0]][hole[1]].s : "impossible";

}

}

Best solution:

Class Solution{

Class Point{

int x;

int y;

int dist;

String str;

public Point (int x, int y, int dist, String str){

this.x = x;

this.y = y;

this.dist = dist;

this.str = str;

}

}

public String findShortestWay(int[][] maze, int[] ball, int[] hole){

int m = maze.length;

int n = maze[0].length;

Point[][] points = new Point[m][n];

Queue<Point> queue = new LinkedList<>();

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (i == ball[0] && j == ball[1]){

point[i][j] = new Point(i, j, 0, "");

queue.offer(point[i][j]);

}else{

point[i][j] = new Point(i, j, Integer.MAX\_VALUE, "");

}

}

}

String[] str = {"d", "r", "u", "l"};

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

while (!queue.isEmpty()){

Point p = queue.poll();

for (int i = 0; i< dirs.length; i++){

int[] dir = dirs[i];

int x = p.x;

int y = p.y;

int step = 0;

String currStr = p.str + str[i];

while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0 && (x != hole[0] || y != hole[1])){

x += dir[0];

y += dir[1];

step++;

}

if (x != hole[0] || y != hole[1]){

x -= dir[0];

y -= dir[1];

step--;

if (p.dist + step < point[x][y].dist || (p.dist + step == point[x][y] && currStr.compare(point[x][y].str) < 0)){

point[x][y].dist = p.dist + step;

point[x][y].str = currStr;

queue.offer(new Point(x, y, p.dist + step, currStr));

}

}else{

if (p.dist + step < point[x][y].dist || (p.dist + step == point[x][y] && currStr.compare(point[x][y].str) < 0)){

point[x][y].dist = p.dist + step;

point[x][y].str = currStr;

}

}

}

}

return points[hole[0]][hole[1]].dist != Integer.MAX\_VALUE ? points[hole[0]][hole[1]].s : "impossible";

}

}

5\_KthLargestElement.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ind K-th largest element in an array.

Notice

You can swap elements in the array

Have you met this question in a real interview? Yes

Example

In array [9,3,2,4,8], the 3rd largest element is 4.

In array [1,2,3,4,5], the 1st largest element is 5, 2nd largest element is 4, 3rd largest element is 3 and etc.

Challenge

O(n) time, O(1) extra memory.

https://leetcode.com/problems/kth-largest-element-in-an-array/discuss/60294/Solution-explained

Quick select: O(n) time Average, Worst Case O(N^2) without random, Random can improve to O(N)

class Solution {

/\*

\* @param k : description of k

\* @param nums : array of nums

\* @return: description of return

\*/

public int kthLargestElement(int k, int[] nums) {

int low = 0;

int high = nums.length - 1;

while (low <= high){

int index = partition(nums, low, high);

if (index == k - 1){

return nums[k - 1];

}else if (index > k - 1){

high = index - 1;

}else{

low = index + 1;

}

}

return -1;

}

private int partition(int[] A, int start, int end){

int head = start;

int pivot = A[start];

start++;

while (start <= end){

while (start <= end && A[start] > pivot){

start++;

}

while (start <= end && A[end] < pivot){

end--;

}

if (start <= end){

swap(A, start, end);

start++;

end--;

}

}

swap(A, head, end);

return end;

}

private void swap(int[] A, int a, int b){

int temp = A[a];

A[a] = A[b];

A[b] = temp;

}

};

//with random number

class Solution {

Random random = new Random();

public int findKthLargest(int[] nums, int k) {

int low = 0;

int high = nums.length - 1;

while (low <= high){

int index = partition(nums, low, high);

if (index == k - 1){

return nums[index];

}else if (index > k -1){

high = index - 1;

}else{

low = index + 1;

}

}

return -1;

}

private int partition(int[] nums, int low, int high){

int head = low;

int pivotIndex = low + random.nextInt(high - low + 1);

int pivot = nums[pivotIndex];

swap(nums, head, pivotIndex);

low++;

while (low <= high){

while (low <= high && nums[low] > pivot){

low++;

}

while (low <= high && nums[high] < pivot){

high--;

}

if (low <= high){

swap(nums, low, high);

low++;

high--;

}

}

swap(nums, head, high);

return high;

}

private void swap(int[] nums, int a, int b){

int temp = nums[a];

nums[a] = nums[b];

nums[b] = temp;

}

}

class Solution {

public int findKthLargest(int[] nums, int k) {

int low = 0;

int high = nums.length - 1;

while (low <= high){

int rank = partition(nums, low, high);

if (rank == k - 1){

return nums[rank];

}else if (rank < k - 1){

low = rank + 1;

}else{

high = rank - 1;

}

}

return -1;

}

private int partition(int[] nums, int low, int high){

int pIndex = low;

int pivot = nums[low];

low++;

while (low <= high){

if (nums[low] < pivot){

swap(nums, low, high);

high--;

}else{

low++;

}

}

swap(nums, pIndex, high);

return high;

}

private void swap(int[] nums, int i, int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

class Solution {

public int findKthLargest(int[] nums, int k) {

int start = 0;

int end = nums.length - 1;

while (start <= end){

int index = partition(nums, start, end);

if (index == k-1){

return nums[index];

}else if (index > k-1){

end = index-1;

}else{

start = index+1;

}

}

return nums[start];

}

private int partition(int[] nums, int start, int end){

Random random = new Random();

int index = start + random.nextInt(end - start + 1);

int pivot = nums[index];

swap(nums, index, start);

int head = start;

start++;

while (start <= end){

while (start <= end && nums[start] >= pivot){

start++;

}

while (start <= end && nums[end] < pivot){

end--;

}

if (start < end){

swap(nums, start, end);

start++;

end--;

}

}

swap(nums, head, end);

return end;

}

private void swap(int[] nums, int i, int j){

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

# 5\_LongestPalindromicSubstring.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s, find the longest palindromic substring in s. You may assume that the maximum length of s is 1000.

Example:

Input: "babad"

Output: "bab"

Note: "aba" is also a valid answer.

Example:

Input: "cbbd"

Output: "bb"

Best solution

class Solution {

public String longestPalindrome(String s) {

if (s == null || s.length() == 0){

return s;

}

String str = "";

for (int i = 0; i < s.length(); i++){

String odd = expand(s, i, i);

String even = expand(s, i, i+1);

if (str.length() < odd.length()){

str = odd;

}

if (str.length() < even.length()){

str = even;

}

}

return str;

}

private String expand(String s, int i, int j){

while (i >= 0 && j < s.length() && s.charAt(i) == s.charAt(j)){

i--;

j++;

}

return s.substring(i+1, j);

}

}

class Solution {

public String longestPalindrome(String s) {

if (s == null || s.length() == 0){

return "";

}

int[] res = new int[2];

for (int i = 0; i < s.length(); i++){

expand(s, i, i, res);

expand(s, i, i+1, res);

}

return s.substring(res[0], res[0] + res[1]);

}

private void expand(String s, int start, int end, int[] res){

while (start >= 0 && end < s.length() && s.charAt(start) == s.charAt(end)){

start--;

end++;

}

if (res[1] < end - start - 1){

res[1] = end - start - 1;

res[0] = start + 1;

}

}

}

Key idea, every time we move to right, we only need to consider whether using this new character as

tail could produce new palindrome string of length (current length +1) or (current length +2)

class Solution {

public String longestPalindrome(String s) {

String ans = "";

int currLen = 0;

for (int i = 0; i < s.length(); i++){

if (isPalindrome(s, i - currLen - 1, i)){

ans = s.substring(i - currLen - 1, i + 1);

currLen += 2;

}else if (isPalindrome(s, i - currLen, i)){

ans = s.substring(i - currLen, i + 1);

currLen++;

}

}

return ans;

}

private boolean isPalindrome(String s, int start, int end){

if (start < 0){

return false;

}

while (start < end){

if (s.charAt(start) == s.charAt(end)){

start++;

end--;

}else{

return false;

}

}

return true;

}

}

Method 3:

public class Solution {

private int lo, maxLen;

public String longestPalindrome(String s) {

int len = s.length();

for (int i = 0; i < len; i++) {

extendPalindrome(s, i, i); //assume odd length, try to extend Palindrome as possible

extendPalindrome(s, i, i+1); //assume even length.

}

return s.substring(lo, lo + maxLen);

}

private void extendPalindrome(String s, int j, int k) {

while (j >= 0 && k < s.length() && s.charAt(j) == s.charAt(k)) {

j--;

k++;

}

if (maxLen < k - j - 1) {

lo = j + 1;

maxLen = k - j - 1;

}

}

}

Method 4: O(n^2) DP Best Solution

dp(i, j) represents whether s(i ... j) can form a palindromic substring, dp(i, j) is true when s(i) equals to s(j) and

s(i+1 ... j-1) is a palindromic substring. When we found a palindrome, check if it's the longest one.

class Solution {

public String longestPalindrome(String s) {

int n = s.length();

boolean[][] dp = new boolean[n][n];

String res = "";

for (int i = n-1; i >= 0; i--){

for (int j = i; j < n; j++){

dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i <= 2 || dp[i+1][j-1]);

if (dp[i][j] && j - i + 1 > res.length()){

res = s.substring(i, j+1);

}

}

}

return res;

}

}

class Solution {

public String longestPalindrome(String s) {

int n = s.length();

boolean[][] dp = new boolean[n][n];

String res = "";

for (int i = 0; i < n; i++){

for (int j = 0; j <= i; j++){

dp[j][i] = (j+1 > i-1 || dp[j+1][i-1]) && s.charAt(j) == s.charAt(i);

if (dp[j][i] && res.length() < i - j + 1){

res = s.substring(j, i+1);

}

}

}

return res;

}

}

500\_Keyboard.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a List of words, return the words that can be typed using letters of alphabet on only one

row's of American keyboard like the image below.

class Solution {

public String[] findWords(String[] words) {

String[] strs = {"QWERTYUIOP","ASDFGHJKL","ZXCVBNM"};

Map<Character, Integer> map = new HashMap<>();

for (int i = 0 ; i < strs.length; i++){

for (int j = 0; j < strs[i].length(); j++){

map.put(strs[i].charAt(j), i);

}

}

List<String> list = new ArrayList<>();

for (String word : words){

int num = map.get(word.toUpperCase().charAt(0));

boolean can = true;

for (int i = 1; i < word.length(); i++){

if (map.get(word.toUpperCase().charAt(i)) != num){

can = false;

break;

}

}

if(can){

list.add(word);

}

}

String[] result = new String[list.size()];

for (int i = 0; i < list.size(); i++){

result[i] = list.get(i);

}

return result;

}

}

501\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary search tree (BST) with duplicates, find all the mode(s) (the most frequently

occurred element) in the given BST.

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than or equal to the node's key.

The right subtree of a node contains only nodes with keys greater than or equal to the node's key.

Both the left and right subtrees must also be binary search trees.

For example:

Given BST [1,null,2,2],

1

\

2

/

2

return [2].

Note: If a tree has more than one mode, you can return them in any order.

Follow up: Could you do that without using any extra space? (Assume that the implicit stack space

incurred due to recursion does not count).

Method 1:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int[] findMode(TreeNode root) {

Map<Integer, Integer> map = new HashMap<>();

helper(root, map);

List<Integer> list = new ArrayList<>();

int maxFreq = 0;

for (Integer i : map.keySet()){

maxFreq = Math.max(maxFreq, map.get(i));

}

for (Integer i : map.keySet()){

if (map.get(i) == maxFreq){

list.add(i);

}

}

int[] result = new int[list.size()];

for (int i = 0; i < result.length; i++){

result[i] = list.get(i);

}

return result;

}

private void helper(TreeNode root, Map<Integer, Integer> map){

if (root == null){

return;

}

helper(root.left, map);

map.put(root.val, map.getOrDefault(root.val, 0) + 1);

helper(root.right, map);

}

}

Method 2: use the concept that BST in order traversal is sorted

so the question is converted to find the mode for a sorted array without using extra space

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

int prevVal;

int count;

int max;

public int[] findMode(TreeNode root) {

List<Integer> list = new ArrayList<>();

helper(root, list);

int[] result = new int[list.size()];

for (int i = 0; i < result.length; i++){

result[i] = list.get(i);

}

return result;

}

private void helper(TreeNode root, List<Integer> list){

if (root == null){

return;

}

helper(root.left, list);

if (root.val == prevVal){

count++;

}else{

prevVal = root.val;

count = 1;

}

if (count > max){

max = count;

list.clear();

list.add(root.val);

}else if (count == max){

list.add(root.val);

}

helper(root.right, list);

}

}

503\_Next.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a circular array (the next element of the last element is the first element of the array),

print the Next Greater Number for every element. The Next Greater Number of a number x is the first

greater number to its traversing-order next in the array, which means you could search circularly to

find its next greater number. If it doesn't exist, output -1 for this number.

Example 1:

Input: [1,2,1]

Output: [2,-1,2]

Explanation: The first 1's next greater number is 2;

The number 2 can't find next greater number;

The second 1's next greater number needs to search circularly, which is also 2.

Note: The length of given array won't exceed 10000.

Method 1: similar as next grater element I, use stack, two different points

1. Use stack to store index as key instead of value due to duplicates allowed

2. Repeat array to implement cirular array

Time complexity: O(2\*n)

Space complexity: O(2\*n)

class Solution {

public int[] nextGreaterElements(int[] nums) {

int[] result = new int[nums.length];

Map<Integer, Integer> map = new HashMap<>();//store index and result

Stack<Integer> stack = new Stack<>();

for (int k = 0; k < 2; k++){

for (int i = 0; i < nums.length; i++){

while (!stack.isEmpty() && nums[stack.peek()] < nums[i]){

map.put(stack.pop(), nums[i]);

}

stack.push(i);

}

}

for (int i = 0; i < nums.length; i++){

result[i] = map.getOrDefault(i, -1);

}

return result;

}

}

Method 2: stack only, no map

class Solution {

public int[] nextGreaterElements(int[] nums) {

int[] result = new int[nums.length];

Arrays.fill(result, -1);

Stack<Integer> stack = new Stack<>();

for (int k = 0; k < 2; k++){

for (int i = 0; i < nums.length; i++){

while (!stack.isEmpty() && nums[stack.peek()] < nums[i]){

result[stack.pop()] = nums[i];

}

stack.push(i);

}

}

return result;

}

}

Method 3: use rolling array + stack

class Solution {

public int[] nextGreaterElements(int[] nums) {

int[] result = new int[nums.length];

Arrays.fill(result, -1);

Stack<Integer> stack = new Stack<>();

int n = nums.length;

for (int i = 0; i < 2 \* n; i++){

while (!stack.isEmpty() && nums[stack.peek()] < nums[i % n]){

result[stack.pop()] = nums[i % n];

}

stack.push(i % n);

}

return result;

}

}

504\_Base.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer, return its base 7 string representation.

Example 1:

Input: 100

Output: "202"

Example 2:

Input: -7

Output: "-10"

Note: The input will be in range of [-1e7, 1e7].

class Solution {

public String convertToBase7(int num) {

if (num == 0){

return String.valueOf(num);

}

int sign = 1;

if (num < 0){

sign = -1;

num = -num;

}

StringBuilder sb = new StringBuilder();

while (num != 0){

sb.append(num % 7);

num /= 7;

};

if (sign == -1){

sb.append('-');

}

return sb.reverse().toString();

}

}

505\_MazeII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right,

but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's start position, the destination and the maze, find the shortest distance for the ball to stop at the destination.

The distance is defined by the number of empty spaces traveled by the ball from the start position (excluded) to the destination

(included). If the ball cannot stop at the destination, return -1.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of

the maze are all walls. The start and destination coordinates are represented by row and column indexes.

Example 1

Input 1: a maze represented by a 2D array

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

0 0 0 0 0

Input 2: start coordinate (rowStart, colStart) = (0, 4)

Input 3: destination coordinate (rowDest, colDest) = (4, 4)

Output: 12

Explanation: One shortest way is : left -> down -> left -> down -> right -> down -> right.

The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.

Example 2

Input 1: a maze represented by a 2D array

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

0 0 0 0 0

Input 2: start coordinate (rowStart, colStart) = (0, 4)

Input 3: destination coordinate (rowDest, colDest) = (3, 2)

Output: -1

Explanation: There is no way for the ball to stop at the destination.

Note:

There is only one ball and one destination in the maze.

Both the ball and the destination exist on an empty space, and they will not be at the same position initially.

The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.

The maze contains at least 2 empty spaces, and both the width and height of the maze won't exceed 100.

Method 1: Better solution

BFS (accepted)

Time complexity : O(m∗n∗max(m,n)). Complete traversal of maze will be done in the worst case.

Further, for every current node chosen, we can travel upto a maximum depth of max(m,n) in any direction.

Space complexity : O(mn) queue size can grow upto m∗n in the worst case

class Solution {

public int shortestDistance(int[][] maze, int[] start, int[] destination) {

int m = maze.length;

int n = maze[0].length;

int[][] distance = new int[m][n];

for (int[] row : distance){

Arrays.fill(row, Integer.MAX\_VALUE);

}

distance[start[0]][start[1]] = 0;

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

qx.offer(start[0]);

qy.offer(start[1]);

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

for (int i = 0; i < dx.length; i++){

int x = cx + dx[i];

int y = cy + dy[i];

int dist = 0;

while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0){

x += dx[i];

y += dy[i];

dist++;

}

if (distance[cx][cy] + dist < distance[x-dx[i]][y-dy[i]]){

distance[x-dx[i]][y-dy[i]] = distance[cx][cy] + dist;

qx.offer(x-dx[i]);

qy.offer(y-dy[i]);

}

}

}

return distance[destination[0]][destination[1]] != Integer.MAX\_VALUE ? distance[destination[0]][destination[1]] : -1;

}

}

public class Solution {

public int shortestDistance(int[][] maze, int[] start, int[] dest) {

int[][] distance = new int[maze.length][maze[0].length];

for (int[] row: distance)

Arrays.fill(row, Integer.MAX\_VALUE);

distance[start[0]][start[1]] = 0;

int[][] dirs={{0, 1} ,{0, -1}, {-1, 0}, {1, 0}};

Queue <int[]> queue = new LinkedList<>();

queue.offer(start);

while (!queue.isEmpty()) {

int[] curr = queue.poll();

for (int[] dir: dirs) {

int x = curr[0] + dir[0];

int y = curr[1] + dir[1];

int step = 0;

while (x >= 0 && y >= 0 && x < maze.length && y < maze[0].length && maze[x][y] == 0) {

x += dir[0];

y += dir[1];

step++;

}

if (distance[curr[0]][curr[1]] + step < distance[x - dir[0]][y - dir[1]]) {

distance[x - dir[0]][y - dir[1]] = distance[curr[0]][curr[1]] + count;

queue.add(new int[] {x - dir[0], y - dir[1]});

}

}

}

return distance[dest[0]][dest[1]] == Integer.MAX\_VALUE ? -1 : distance[dest[0]][dest[1]];

}

}

Method 2:

backtracking (TLE)

class Solution {

int minDist = Integer.MAX\_VALUE;

public int shortestDistance(int[][] maze, int[] start, int[] destination) {

boolean[][] visited = new boolean[maze.length][maze[0].length];

visited[start[0]][start[1]] = true;

dfs(maze, start, destination, visited, 0);

return minDist != Integer.MAX\_VALUE ? minDist : -1;

}

private void dfs(int[][] maze, int[] start, int[] destination, boolean[][] visited, int prevDist){

if (start[0] == destination[0] && start[1] == destination[1]){

minDist = Math.min(minDist, prevDist);

return;

}

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

int m = maze.length;

int n = maze[0].length;

for (int i = 0; i < dx.length; i++){

int x = start[0] + dx[i];

int y = start[1] + dy[i];

int dist = prevDist;

while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0){

x += dx[i];

y += dy[i];

dist++;

}

if (!visited[x-dx[i]][y-dy[i]]){

visited[x-dx[i]][y-dy[i]] = true;

dfs(maze, new int[]{x-dx[i], y-dy[i]}, destination, visited, dist);

visited[x-dx[i]][y-dy[i]] = false;

}

}

}

}

DFS (accepted)

class Solution {

public int shortestDistance(int[][] maze, int[] start, int[] destination) {

int[][] distance = new int[maze.length][maze[0].length];

for (int[] row : distance){

Arrays.fill(row, Integer.MAX\_VALUE);

}

distance[start[0]][start[1]] = 0;

dfs(maze, start, destination, distance);

return distance[destination[0]][destination[1]] != Integer.MAX\_VALUE ? distance[destination[0]][destination[1]] : -1;

}

private void dfs(int[][] maze, int[] start, int[] destination, int[][] distance){

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

int m = maze.length;

int n = maze[0].length;

for (int i = 0; i < dx.length; i++){

int x = start[0] + dx[i];

int y = start[1] + dy[i];

int dist = 0;

while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] == 0){

x += dx[i];

y += dy[i];

dist++;

}

if (distance[start[0]][start[1]] + dist < distance[x-dx[i]][y-dy[i]]){

distance[x-dx[i]][y-dy[i]] = distance[start[0]][start[1]] + dist;

dfs(maze, new int[]{x-dx[i], y-dy[i]}, destination, distance);

}

}

}

}

Time complexity : O(m∗n∗max(m,n)). Complete traversal of maze will be done in the worst case.

Space complexity : O(mn).

Method 3: Dijkstra Algorithm

Time complexity : O((mn)^2). Complete traversal of maze will be done in the worst case and function minDistance takes O(mn) time.

Space complexity : O(mn). distancedistancedistance array of size m∗n is used.

public class Solution {

public int shortestDistance(int[][] maze, int[] start, int[] dest) {

int[][] distance = new int[maze.length][maze[0].length];

boolean[][] visited = new boolean[maze.length][maze[0].length];

for (int[] row: distance)

Arrays.fill(row, Integer.MAX\_VALUE);

distance[start[0]][start[1]] = 0;

dijkstra(maze, distance, visited);

return distance[dest[0]][dest[1]] == Integer.MAX\_VALUE ? -1 : distance[dest[0]][dest[1]];

}

public int[] minDistance(int[][] distance, boolean[][] visited) {

int[] min={-1,-1};

int min\_val = Integer.MAX\_VALUE;

for (int i = 0; i < distance.length; i++) {

for (int j = 0; j < distance[0].length; j++) {

if (!visited[i][j] && distance[i][j] < min\_val) {

min = new int[] {i, j};

min\_val = distance[i][j];

}

}

}

return min;

}

public void dijkstra(int[][] maze, int[][] distance, boolean[][] visited) {

int[][] dirs={{0,1},{0,-1},{-1,0},{1,0}};

while (true) {

int[] s = minDistance(distance, visited);

if (s[0] < 0)

break;

visited[s[0]][s[1]] = true;

for (int[] dir: dirs) {

int x = s[0] + dir[0];

int y = s[1] + dir[1];

int count = 0;

while (x >= 0 && y >= 0 && x < maze.length && y < maze[0].length && maze[x][y] == 0) {

x += dir[0];

y += dir[1];

count++;

}

if (distance[s[0]][s[1]] + count < distance[x - dir[0]][y - dir[1]]) {

distance[x - dir[0]][y - dir[1]] = distance[s[0]][s[1]] + count;

}

}

}

}

}

Method 4: Dijkstra Algorithm and Priority Queue (best solution)

Time complexity : O(mn∗log(mn)). Complete traversal of maze will be done in the worst case giving a factor of mn.

Further, poll method is a combination of heapifying(O(log(n)) and removing the top elementO(1) from the priority queue,

and it takes O(n) time for nnn elements. In the current case, poll introduces a factor of log(mn)log(mn)log(mn).

Space complexity : O(mn). distancedistancedistance array of size m∗n is used and queuequeuequeue size can grow upto m∗nm\*nm∗n in

worst case.

public class Solution {

public int shortestDistance(int[][] maze, int[] start, int[] dest) {

int[][] distance = new int[maze.length][maze[0].length];

for (int[] row: distance)

Arrays.fill(row, Integer.MAX\_VALUE);

distance[start[0]][start[1]] = 0;

dijkstra(maze, start, distance);

return distance[dest[0]][dest[1]] == Integer.MAX\_VALUE ? -1 : distance[dest[0]][dest[1]];

}

public void dijkstra(int[][] maze, int[] start, int[][] distance) {

int[][] dirs={{0,1},{0,-1},{-1,0},{1,0}};

PriorityQueue < int[] > queue = new PriorityQueue < > ((a, b) -> a[2] - b[2]);

queue.offer(new int[]{start[0],start[1],0});

while (!queue.isEmpty()) {

int[] s = queue.poll();

if(distance[s[0]][s[1]] < s[2])

continue;

for (int[] dir: dirs) {

int x = s[0] + dir[0];

int y = s[1] + dir[1];

int count = 0;

while (x >= 0 && y >= 0 && x < maze.length && y < maze[0].length && maze[x][y] == 0) {

x += dir[0];

y += dir[1];

count++;

}

if (distance[s[0]][s[1]] + count < distance[x - dir[0]][y - dir[1]]) {

distance[x - dir[0]][y - dir[1]] = distance[s[0]][s[1]] + count;

queue.offer(new int[]{x - dir[0], y - dir[1], distance[x - dir[0]][y - dir[1]]});

}

}

}

}

}

506\_Relative.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given scores of N athletes, find their relative ranks and the people with the top three highest scores,

who will be awarded medals: "Gold Medal", "Silver Medal" and "Bronze Medal".

Example 1:

Input: [5, 4, 3, 2, 1]

Output: ["Gold Medal", "Silver Medal", "Bronze Medal", "4", "5"]

Explanation: The first three athletes got the top three highest scores, so they got "Gold Medal", "Silver Medal" and "Bronze Medal".

For the left two athletes, you just need to output their relative ranks according to their scores.

Note:

N is a positive integer and won't exceed 10,000.

All the scores of athletes are guaranteed to be unique.

Method 1:

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public String[] findRelativeRanks(int[] nums) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++){

map.put(nums[i], i);

}

Arrays.sort(nums);

int n = nums.length;

String[] result = new String[n];

for (int i = 0; i < n; i++){

int index = map.get(nums[i]);

if (i == n - 1){

result[index] = "Gold Medal";

}else if (i == n - 2){

result[index] = "Silver Medal";

}else if (i == n - 3){

result[index] = "Bronze Medal";

}else{

result[index] = Integer.toString(n-i);

}

}

return result;

}

}

class Solution {

public String[] findRelativeRanks(int[] nums) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++){

map.put(nums[i], i);

}

Arrays.sort(nums);

String[] res = new String[nums.length];

for (int i = nums.length - 1; i >= 0; i--){

int index = map.get(nums[i]);

if (i == nums.length - 1){

res[index] = "Gold Medal";

} else if (i == nums.length - 2){

res[index] = "Silver Medal";

}else if (i == nums.length - 3){

res[index] = "Bronze Medal";

}else{

res[index] = String.valueOf(nums.length - i);

}

}

return res;

}

}

507\_Perfect.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We define the Perfect Number is a positive integer that is equal to the sum of all its positive divisors except itself.

Now, given an integer n, write a function that returns true when it is a perfect number and false when it is not.

Example:

Input: 28

Output: True

Explanation: 28 = 1 + 2 + 4 + 7 + 14

Note: The input number n will not exceed 100,000,000. (1e8)

class Solution {

public boolean checkPerfectNumber(int num) {

if (num <= 1){

return false;

}

int sum = 0;

for (int i = num / 2; i > (int) Math.sqrt(num); i--){

if (num % i == 0){

sum += i;

if (i != num / i){

sum += num / i;

}

}

}

return sum + 1 == num;

}

}

508\_MostFrequentSubtreeSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the root of a tree, you are asked to find the most frequent subtree sum. The subtree sum of a

node is defined as the sum of all the node values formed by the subtree rooted at that node (including the node itself).

So what is the most frequent subtree sum value? If there is a tie, return all the values with the highest frequency in

any order.

Examples 1

Input:

5

/ \

2 -3

return [2, -3, 4], since all the values happen only once, return all of them in any order.

Examples 2

Input:

5

/ \

2 -5

return [2], since 2 happens twice, however -5 only occur once.

Note: You may assume the sum of values in any subtree is in the range of 32-bit signed integer.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int[] findFrequentTreeSum(TreeNode root) {

Map<Integer, Integer> map = new HashMap<>();

int totalSum = subtreeSum(root, map);

int maxfreq = 0;

int sum = 0;

for (Integer i : map.values()){

if (i > maxfreq){

maxfreq = i;

}

}

List<Integer> result = new ArrayList<>();

for (Map.Entry<Integer, Integer> entry : map.entrySet()){

if (maxfreq == entry.getValue()){

sum = entry.getKey();

result.add(sum);

}

}

int[] ans = new int[result.size()];

for (int i = 0; i < result.size(); i++){

ans[i] = result.get(i);

}

return ans;

}

private int subtreeSum(TreeNode root, Map<Integer, Integer> map){

if (root == null){

return 0;

}

int left = subtreeSum(root.left, map);

int right = subtreeSum(root.right, map);

int sum = left + right + root.val;

map.put(sum, map.getOrDefault(sum, 0) + 1);

return sum;

}

}

51\_N-Queens.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other.

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.'

both indicate a queen and an empty space respectively.

Have you met this question in a real interview? Yes

Example

There exist two distinct solutions to the 4-queens puzzle:

[

// Solution 1

[".Q..",

"...Q",

"Q...",

"..Q."

],

// Solution 2

["..Q.",

"Q...",

"...Q",

".Q.."

]

]

public class Solution {

/\*

\* @param n: The number of queens

\* @return: All distinct solutions

\*/

public List<List<String>> solveNQueens(int n) {

List<List<String>> result = new ArrayList<>();

if (n <=0){

return result;

}

search(result, new ArrayList<Integer>(), n);

return result;

}

private void search(List<List<String>> result, List<Integer> cols, int n){

if (cols.size() == n){

result.add(drawChessBoard(new ArrayList<Integer>(cols)));

return;

}

for (int colIndex = 0; colIndex < n; colIndex++){

if (!isValid(cols, colIndex)){

continue;

}

cols.add(colIndex);

search(result, cols, n);

cols.remove(cols.size() - 1);

}

}

private boolean isValid(List<Integer> cols, int colIndex){

int rowIndex = cols.size();

for (int i = 0; i < rowIndex; i++){

if (cols.get(i) == colIndex){

return false;

}

if (cols.get(i) + i == colIndex + rowIndex){

return false;

}

if (cols.get(i) - i == colIndex - rowIndex){

return false;

}

}

return true;

}

private List<String> drawChessBoard(List<Integer> cols){

ArrayList<String> result = new ArrayList<>();

int size = cols.size();

for (int i = 0; i < size; i++){

StringBuilder sb = new StringBuilder();

for (int j = 0; j < size; j++){

if (j == cols.get(i)){

sb.append('Q');

}else{

sb.append('.');

}

}

result.add(sb.toString());

}

return result;

}

}

511\_SwapTwoNodesLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a linked list and two values v1 and v2. Swap the two nodes in the linked list with values v1 and v2.

It's guaranteed there is no duplicate values in the linked list. If v1 or v2 does not exist in the given linked list,

do nothing.

Notice

You should swap the two nodes with values v1 and v2. Do not directly swap the values of the two nodes.

Have you met this question in a real interview? Yes

Example

Given 1->2->3->4->null and v1 = 2, v2 = 4.

Return 1->4->3->2->null.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

/\*

\* @param head: a ListNode

\* @param v1: An integer

\* @param v2: An integer

\* @return: a new head of singly-linked list

\*/

public ListNode swapNodes(ListNode head, int v1, int v2) {

ListNode dummy = new ListNode(-1);

dummy.next = head;

ListNode prev = dummy;

ListNode curt = head;

int count = 0;

ListNode v1Prev = dummy, v2Prev = dummy, v1Curt = dummy, v2Curt = dummy;

while (curt != null){

if (curt.val == v1){

v1Prev = prev;

v1Curt = curt;

count++;

}

if (curt.val == v2){

v2Prev = prev;

v2Curt = curt;

count++;

}

if (count == 2){

v1Prev.next = v2Curt;

v2Prev.next = v1Curt;

ListNode temp = v1Curt.next;

v1Curt.next = v2Curt.next;

v2Curt.next = temp;

break;

}

prev = curt;

curt = curt.next;

}

return dummy.next;

}

}

class Solution {

public ListNode swapPairs(ListNode head) {

if (head == null || head.next == null){

return head;

}

ListNode dummy = new ListNode(-1);

dummy.next = head;

ListNode prev = dummy;

ListNode cur = head;

ListNode next = head.next;

while (cur != null && next != null){

ListNode temp = next.next;

next.next = cur;

cur.next = temp;

prev.next = next;

prev = cur;

cur = temp;

if (temp != null){

next= temp.next;

}

}

return dummy.next;

}

}

512\_DecodeWays.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' -> 1

'B' -> 2

...

'Z' -> 26

Given an encoded message containing digits, determine the total number of ways to decode it.

Have you met this question in a real interview? Yes

Example

Given encoded message 12, it could be decoded as AB (1 2) or L (12).

The number of ways decoding 12 is 2.

Method 1:

Dynamic Programming:

• 此题解法:考虑最后一位怎么分解?

• f[n]表示从第0层走到第n层一共有多少种方法

• f[i]=f[i-1](条件s[i]!=0) + f[i-2](条件是s[i-1]与s[i]组成的数字在10-26之间)

• Note that the length of f should be s.length + 1

public class Solution {

/\*

\* @param s: a string, encoded message

\* @return: an integer, the number of ways decoding

\*/

public int numDecodings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int[] f = new int[s.length() + 1];

f[0] = 1;

f[1] = s.charAt(0) != '0' ? 1 : 0;

for (int i = 2; i <= s.length(); i++){

if (s.charAt(i-1) != '0'){

f[i] += f[i-1];

}

int twoDigits = (s.charAt(i-2) - '0') \* 10 + (s.charAt(i-1) - '0');

if (twoDigits >= 10 && twoDigits <= 26){

f[i] += f[i-2];

}

}

return f[s.length()];

}

}

Method 2:

public class Solution {

/\*

\* @param s: a string, encoded message

\* @return: an integer, the number of ways decoding

\*/

public int numDecodings(String s) {

if (s == null || s.length() == 0){

return 0;

}

char[] sCharArr = s.toCharArray();

int[] f = new int[s.length() + 1];

f[0] = 1;

f[1] = sCharArr[0] != '0'? 1 : 0;

for (int i = 2; i <= s.length(); i++){

if (sCharArr[i-1] != '0'){

f[i] += f[i-1];

}

int twoDigits = (sCharArr[i-2] - '0') \* 10 + (sCharArr[i-1] - '0');

if (twoDigits >= 10 && twoDigits <= 26){

f[i] += f[i-2];

}

}

return f[s.length()];

}

}

Better:

class Solution {

public int numDecodings(String s) {

if (s.charAt(0) == '0'){

return 0;

}

int n = s.length();

int[] dp = new int[n+1];

dp[0] = 1;

dp[1] = 1;

for (int i = 2; i <= n; i++){

int curr = (int)(s.charAt(i-1) - '0');

int prev = (int)(s.charAt(i-2) - '0');

int num = prev \* 10 + curr;

if (curr != 0){

dp[i] += dp[i-1];

}

if (num >= 10 && num <= 26){

dp[i] += dp[i-2];

}

}

return dp[n];

}

}

Best solution:

class Solution {

public int numDecodings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int n = s.length();

if (n == 1){

if (s.charAt(0) != '0'){

return 1;

}else{

return 0;

}

}

if (s.charAt(0) == '0'){

return 0;

}

int[] dp = new int[n+1];

dp[0] = 1;

dp[1] = 1;

for (int i = 2; i <= n; i++){

int curr = (int)(s.charAt(i-1) - '0');

int prev = (int)(s.charAt(i-2) - '0');

if (curr != 0){

dp[i] += dp[i-1];

}

int num = prev \* 10 + curr;

if (num >= 10 && num <= 26){

dp[i] += dp[i-2];

}

}

return dp[n];

}

}

513\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, find the leftmost value in the last row of the tree.

Example 1:

Input:

2

/ \

1 3

Output:

1

Example 2:

Input:

1

/ \

2 3

/ / \

4 5 6

/

7

Output:

7

Note: You may assume the tree (i.e., the given root node) is not NULL.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int findBottomLeftValue(TreeNode root) {

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

int ans = root.val;

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

if (i == 0){

ans = node.val;

}

if (node.left != null){

queue.offer(node.left);

}

if (node.right != null){

queue.offer(node.right);

}

}

}

return ans;

}

}

514\_Freedom.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In the video game Fallout 4, the quest "Road to Freedom" requires players to reach a metal dial called the "Freedom Trail Ring", and use the dial to spell a specific keyword in order to open the door.

Given a string ring, which represents the code engraved on the outer ring and another string key, which represents the keyword needs to be spelled. You need to find the minimum number of steps in order to spell all the characters in the keyword.

Initially, the first character of the ring is aligned at 12:00 direction. You need to spell all the characters in the string key one by one by rotating the ring clockwise or anticlockwise to make each character of the string key aligned at 12:00 direction and then by pressing the center button.

At the stage of rotating the ring to spell the key character key[i]:

You can rotate the ring clockwise or anticlockwise one place, which counts as 1 step. The final purpose of the rotation is to align one of the string ring's characters at the 12:00 direction, where this character must equal to the character key[i].

If the character key[i] has been aligned at the 12:00 direction, you need to press the center button to spell, which also counts as 1 step. After the pressing, you could begin to spell the next character in the key (next stage), otherwise, you've finished all the spelling.

Example:

Input: ring = "godding", key = "gd"

Output: 4

Explanation:

For the first key character 'g', since it is already in place, we just need 1 step to spell this character.

For the second key character 'd', we need to rotate the ring "godding" anticlockwise by two steps to make it become "ddinggo".

Also, we need 1 more step for spelling.

So the final output is 4.

Note:

Length of both ring and key will be in range 1 to 100.

There are only lowercase letters in both strings and might be some duplcate characters in both strings.

It's guaranteed that string key could always be spelled by rotating the string ring.

Method 1:

Time complexity: O(m\*n^2)

Space complexity: O(mn)

class Solution {

public int findRotateSteps(String ring, String key) {

int m = key.length();

int n = ring.length();

int[][] dp = new int[m][n];

for (int j = 0; j < n; j++){

if (key.charAt(0) == ring.charAt(j)){

dp[0][j]= Math.min(j + 1, n - j + 1);

}

}

for (int i = 1; i < m; i++){

for (int j = 0; j < n ;j++){

if (key.charAt(i) == ring.charAt(j)){

dp[i][j] = Integer.MAX\_VALUE;

for (int k = 0; k < n; k++){

if (dp[i-1][k] != 0){

int diff = Math.abs(j - k);

int min = Math.min(diff + 1, n - diff + 1);

dp[i][j] = Math.min(dp[i][j], dp[i-1][k] + min);

}

}

}

}

}

int ans = Integer.MAX\_VALUE;

for (int j = 0; j < n; j++){

if (dp[m-1][j] != 0){

ans = Math.min(ans, dp[m-1][j]);

}

}

return ans;

}

}

515\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You need to find the largest value in each row of a binary tree.

Example:

Input:

1

/ \

3 2

/ \ \

5 3 9

Output: [1, 3, 9]

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

Method 1: BFS

class Solution {

public List<Integer> largestValues(TreeNode root) {

List<Integer> res = new ArrayList<>();

if (root == null){

return res;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

int size = queue.size();

int max = Integer.MIN\_VALUE;

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

max = Math.max(max, node.val);

if (node.left != null){

queue.offer(node.left);

}

if (node.right != null){

queue.offer(node.right);

}

}

res.add(max);

}

return res;

}

}

Method 2: DFS

Just a simple pre-order traverse idea. Use depth to expand result list size and put the max value in the appropriate position.

class Solution {

public List<Integer> largestValues(TreeNode root) {

List<Integer> res = new ArrayList<>();

preOrder(res, root, 0);

return res;

}

private void preOrder(List<Integer> res, TreeNode root, int depth){

if (root == null){

return;

}

if (depth == res.size()){

res.add(root.val);

}else{

res.set(depth, Math.max(root.val, res.get(depth)));

}

preOrder(res, root.left, depth+1);

preOrder(res, root.right, depth+1);

}

}

# 517\_Super.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have n super washing machines on a line. Initially, each washing machine has some dresses or is empty.

For each move, you could choose any m (1 ≤ m ≤ n) washing machines, and pass one dress of each washing machine to one of its adjacent

washing machines at the same time .

Given an integer array representing the number of dresses in each washing machine from left to right on the line, you should find the

minimum number of moves to make all the washing machines have the same number of dresses. If it is not possible to do it, return -1.

Example1

Input: [1,0,5]

Output: 3

Explanation:

1st move: 1 0 <-- 5 => 1 1 4

2nd move: 1 <-- 1 <-- 4 => 2 1 3

3rd move: 2 1 <-- 3 => 2 2 2

Example2

Input: [0,3,0]

Output: 2

Explanation:

1st move: 0 <-- 3 0 => 1 2 0

2nd move: 1 2 --> 0 => 1 1 1

Example3

Input: [0,2,0]

Output: -1

Explanation:

It's impossible to make all the three washing machines have the same number of dresses.

Note:

The range of n is [1, 10000].

The range of dresses number in a super washing machine is [0, 1e5].

https://leetcode.com/problems/super-washing-machines/discuss/99185/Super-Short-and-Easy-Java-O(n)-Solution

class Solution {

public int findMinMoves(int[] machines) {

int n = machines.length;

int sum = 0;

for (int i : machines){

sum += i;

}

if (sum % n != 0){

return -1;

}

int ave = sum / n;

int global = 0;

int local = 0;

int cand = 0;

for (int i = 0; i < n; i++){

int diff = machines[i] - ave;

cand += diff;

local = Math.max(diff, Math.abs(cand));

global = Math.max(global, local);

}

return global;

}

}

# 518\_CoinChange2.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given coins of different denominations and a total amount of money. Write a function to compute

the number of combinations that make up that amount. You may assume that you have infinite number of each kind of coin.

Note: You can assume that

0 <= amount <= 5000

1 <= coin <= 5000

the number of coins is less than 500

the answer is guaranteed to fit into signed 32-bit integer

Example 1:

Input: amount = 5, coins = [1, 2, 5]

Output: 4

Explanation: there are four ways to make up the amount:

5=5

5=2+2+1

5=2+1+1+1

5=1+1+1+1+1

Example 2:

Input: amount = 3, coins = [2]

Output: 0

Explanation: the amount of 3 cannot be made up just with coins of 2.

Example 3:

Input: amount = 10, coins = [10]

Output: 1

https://leetcode.com/problems/coin-change-2/discuss/141076/Logical-Thinking-with-Clear-Java-Code

class Solution {

public int change(int amount, int[] coins) {

int[] dp = new int[amount+1];

dp[0] = 1;

for (int i = 0; i < coins.length; i++){

for (int j = 1; j <= amount; j++){

if (j - coins[i] >= 0){

dp[j] += dp[j-coins[i]];

}

}

}

/\* this following code is not working : you will need to derive an equivalent formula to update dp

which may not be so obvious since you must deal with duplicated cases.

You can check #322 for Coin Change 1 where we could easily swap two for-loops since there we only compute min. numbers.

for (int i = 1; i <= amount; i++){

for (int j = 0; j < coins.length; j++){

if (i - coins[j] >= 0){

dp[i] += dp[i-coins[j]];

}

}

}\*/

return dp[amount];

}

}

/\*

Actually the above solution is reduced from 2D space complexity to 1D space complexity

backpack DP:

dp denotes the number of ways to reach amount with the number of count coins:

two cases: 1. take the last coin: the amount must be greater than lastCoin. then

dp[amount][count] += dp[amount - lastCoin][count-1];

2. not take the last coin: dp[amount][count] += dp[amount][count-1]

dp[amount][count] = dp[amount - lastCoin][count-1] + dp[amount][count-1]

\*/

then since count only replies on prevous step, reduced to 1D space dp[amount]

# 52\_N-QueensII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.

class Solution {

public int totalNQueens(int n) {

if (n <= 0){

return 0;

}

List<List<Integer>> result = new ArrayList<>();

dfs(result, new ArrayList<Integer>(), n);

return result.size();

}

private void dfs(List<List<Integer>> result, List<Integer> cols, int n){

if (cols.size() == n){

result.add(new ArrayList<Integer>(cols));

return;

}

for (int i = 0; i < n; i++){

if (isValid(cols, i)){

cols.add(i);

dfs(result, cols, n);

cols.remove(cols.size() - 1);

}

}

}

private boolean isValid(List<Integer> cols, int colIndex){

int rowIndex = cols.size();

for (int i = 0; i < rowIndex; i++){

if (colIndex == cols.get(i)){

return false;

}

if (colIndex + rowIndex == cols.get(i) + i){

return false;

}

if (colIndex - rowIndex == cols.get(i) - i){

return false;

}

}

return true;

}

}

# 520\_Detect.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a word, you need to judge whether the usage of capitals in it is right or not.

We define the usage of capitals in a word to be right when one of the following cases holds:

All letters in this word are capitals, like "USA".

All letters in this word are not capitals, like "leetcode".

Only the first letter in this word is capital if it has more than one letter, like "Google".

Otherwise, we define that this word doesn't use capitals in a right way.

Example 1:

Input: "USA"

Output: True

Example 2:

Input: "FlaG"

Output: False

Note: The input will be a non-empty word consisting of uppercase and lowercase latin letters.

Method 1: counting

class Solution {

public boolean detectCapitalUse(String word) {

int count = 0;

for (int i = 0; i < word.length(); i++){

if (Character.isUpperCase(word.charAt(i))){

count++;

}

}

return (count == 0 || count == word.length() || (count == 1 && Character.isUpperCase(word.charAt(0))));

}

}

Method 2:

class Solution {

public boolean detectCapitalUse(String word) {

if (word == null || word.length() < 2){

return true;

}

if (word.substring(1).equals(word.substring(1).toLowerCase())){

return true;

}

if (word.equals(word.toUpperCase())){

return true;

}

return false;

}

}

Method 3:

class Solution {

public boolean detectCapitalUse(String word) {

if (word == null || word.length() == 0){

return false;

}

boolean startCap = false;

if (word.charAt(0) - 'A' < 26){

startCap = true;

}

boolean secondCap = false;

for (int i = 1; i < word.length(); i++){

if (!startCap){

if (word.charAt(i) - 'A' < 26)

return false;

}else{

if (i == 1){

if (word.charAt(1) - 'A' < 26)

secondCap = true;

}else{

if (!secondCap){//second one is lower case

if (word.charAt(i) - 'A' < 26)

return false;

}else{//second one is upper case

if (word.charAt(i) - 'A' > 26)

return false;

}

}

}

}

return true;

}

}

class Solution {

public boolean detectCapitalUse(String word) {

if (word == null || word.length() == 0){

return false;

}

boolean startCap = false;

if (Character.isUpperCase(word.charAt(0))){

startCap = true;

}

boolean secondCap = false;

for (int i = 1; i < word.length(); i++){

if (!startCap){

if (Character.isUpperCase(word.charAt(i)))

return false;

}else{

if (i == 1){

if (Character.isUpperCase(word.charAt(1)))

secondCap = true;

}else{

if (!secondCap){//second one is lower case

if (Character.isUpperCase(word.charAt(i)))

return false;

}else{//second one is upper case

if (Character.isLowerCase(word.charAt(i)))

return false;

}

}

}

}

return true;

}

}

Best solution:

class Solution {

public boolean detectCapitalUse(String word) {

boolean cap = false;

if (Character.isUpperCase(word.charAt(word.length() - 1))){

cap = true;

}

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (i == 0){

if (Character.isLowerCase(c) && cap){

return false;

}

}else{

if (Character.isLowerCase(c) && cap || Character.isUpperCase(c) && !cap){

return false;

}

}

}

return true;

}

}

# 523\_Continuous.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of non-negative numbers and a target integer k, write a function to check if the array has a continuous

subarray of size at least 2 that sums up to the multiple of k, that is, sums up to n\*k where n is also an integer.

Example 1:

Input: [23, 2, 4, 6, 7], k=6

Output: True

Explanation: Because [2, 4] is a continuous subarray of size 2 and sums up to 6.

Example 2:

Input: [23, 2, 6, 4, 7], k=6

Output: True

Explanation: Because [23, 2, 6, 4, 7] is an continuous subarray of size 5 and sums up to 42.

Note:

The length of the array won't exceed 10,000.

You may assume the sum of all the numbers is in the range of a signed 32-bit integer.

Best solution:

Method 1: Similar to Leetcode 560 Subarray Sum Equals K

Note that (a+b) % c = a % c + b % c,

so to find multiple of k, just find presum % k are equal

class Solution {

public boolean checkSubarraySum(int[] nums, int k) {

Map<Integer, Integer> map = new HashMap<>();

map.put(0, -1);

int preSum = 0;

for (int i = 0; i < nums.length; i++){

preSum += nums[i];

if (k != 0){

preSum = preSum % k;

}

if (map.containsKey(preSum)){

if (i - map.get(preSum) > 1){

return true;

}

}else{

map.put(preSum, i);

}

}

return false;

}

}

class Solution {

public boolean checkSubarraySum(int[] nums, int k) {

int n = nums.length;

int sum = 0;

Map<Integer, Integer> map = new HashMap<>();

map.put(0, -1);

for (int i = 0; i < n; i++){

sum += nums[i];

if (k != 0){

sum %= k;

}

if (map.containsKey(sum)){

if (i - map.get(sum) > 1){

return true;

}

}else{

map.put(sum, i);

}

}

return false;

}

}

# 524\_Longest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string and a string dictionary, find the longest string in the dictionary that can be formed by

deleting some characters of the given string. If there are more than one possible results, return the longest

word with the smallest lexicographical order. If there is no possible result, return the empty string.

Example 1:

Input:

s = "abpcplea", d = ["ale","apple","monkey","plea"]

Output:

"apple"

Example 2:

Input:

s = "abpcplea", d = ["a","b","c"]

Output:

"a"

Note:

All the strings in the input will only contain lower-case letters.

The size of the dictionary won't exceed 1,000.

The length of all the strings in the input won't exceed 1,000.

Method 1: sorting

Time complexity: O(x\*n\*logn + n\*x) Here nn refers to the number of strings in list dd and xx refers to average string length.

class Solution {

public String findLongestWord(String s, List<String> d) {

Collections.sort(d, new Comparator<String>(){

public int compare (String a, String b){

if (a.length() == b.length()){

return a.compareTo(b);

}

return b.length() - a.length();

}

});

for (String str : d){

int i = 0; // str string

int j = 0; // s string

while (i < str.length() && j < s.length()){

while (j < s.length() && str.charAt(i) != s.charAt(j)){

j++;

}

if (j == s.length()){

break;

}

i++;

j++;

}

if (i == str.length()){

return str;

}

}

return "";

}

}

Method 2: Better version of Two pointers

class Solution {

public String findLongestWord(String s, List<String> d) {

Collections.sort(d, new Comparator<String>(){

public int compare (String a, String b){

if (a.length() == b.length()){

return a.compareTo(b);

}

return b.length() - a.length();

}

});

for (String str : d){

int i = 0; // str string

int j = 0; // s string

while (i < str.length() && j < s.length()){

if (str.charAt(i) == s.charAt(j)){

i++;

}

j++;

}

if (i == str.length()){

return str;

}

}

return "";

}

}

Method 3: without sorting

Time complexity: O(n\*x) Here nn refers to the number of strings in list dd and xx refers to average string length.

class Solution {

public String findLongestWord(String s, List<String> d) {

String ans = "";

for (String str : d){

int i = 0; // str string

int j = 0; // s string

while (i < str.length() && j < s.length()){

if (str.charAt(i) == s.charAt(j)){

i++;

}

j++;

}

if (i == str.length() && str.length() >= ans.length()){

if (str.length() > ans.length() || str.compareTo(ans) < 0){

ans = str;

}

}

}

return ans;

}

}

# 525\_Contiguous.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1.

Example 1:

Input: [0,1]

Output: 2

Explanation: [0, 1] is the longest contiguous subarray with equal number of 0 and 1.

Example 2:

Input: [0,1,0]

Output: 2

Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1.

Note: The length of the given binary array will not exceed 50,000.

Method 1: TLE

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public int findMaxLength(int[] nums) {

int dp = 0;

for (int i = 0; i < nums.length; i++){

int one = 0;

int zero = 0;

for (int j = i; j < nums.length; j++){

if (nums[j] == 1){

one++;

}else{

zero++;

}

if (one == zero){

dp = Math.max(dp, one \* 2);

}

}

}

return dp;

}

}

Method 2: preSum + HashMap

In this approach, we make use of a countcount variable, which is used to store the relative number of ones and zeros

encountered so far while traversing the array. The countcount variable is incremented by one for every \text{1}1 encountered

and the same is decremented by one for every \text{0}0 encountered.

We start traversing the array from the beginning. If at any moment, the countcount becomes zero, it implies that we've encountered equal number of zeros and ones from the beginning till the current index of the array(ii). Not only this, another point to be noted is that if we encounter the same countcount twice while traversing the array, it means that the number of zeros and ones are equal between the indices

corresponding to the equal countcount values. The following figure illustrates the observation for the sequence [0 0 1 0 0 0 1 1]:

https://leetcode.com/articles/contiguous-array/

class Solution {

public int findMaxLength(int[] nums) {

int max = 0;

int count = 0;

Map<Integer, Integer> map = new HashMap<>();//map stores sum-index pair

map.put(0, -1); //initialize: sum = 0; index = -1;

for (int i = 0; i < nums.length; i++){

if (nums[i] == 0){

count--;

}else{

count++;

}

if (!map.containsKey(count)){

map.put(count, i);

}else{

max = Math.max(max, i - map.get(count));

}

}

return max;

}

}

# 526\_Beautiful.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Suppose you have N integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these N numbers successfully if one of the following is true for the ith position (1 <= i <= N) in this array:

The number at the ith position is divisible by i.

i is divisible by the number at the ith position.

Now given N, how many beautiful arrangements can you construct?

Example 1:

Input: 2

Output: 2

Explanation:

The first beautiful arrangement is [1, 2]:

Number at the 1st position (i=1) is 1, and 1 is divisible by i (i=1).

Number at the 2nd position (i=2) is 2, and 2 is divisible by i (i=2).

The second beautiful arrangement is [2, 1]:

Number at the 1st position (i=1) is 2, and 2 is divisible by i (i=1).

Number at the 2nd position (i=2) is 1, and i (i=2) is divisible by 1.

Method: Backtracking

class Solution {

public int countArrangement(int N) {

List<List<Integer>> res = new ArrayList<>();

int[] arr = new int[N];

for (int i = 0; i < N; i++){

arr[i] = i+1;

}

backtrack(res, new ArrayList<Integer>(), arr, 1);

return res.size();

}

private void backtrack(List<List<Integer>> res, List<Integer> item, int[] arr, int pos){

if (pos == arr.length + 1){

if (item.size() == arr.length){

res.add(new ArrayList<Integer>(item));

}

return;

}

for (int i = 0; i < arr.length; i++){

if (item.contains(arr[i])){

continue;

}

if (arr[i] % pos == 0 || pos % arr[i] == 0){

item.add(arr[i]);

backtrack(res, item, arr, pos+1);

item.remove(item.size() - 1);

}

}

}

}

Backtracking again

class Solution {

int count = 0;

public int countArrangement(int N) {

backtrack(N, 1, new boolean[N+1]);

return count;

}

private void backtrack(int N, int pos, boolean[] visited){

if (pos == N + 1){

count++;

return;

}

for (int i = 1; i <= N; i++){

if (!visited[i] && (i % pos == 0 || pos % i == 0)){

visited[i] = true;

backtrack(N, pos+1, visited);

visited[i] = false;

}

}

}

}

class Solution {

public int countArrangement(int N) {

List<List<Integer>> res = new ArrayList<>();

int[] arr = new int[N];

for (int i = 0; i < N; i++){

arr[i] = i + 1;

}

boolean[] visited = new boolean[N];

dfs(res, new ArrayList<>(), arr, 0, visited);

return res.size();

}

private void dfs(List<List<Integer>> res, List<Integer> item, int[] arr, int pos, boolean[] visited){

if (pos == arr.length){

if (item.size() == arr.length){

res.add(new ArrayList<>(item));

}

return;

}

for (int i = 0; i < arr.length; i++){

if (visited[i]){

continue;

}

if (arr[i] % (pos+1) == 0 || (pos+1) % arr[i] == 0){

visited[i] = true;

item.add(arr[i]);

dfs(res, item, arr, pos + 1, visited);

item.remove(item.size() - 1);

visited[i] = false;

}

}

}

}

# 528\_FlattenNestedListIterator.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Notice

You don't need to implement the remove method.

Have you met this question in a real interview? Yes

Example

Given the list [[1,1],2,[1,1]], By calling next repeatedly until hasNext returns false, the order

of elements returned by next should be: [1,1,2,1,1].

Given the list [1,[4,[6]]], By calling next repeatedly until hasNext returns false, the order of

elements returned by next should be: [1,4,6].

/\*\*

\* // This is the interface that allows for creating nested lists.

\* // You should not implement it, or speculate about its implementation

\* public interface NestedInteger {

\*

\* // @return true if this NestedInteger holds a single integer,

\* // rather than a nested list.

\* public boolean isInteger();

\*

\* // @return the single integer that this NestedInteger holds,

\* // if it holds a single integer

\* // Return null if this NestedInteger holds a nested list

\* public Integer getInteger();

\*

\* // @return the nested list that this NestedInteger holds,

\* // if it holds a nested list

\* // Return null if this NestedInteger holds a single integer

\* public List<NestedInteger> getList();

\* }

\*/

import java.util.Iterator;

public class NestedIterator implements Iterator<Integer> {

Stack<NestedInteger> stack;

public NestedIterator(List<NestedInteger> nestedList) {

stack = new Stack<>();

pushListToStack(nestedList);

}

// @return {int} the next element in the iteration

@Override

public Integer next() {

if(!hasNext()){

return null;

}

return stack.pop().getInteger();

}

// @return {boolean} true if the iteration has more element or false

@Override

public boolean hasNext() {

while (!stack.isEmpty() && !stack.peek().isInteger()){

pushListToStack(stack.pop().getList());

}

return !stack.isEmpty();

}

private void pushListToStack(List<NestedInteger> list){

Stack<NestedInteger> temp = new Stack<>();

for (NestedInteger n : list){

temp.push(n);

}

while (!temp.isEmpty()){

stack.push(temp.pop());

}

}

@Override

public void remove() {}

}

/\*\*

\* Your NestedIterator object will be instantiated and called as such:

\* NestedIterator i = new NestedIterator(nestedList);

\* while (i.hasNext()) v.add(i.next());

\*/

Best soluton:

/\*\*

\* // This is the interface that allows for creating nested lists.

\* // You should not implement it, or speculate about its implementation

\* public interface NestedInteger {

\*

\* // @return true if this NestedInteger holds a single integer, rather than a nested list.

\* public boolean isInteger();

\*

\* // @return the single integer that this NestedInteger holds, if it holds a single integer

\* // Return null if this NestedInteger holds a nested list

\* public Integer getInteger();

\*

\* // @return the nested list that this NestedInteger holds, if it holds a nested list

\* // Return null if this NestedInteger holds a single integer

\* public List<NestedInteger> getList();

\* }

\*/

public class NestedIterator implements Iterator<Integer> {

Stack<NestedInteger> stack;

public NestedIterator(List<NestedInteger> nestedList) {

stack = new Stack<>();

for (int i = nestedList.size() - 1; i >= 0; i--){

stack.push(nestedList.get(i));

}

}

@Override

public Integer next() {

return stack.pop().getInteger();

}

@Override

public boolean hasNext() {

while (!stack.isEmpty()){

NestedInteger curr = stack.peek();

if (curr.isInteger()){

return true;

}

stack.pop();

List<NestedInteger> list = curr.getList();

for (int i = list.size() - 1; i >= 0; i--){

stack.push(list.get(i));

}

}

return false;

}

}

/\*\*

\* Your NestedIterator object will be instantiated and called as such:

\* NestedIterator i = new NestedIterator(nestedList);

\* while (i.hasNext()) v[f()] = i.next();

\*/

528\_Random.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array w of positive integers, where w[i] describes the weight of index i, write a function pickIndex which randomly

picks an index in proportion to its weight.

Note:

1 <= w.length <= 10000

1 <= w[i] <= 10^5

pickIndex will be called at most 10000 times.

Example 1:

Input:

["Solution","pickIndex"]

[[[1]],[]]

Output: [null,0]

Example 2:

Input:

["Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"]

[[[1,3]],[],[],[],[],[]]

Output: [null,0,1,1,1,0]

Explanation of Input Syntax:

The input is two lists: the subroutines called and their arguments. Solution's constructor has one argument, the array w. pickIndex has no arguments. Arguments are always wrapped with a list, even if there aren't any.

https://leetcode.com/problems/random-pick-with-weight/discuss/154044/Java-accumulated-freq-sum-and-binary-search

Check binary search https://github.com/optimisea/Leetcode/blob/master/Java/35\_Search.java

class Solution {

Random random;

int[] wSum;

public Solution(int[] w) {

random = new Random();

wSum = new int[w.length];

wSum[0] = w[0];

for (int i = 1; i < w.length; i++){

wSum[i] = wSum[i-1] + w[i];

}

}

public int pickIndex() {

int n = wSum.length;

int max = wSum[n-1];

int target = random.nextInt(max) + 1;

//the below will be the same as Leetcode 35: search insert position

int start = 0;

int end = n-1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (wSum[mid] == target){

return mid;

}else if (wSum[mid] < target){

start = mid;

}else{

end = mid;

}

}

if (wSum[start] >= target){

return start;

}else if (wSum[end] >= target){

return end;

}

return n-1;

}

}

Better version for binary search for insert position

class Solution {

Random random;

int[] wSum;

public Solution(int[] w) {

random = new Random();

wSum = new int[w.length];

wSum[0] = w[0];

for (int i = 1; i < w.length; i++){

wSum[i] = wSum[i-1] + w[i];

}

}

public int pickIndex() {

int n = wSum.length;

int max = wSum[n-1];

int target = random.nextInt(max) + 1;

//the below will be the same as Leetcode 35: search insert position

int start = 0;

int end = n-1;

while (start <= end){

int mid = start + (end - start) / 2;

if (wSum[mid] == target){

return mid;

}else if (wSum[mid] < target){

start = mid + 1;

}else{

end = mid - 1;

}

}

return start;

}

}

529\_Minesweeper.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

https://leetcode.com/problems/minesweeper/description/

Method 1:

BFS:

class Solution {

public char[][] updateBoard(char[][] board, int[] click) {

int m = board.length;

int n = board[0].length;

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

int[] dx = {1, 0, -1, -1, -1, 0, 1, 1};

int[] dy = {1, 1, 1, 0, -1, -1, -1, 0};

qx.offer(click[0]);

qy.offer(click[1]);

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

if (board[cx][cy] == 'M'){

board[cx][cy] = 'X';

}else{ //board[cx][cy] == 'E' or 'B'

int count = getMineNumber(board, cx, cy);

if (count > 0){

board[cx][cy] = (char) (count + '0'); // stop search

}else{

board[cx][cy] = 'B';

for (int i = 0; i < dx.length; i++){

int nx = cx + dx[i];

int ny = cy + dy[i];

if (nx >= 0 && nx < m && ny >=0 && ny < n && board[nx][ny] == 'E'){

qx.offer(nx);

qy.offer(ny);

board[nx][ny] = 'B'; //avoid to add into queue again

}

}

}

}

}

return board;

}

private int getMineNumber(char[][] board, int nx, int ny){

int count = 0;

int m = board.length;

int n = board[0].length;

int[] dx = {1, 0, -1, -1, -1, 0, 1, 1};

int[] dy = {1, 1, 1, 0, -1, -1, -1, 0};

for (int i = 0; i < dx.length; i++){

int x = nx + dx[i];

int y = ny + dy[i];

if (x >= 0 && x < m && y >= 0&& y < n){

if (board[x][y] == 'M' || board[x][y] == 'X'){

count++;

}

}

}

return count;

}

}

53\_MaximumSubarray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [-2,1,-3,4,-1,2,1,-5,4],

the contiguous subarray [4,-1,2,1] has the largest sum = 6.

Check Leetcode 918 for the case of circular array

Another extension: maximum product subarray:

https://github.com/optimisea/Leetcode/blob/master/Java/152\_MaximumProductSubarray.java

Method 1: Greey

class Solution {

public int maxSubArray(int[] nums) {

int max = Integer.MIN\_VALUE;

int sum = 0;

for (int num : nums){

sum += num;

max = Math.max(max, sum);

if (sum < 0){

sum = 0;

}

}

return max;

}

}

Better to handle corner case

class Solution {

public int maxSubArray(int[] nums) {

int global = Integer.MIN\_VALUE;

int local = 0;

for (int i = 0; i< nums.length; i++){

local = Math.max(nums[i], local + nums[i]);

global = Math.max(global, local);

}

return global;

}

}

Method 2: DP

f[i] denotes the maxium value of the continuous subarray that ends at the first i elements

public class Solution {

/\*\*

\* @param nums: A list of integers

\* @return: A integer indicate the sum of max subarray

\*/

public int maxSubArray(int[] nums) {

int n = nums.length;

int[] f = new int[n + 1];

f[0] = 0;

for (int i = 1; i <= n; i++){

f[i] = Math.max(nums[i-1], nums[i-1] + f[i-1]);

}

int max = Integer.MIN\_VALUE;

for (int i = 1; i <= n; i++){

max = Math.max(max, f[i]);

}

return max;

}

}

Method 3: rolling array

public class Solution {

/\*\*

\* @param nums: A list of integers

\* @return: A integer indicate the sum of max subarray

\*/

public int maxSubArray(int[] nums) {

int n = nums.length;

int[] f = new int[2];

f[0] = 0;

int max = Integer.MIN\_VALUE;

for (int i = 1; i <= n; i++){

f[i%2] = Math.max(nums[i-1], nums[i-1] + f[(i-1)%2]);

max = Math.max(max, f[i%2]);

}

return max;

}

}

Method 4: DP

Space complexity: O(1)

class Solution {

public int maxSubArray(int[] nums) {

int global = nums[0];

int local = nums[0];

for (int i = 1; i< nums.length; i++){

local = Math.max(nums[i], local + nums[i]);

global = Math.max(global, local);

}

return global;

}

}

If the required output is changed to the desired array not just sum

class Solution {

public int[] maxSubArray(int[] nums) {

int global = Integer.MIN\_VALUE;

int local = 0;

int start = 0;

int end = 0;

int[] index = new int[2];

for (int i = 0; i< nums.length; i++){

if (nums[i] > local + nums[i]){

start = i;

local = nums[i];

}else{

local += nums[i];

}

if (local > global){

end = i;

global = local;

index[0] = start;

index[1] = end;

}

}

int len = index[1] - index[0] + 1;

int[] res = new int[len];

for (int i = index[0]; i <= index[1]; i++){

res[i-index[0]] = nums[i];

}

return res;

}

}

class Solution {

public int[] maxSubArray(int[] nums) {

int max = Integer.MIN\_VALUE;

int sum = 0;

int start = 0;

int end = 0;

int[] index = new int[2];

for (int i = 0; i < nums.length; i++){

sum += nums[i];

if (sum > max){

end = i;

max = sum;

index[0] = start;

index[1] = end;

}

if (sum < 0){

sum = 0;

start = i + 1;

}

}

int len = index[1] - index[0] + 1;

int[] res = new int[len];

for (int i = index[0]; i <= index[1]; i++){

res[i-index[0]] = nums[i];

System.out.println(nums[i]);

}

return max;

}

}

https://github.com/optimisea/Leetcode/blob/master/Java/402\_Continuous.java

public List<Integer> continuousSubarraySum(int[] A) {

List<Integer> result = new ArrayList<>();

result.add(0, 0);

result.add(1, 0);

int local = 0;

int global = Integer.MIN\_VALUE;

int left = 0;

int right = 0;

for (int i = 0; i < A.length; i++){

if (local < 0){

local = A[i];

left = i;

right = i;

}else{

local += A[i];

right = i;

}

if (local >= global){

global = local;

result.set(0, left);

result.set(1, right);

}

}

return result;

}

public class Solution {

/\*

\* @param A: An integer array

\* @return: A list of integers includes the index of the first number and the index of the last number

\*/

public List<Integer> continuousSubarraySum(int[] A) {

List<Integer> res = new ArrayList<>();

res.add(0);

res.add(1);

int sum = 0;

int max = Integer.MIN\_VALUE;

int start = 0;

int end = 0;

for (int i = 0; i < A.length; i++){

sum += A[i];

if (sum > max){

max = sum;

end = i;

res.set(0, start);

res.set(1, end);

}

if (sum < 0){

sum = 0;

start = i + 1;

}

}

return res;

}

}

530\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary search tree with non-negative values, find the minimum absolute difference

between values of any two nodes.

Example:

Input:

1

\

3

/

2

Output:

1

Explanation:

The minimum absolute difference is 1, which is the difference between 2 and 1 (or between 2 and 3).

Method 1:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int getMinimumDifference(TreeNode root) {

List<Integer> list = new ArrayList<>();

preOrder(root, list);

Collections.sort(list);

int diff = Integer.MAX\_VALUE;

for (int i = 0 ; i < list.size() - 1; i++){

int candid = Math.abs(list.get(i+1) - list.get(i));

if (candid < diff){

diff = candid;

}

}

return diff;

}

private void preOrder(TreeNode root, List<Integer> list){

if (root == null){

return;

}

list.add(root.val);

preOrder(root.left, list);

preOrder(root.right, list);

}

}

Since this is a BST, the inorder traversal of its nodes results in a sorted list of values. Thus,

the minimum absolute difference must occur in any adjacently traversed nodes.

I use the global variable "prev" to keep track of each node's inorder predecessor.

Method 2:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

int ans = Integer.MAX\_VALUE;

TreeNode prev = null;

public int getMinimumDifference(TreeNode root) {

inOrder(root);

return ans;

}

private void inOrder(TreeNode root){

if (root == null){

return;

}

inOrder(root.left);

if (prev != null){

ans = Math.min(ans, Math.abs(root.val - prev.val));

}

prev = root;

inOrder(root.right);

}

}

Method 3:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

TreeNode prev = null;

public int getMinimumDifference(TreeNode root) {

int[] ans = new int[1];

ans[0] = Integer.MAX\_VALUE;

inOrder(root, ans);

return ans[0];

}

private void inOrder(TreeNode root, int[] ans){

if (root == null){

return;

}

inOrder(root.left, ans);

if (prev != null){

ans[0] = Math.min(ans[0], root.val - prev.val);

}

prev = root;

inOrder(root.right, ans);

}

}

Wrong answer below,

the answer below is the minimum difference between root and children

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int minDiffInBST(TreeNode root) {

int[] ans = new int[1];

ans[0] = Integer.MAX\_VALUE;

inOrder(root, null, ans);

return ans[0];

}

private void inOrder(TreeNode root, TreeNode prev, int[] ans){

if (root == null){

return;

}

inOrder(root.left, root, ans);

if (prev != null){

ans[0] = Math.min(ans[0], Math.abs(root.val - prev.val));

}

inOrder(root.right, root, ans);

}

}

Method 4: in order traversal Best solution

class Solution {

public int getMinimumDifference(TreeNode root) {

Stack<TreeNode> stack = new Stack<>();

TreeNode curr = root;

TreeNode prev = null;

int min = Integer.MAX\_VALUE;

while (curr != null || !stack.isEmpty()){

while (curr != null){

stack.push(curr);

curr = curr.left;

}

TreeNode node = stack.pop();

if (prev != null){

min = Math.min(min, node.val - prev.val);

}

prev = node;

curr = node.right;

}

return min;

}

}

531\_Lonely.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a picture consisting of black and white pixels, find the number of black lonely pixels.

The picture is represented by a 2D char array consisting of 'B' and 'W', which means black and white pixels respectively.

A black lonely pixel is character 'B' that located at a specific position where the same row and same column don't have any other

black pixels.

Example:

Input:

[['W', 'W', 'B'],

['W', 'B', 'W'],

['B', 'W', 'W']]

Output: 3

Explanation: All the three 'B's are black lonely pixels.

Note:

The range of width and height of the input 2D array is [1,500].

class Solution {

public int findLonelyPixel(char[][] picture) {

if (picture == null || picture.length == 0){

return 0;

}

int count = 0;

int m = picture.length;

int n = picture[0].length;

int[] rows = new int[m];

int[] cols = new int[n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (picture[i][j] == 'B'){

rows[i]++;

cols[j]++;

}

}

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (picture[i][j] == 'B' && rows[i] == 1 && cols[j] == 1){

count++;

}

}

}

return count;

}

}

532\_K-diffPairsArray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers and an integer k, you need to find the number of unique k-diff pairs in the array.

Here a k-diff pair is defined as an integer pair (i, j), where i and j are both numbers in the array and their

absolute difference is k.

Example 1:

Input: [3, 1, 4, 1, 5], k = 2

Output: 2

Explanation: There are two 2-diff pairs in the array, (1, 3) and (3, 5).

Although we have two 1s in the input, we should only return the number of unique pairs.

Example 2:

Input:[1, 2, 3, 4, 5], k = 1

Output: 4

Explanation: There are four 1-diff pairs in the array, (1, 2), (2, 3), (3, 4) and (4, 5).

Example 3:

Input: [1, 3, 1, 5, 4], k = 0

Output: 1

Explanation: There is one 0-diff pair in the array, (1, 1).

Note:

The pairs (i, j) and (j, i) count as the same pair.

The length of the array won't exceed 10,000.

All the integers in the given input belong to the range: [-1e7, 1e7].

Method 1:Two points

Time complexity: O(nlogn)

class Solution {

public int findPairs(int[] nums, int k) {

if (nums == null || nums.length < 2){

return 0;

}

Arrays.sort(nums);

int left = 0;

int right = 1;

int count = 0;

while (right < nums.length && left <= right){

if (left == right){

right++;

}

if (right < nums.length && nums[right] - nums[left] == k){

count++;

left++;

right++;

while (right < nums.length && nums[right] == nums[right - 1]){

right++;

}

while (left < right && nums[left] == nums[left - 1]){

left++;

}

}else if (right < nums.length && nums[right] - nums[left] > k){

left++;

}else if (right < nums.length && nums[right] - nums[left] < k) {

right++;

}

}

return count;

}

}

Method 2:HashMap

Time complexity: O(n)

class Solution {

public int findPairs(int[] nums, int k) {

if (nums == null || nums.length < 2 || k < 0 ){

return 0;

}

int count = 0;

Map<Integer, Integer> map = new HashMap<>();

for (int i : nums){

map.put(i, map.getOrDefault(i, 0) + 1);

}

for (Map.Entry<Integer, Integer> entry: map.entrySet()){

if (k == 0){

if (entry.getValue() > 1){

count++;

}

}else{

if (map.containsKey(entry.getKey() + k)){

count++;

}

}

}

return count;

}

}

535\_url.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Note: This is a companion problem to the System Design problem: Design TinyURL.

TinyURL is a URL shortening service where you enter a URL such as https://leetcode.com/problems/design-tinyurl and

it returns a short URL such as http://tinyurl.com/4e9iAk.

Design the encode and decode methods for the TinyURL service. There is no restriction on how your encode/decode

algorithm should work. You just need to ensure that a URL can be encoded to a tiny URL and the tiny URL can be

decoded to the original URL.

public class Codec {

Map<Integer, String> map = new HashMap<>();

// Encodes a URL to a shortened URL.

public String encode(String longUrl) {

int hash = longUrl.hashCode();

map.put(hash, longUrl);

return "http://tinyurl.com/" + hash;

}

// Decodes a shortened URL to its original URL.

public String decode(String shortUrl) {

int index = shortUrl.lastIndexOf("/");

return map.get(Integer.parseInt(shortUrl.substring(index+1)));

}

}

// Your Codec object will be instantiated and called as such:

// Codec codec = new Codec();

// codec.decode(codec.encode(url));

536\_ConstructBinaryTreefromString.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You need to construct a binary tree from a string consisting of parenthesis and integers.

The whole input represents a binary tree. It contains an integer followed by zero, one or two pairs of parenthesis. The integer represents the root's value and a pair of parenthesis contains a child binary tree with the same structure.

You always start to construct the left child node of the parent first if it exists.

Example:

Input: "4(2(3)(1))(6(5))"

Output: return the tree root node representing the following tree:

4

/ \

2 6

/ \ /

3 1 5

Note:

There will only be '(', ')', '-' and '0' ~ '9' in the input string.

An empty tree is represented by "" instead of "()".

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

Method: dfs - pre order traversal

The string has 3 parts : root (left)(right). However, (left) and (right) might be empty.

Actually, after we find the left part, we can break the loop and build right child using

the rest string directly if the rest is not empty.

class Solution {

public TreeNode str2tree(String s) {

if (s == null || s.length() == 0 || s.equals("")){

return null;

}

int index = s.indexOf('(');

int num = index == -1 ? Integer.parseInt(s) : Integer.parseInt(s.substring(0, index));

TreeNode root = new TreeNode(num);

if (index == -1){

return root;

}

int i = index;

int count = 0;

while (i < s.length()){

if (s.charAt(i) == '('){

count++;

}else if (s.charAt(i) == ')'){

count--;

}

if (count == 0){

root.left = str2tree(s.substring(index + 1, i));

break;

}

i++;

}

if (i < s.length() - 1){

root.right = str2tree(s.substring(i+2, s.length() - 1));

}

return root;

}

}

537\_ComplexNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings representing two complex numbers.

You need to return a string representing their multiplication. Note i2 = -1 according to the definition.

Example 1:

Input: "1+1i", "1+1i"

Output: "0+2i"

Explanation: (1 + i) \* (1 + i) = 1 + i2 + 2 \* i = 2i, and you need convert it to the form of 0+2i.

Example 2:

Input: "1+-1i", "1+-1i"

Output: "0+-2i"

Explanation: (1 - i) \* (1 - i) = 1 + i2 - 2 \* i = -2i, and you need convert it to the form of 0+-2i.

Note:

The input strings will not have extra blank.

The input strings will be given in the form of a+bi, where the integer a and b will both belong to the range of [-100, 100]. And the output should be also in this form.

class Solution {

public String complexNumberMultiply(String a, String b) {

int[] aNum = parse(a);

int[] bNum = parse(b);

String real = String.valueOf(aNum[0] \* bNum[0] - aNum[1] \* bNum[1]);

String imag = String.valueOf(aNum[0] \* bNum[1] + aNum[1] \* bNum[0]);

return real + "+" + imag +"i";

}

private int[] parse(String a){

int[] result = new int[2];

int index = 0;

for (int k = 0; k < a.length(); k++){

if (a.charAt(k) == '+'){

result[0] = Integer.parseInt(a.substring(index, k));

index = k + 1;

}

if (a.charAt(k) == 'i'){

result[1] = Integer.parseInt(a.substring(index, k));

}

}

return result;

}

}

class Solution {

public String complexNumberMultiply(String a, String b) {

int[] A = parse(a);

int[] B = parse(b);

StringBuilder sb = new StringBuilder();

int real = A[0] \* B[0] - A[1] \* B[1];

int imag = A[0] \* B[1] + A[1] \* B[0];

sb.append(real + "+" + imag + "i");

return sb.toString();

}

private int[] parse(String S){

int[] res = new int[2];

int i = 0;

int index = S.indexOf("+");

res[0] = Integer.parseInt(S.substring(0, index));

res[1] = Integer.parseInt(S.substring(index+1, S.length() - 1));

return res;

}

}

538\_ConvertBSTtoGreaterTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.

Example:

Input: The root of a Binary Search Tree like this:

5

/ \

2 13

Output: The root of a Greater Tree like this:

18

/ \

20 13

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode convertBST(TreeNode root) {

if (root == null){

return root;

}

int totalSum = reverseInorder(root, 0);

return root;

}

private int reverseInorder(TreeNode root, int sum){

if (root == null){

return sum;

}

int right = reverseInorder(root.right, sum);

root.val = right + root.val;

int left = reverseInorder(root.left, root.val);

return left;

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

int sum = 0;

public TreeNode convertBST(TreeNode root) {

dfs(root);

return root;

}

private void dfs(TreeNode root){

if (root == null){

return;

}

dfs(root.right);

sum += root.val;

root.val = sum;

dfs(root.left);

}

}

539\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of 24-hour clock time points in "Hour:Minutes" format, find the minimum minutes difference between

any two time points in the list.

Example 1:

Input: ["23:59","00:00"]

Output: 1

Note:

The number of time points in the given list is at least 2 and won't exceed 20000.

The input time is legal and ranges from 00:00 to 23:59.

class Solution {

public int findMinDifference(List<String> timePoints) {

if (timePoints.size() < 2){

return 0;

}

Collections.sort(timePoints, new Comparator<String>(){

public int compare (String a, String b){

String[] aTime = a.split(":");

String[] bTime = b.split(":");

if (Integer.parseInt(aTime[0]) == Integer.parseInt(bTime[0])){

return Integer.parseInt(aTime[1]) - Integer.parseInt(bTime[1]);

}

return Integer.parseInt(aTime[0]) - Integer.parseInt(bTime[0]);

}

});

timePoints.add(timePoints.get(0));

int max = 24 \* 60 ;

int diff = max;

for (int i = 0; i < timePoints.size() - 1; i++){

String[] a = timePoints.get(i+1).split(":");

String[] b = timePoints.get(i).split(":");

int aVal = Integer.parseInt(a[0]) \* 60 + Integer.parseInt(a[1]);

int bVal = Integer.parseInt(b[0]) \* 60 + Integer.parseInt(b[1]);

int candid = Math.abs(aVal - bVal);

if (candid > max / 2){

candid = max - candid;

}

diff = Math.min(diff, candid);

}

return diff;

}

}

54\_SpiralMatrix.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.

For example,

Given the following matrix:

[

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

]

You should return [1,2,3,6,9,8,7,4,5].

class Solution {

public List<Integer> spiralOrder(int[][] matrix) {

List<Integer> result = new ArrayList<>();

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return result;

}

int m = matrix.length;

int n = matrix[0].length;

int left = 0;

int right = n - 1;

int top = 0;

int bottom = m - 1;

while (top <= bottom && left <= right){

for (int i = left; i <= right; i++){

result.add(matrix[top][i]);

}

top++;

for (int i = top; i <= bottom; i++){

result.add(matrix[i][right]);

}

right--;

if (top > bottom || left > right){

break;

}

for (int i = right; i>= left; i--){

result.add(matrix[bottom][i]);

}

bottom--;

for (int i = bottom; i >= top; i--){

result.add(matrix[i][left]);

}

left++;

}

return result;

}

}

class Solution {

public List<Integer> spiralOrder(int[][] matrix) {

List<Integer> res = new ArrayList<>();

if (matrix == null || matrix.length == 0){

return res;

}

int m = matrix.length;

int n = matrix[0].length;

int top = 0;

int bottom = m - 1;

int left = 0;

int right = n - 1;

while (top <= bottom && left <= right){

for (int j = left; j <= right; j++){

res.add(matrix[top][j]);

}

top++;

if (top > bottom){

break;

}

for (int i = top; i <= bottom; i++){

res.add(matrix[i][right]);

}

right--;

if (left > right){

break;

}

for (int j = right; j >= left; j--){

res.add(matrix[bottom][j]);

}

bottom--;

if (top > bottom){

break;

}

for (int i = bottom; i >= top; i--){

res.add(matrix[i][left]);

}

left++;

}

return res;

}

}

540\_Single.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted array consisting of only integers where every element appears twice except for one element which appears once.

Find this single element that appears only once.

Example 1:

Input: [1,1,2,3,3,4,4,8,8]

Output: 2

Example 2:

Input: [3,3,7,7,10,11,11]

Output: 10

Note: Your solution should run in O(log n) time and O(1) space.

Method 1:

class Solution {

public int singleNonDuplicate(int[] nums) {

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] != nums[mid+1] && nums[mid] != nums[mid-1]){

return nums[mid];

}

if (isEven(start, end)){

if (nums[mid] == nums[mid-1]){

end = mid;

}else if (nums[mid] == nums[mid+1]){

start = mid;

}

}else{

if (nums[mid] == nums[mid-1]){

start = mid + 1;

}else if (nums[mid] == nums[mid+1]){

end = mid - 1;

}

}

}

return nums[start];

}

private boolean isEven (int start, int end){

int len = end - start + 1;

return ((len - 1) /2) % 2 == 0;

}

}

Method 2:

class Solution {

public int singleNonDuplicate(int[] nums) {

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] != nums[mid+1] && nums[mid] != nums[mid-1]){

return nums[mid];

}

if (mid % 2 == 0){

if (nums[mid] == nums[mid-1]){

end = mid;

}else if (nums[mid] == nums[mid+1]){

start = mid;

}

}else{

if (nums[mid] == nums[mid-1]){

start = mid + 1;

}else if (nums[mid] == nums[mid+1]){

end = mid - 1;

}

}

}

return nums[start];

}

}

Better solution:

class Solution {

public int singleNonDuplicate(int[] nums) {

int start = 0;

int end = nums.length - 1;

while (start < end){

int mid = start + (end - start) / 2;

int left = nums[mid-1];

int right = nums[mid+1];

if (nums[mid] != left && nums[mid] != right){

return nums[mid];

}

if (mid % 2 == 0){

if (nums[mid] == nums[mid-1]){

end = mid -1;

}else if (nums[mid] == nums[mid+1]){

start = mid + 1;

}

}else{

if (nums[mid] == nums[mid-1]){

start = mid + 1;

}else if (nums[mid] == nums[mid+1]){

end = mid - 1;

}

}

}

return nums[start];

}

}

Method 3

https://leetcode.com/problems/single-element-in-a-sorted-array/discuss/100754/Java-Binary-Search-short-(7l)-O(log(n))-w-explanations

541\_Reverse.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string and an integer k, you need to reverse the first k characters for every 2k

characters counting from the start of the string. If there are less than k characters left,

reverse all of them. If there are less than 2k but greater than or equal to k characters,

then reverse the first k characters and left the other as original.

Example:

Input: s = "abcdefg", k = 2

Output: "bacdfeg"

Restrictions:

The string consists of lower English letters only.

Length of the given string and k will in the range [1, 10000]

Method 1: Best solution

class Solution {

public String reverseStr(String s, int k) {

char[] charArray = s.toCharArray();

for (int start = 0; start < s.length(); start += 2\*k){

int i = start;

int j = Math.min(i + k - 1, s.length() - 1);

while (i < j){

char temp = charArray[i];

charArray[i] = charArray[j];

charArray[j] = temp;

i++;

j--;

}

}

return new String(charArray);

}

}

class Solution {

public String reverseStr(String s, int k) {

int n = s.length();

StringBuilder sb = new StringBuilder();

for (int j = 0; j < n; j += 2\*k){

StringBuilder temp = new StringBuilder();

for (int i = 0; i < k && i + j < n; i++){

temp.append(s.charAt(i+j));

}

sb.append(temp.reverse().toString());

for (int i = k; i < 2\*k && i + j < n; i++){

sb.append(s.charAt(i+j));

}

}

return sb.toString();

}

}

542\_Matrix.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a matrix consists of 0 and 1, find the distance of the nearest 0 for each cell.

The distance between two adjacent cells is 1.

Example 1:

Input:

0 0 0

0 1 0

0 0 0

Output:

0 0 0

0 1 0

0 0 0

Example 2:

Input:

0 0 0

0 1 0

1 1 1

Output:

0 0 0

0 1 0

1 2 1

Note:

The number of elements of the given matrix will not exceed 10,000.

There are at least one 0 in the given matrix.

The cells are adjacent in only four directions: up, down, left and right.

Method 1: BFS TLE

Time complexity: O(m^2\*n^2)

Space compleixity: O(m\*n)

class Solution {

public int[][] updateMatrix(int[][] matrix) {

int m = matrix.length;

int n = matrix[0].length;

int[][] res = new int[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (matrix[i][j] == 0){

res[i][j] = 0;

}else{

res[i][j] = bfs(matrix, i, j);

}

}

}

return res;

}

private int bfs(int[][] matrix, int i, int j){

int m = matrix.length;

int n = matrix[0].length;

boolean[][] visited = new boolean[m][n];

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

qx.offer(i);

qy.offer(j);

visited[i][j] = true;

int level = 0;

while (!qx.isEmpty()){

int size = qx.size();

level++;

for (int l = 0; l < size; l++){

int cx = qx.poll();

int cy = qy.poll();

for (int k = 0; k < dx.length; k++){

int x = cx + dx[k];

int y = cy + dy[k];

if (x >= 0 && x < m && y >= 0 && y < n){

if (matrix[x][y] == 0){

return level;

}

if (!visited[x][y]){

qx.offer(x);

qy.offer(y);

visited[x][y] = true;

}

}

}

}

}

return level;

}

}

Method 2: BFS

Time complexity: O(m\*n)

Space complexity: O(m\*n)

class Solution {

public int[][] updateMatrix(int[][] matrix) {

int m = matrix.length;

int n = matrix[0].length;

int[][] res = new int[m][n];

boolean[][] visited = new boolean[m][n];

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

res[i][j] = matrix[i][j];

if (matrix[i][j] == 0){

qx.offer(i);

qy.offer(j);

visited[i][j] = true;

}

}

}

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

for (int k = 0; k < dx.length; k++){

int x = cx + dx[k];

int y = cy + dy[k];

if (x >= 0 && x < m && y >= 0 && y < n && !visited[x][y]){

res[x][y] = res[cx][cy] + 1;

qx.offer(x);

qy.offer(y);

visited[x][y] = true;

}

}

}

return res;

}

}

Method 3: BFS

Time complexity: O(mn)

Space complexity: O(mn)

class Solution {

public int[][] updateMatrix(int[][] matrix) {

int m = matrix.length;

int n = matrix[0].length;

int[][] res = new int[m][n];

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

Queue<int[]> q = new LinkedList<>();

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (matrix[i][j] == 0){

q.offer(new int[]{i, j});

res[i][j] = 0;

}else{

res[i][j] = Integer.MAX\_VALUE;

}

}

}

while (!q.isEmpty()){

int[] cell = q.poll();

for (int k = 0; k < dx.length; k++){

int x = cell[0] + dx[k];

int y = cell[1] + dy[k];

if (x >= 0 && x < m && y >= 0 && y < n && res[x][y] > res[cell[0]][cell[1]]){

res[x][y] = res[cell[0]][cell[1]] + 1;

q.offer(new int[]{x, y});

}

}

}

return res;

}

}

Method 4: Best solution

class Solution {

public int[][] updateMatrix(int[][] matrix) {

if (matrix == null || matrix.length == 0){

return new int[0][0];

}

int m = matrix.length;

int n = matrix[0].length;

Queue<int[]> queue = new LinkedList<>();

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (matrix[i][j] == 0){

queue.offer(new int[]{i, j});

visited[i][j] = true;

}

}

}

int[][] res = new int[m][n];

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

int level = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

int[] curr = queue.poll();

int x = curr[0];

int y = curr[1];

res[x][y] = level;

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

visited[nx][ny] = true;

queue.offer(new int[]{nx, ny});

}

}

}

level++;

}

return res;

}

}

# 543\_Diameter.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a

binary tree is the length of the longest path between any two nodes in a tree. This path may or

may not pass through the root.

Example:

Given a binary tree

1

/ \

2 3

/ \

4 5

Return 3, which is the length of the path [4,2,1,3] or [5,2,1,3].

Note: The length of path between two nodes is represented by the number of edges between them.

https://leetcode.com/articles/diameter-of-binary-tree/

Method 1:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

private int ans;

public int diameterOfBinaryTree(TreeNode root) {

ans = 1;

depth(root);

return ans - 1;

}

private int depth(TreeNode root){

if (root == null){

return 0;

}

int L = depth(root.left);

int R = depth(root.right);

ans = Math.max(ans, L+R+1);

return Math.max(L, R) + 1;

}

}

Method 2: without using global variable

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int diameterOfBinaryTree(TreeNode root) {

int[] ans = new int[1];

depth(root, ans);

return ans[0] - 1 < 0 ? 0 : ans[0] - 1;

}

private int depth(TreeNode root, int[] ans){

if (root == null){

return 0;

}

int L = depth(root.left, ans);

int R = depth(root.right, ans);

ans[0] = Math.max(ans[0], L+R+1);

return Math.max(L, R) + 1;

}

}

Best: the same method as Binary Tree Maximum Path Sum, Longest Univalue Path

best solution:

If need find for n-ary tree, just find the longest and 2nd longest in all children, then similar as below

consider left as the longest,

and right as the 2nd longest

https://www.1point3acres.com/bbs/forum.php?mod=viewthread&tid=514267&extra=page%3D2%26filter%3Dsortid%26sortid%3D311%26searchoption%255B3046%255D%255Bvalue%255D%3D2%26searchoption%255B3046%255D%255Btype%255D%3Dradio%26searchoption%255B3109%255D%255Bvalue%255D%3D2%26searchoption%255B3109%255D%255Btype%255D%3Dradio%26orderby%3Ddateline&page=2

class Solution {

int max = Integer.MIN\_VALUE;

public int diameterOfBinaryTree(TreeNode root) {

if (root == null){

return 0;

}

maxDiaIncludeRoot(root);

return max;

}

private int maxDiaIncludeRoot(TreeNode root){

if (root == null){

return -1;

}

int left = maxDiaIncludeRoot(root.left);

int right = maxDiaIncludeRoot(root.right);

int leftMax = left + 1;

int rightMax = right + 1;

max = Math.max(max, leftMax + rightMax);

return Math.max(leftMax, rightMax);

}

}

class Solution {

public int diameterOfBinaryTree(TreeNode root) {

if (root == null){

return 0;

}

int[] ans = new int[1];

diameterInlcudeRoot(root, ans);

return ans[0];

}

private int diameterInlcudeRoot(TreeNode root, int[] ans){

if (root == null){

return 0;

}

int left = diameterInlcudeRoot(root.left, ans);

int right = diameterInlcudeRoot(root.right, ans);

ans[0] = Math.max(ans[0], left + right);

return Math.max(left, right) + 1;

}

}

# 544\_Output.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

During the NBA playoffs, we always arrange the rather strong team to play with the rather weak team, like make the rank 1 team play

with the rank nth team, which is a good strategy to make the contest more interesting. Now, you're given n teams, you need to output

their final contest matches in the form of a string.

The n teams are given in the form of positive integers from 1 to n, which represents their initial rank. (Rank 1 is the strongest

team and Rank n is the weakest team.) We'll use parentheses('(', ')') and commas(',') to represent the contest team pairing -

parentheses('(' , ')') for pairing and commas(',') for partition. During the pairing process in each round, you always need to

follow the strategy of making the rather strong one pair with the rather weak one.

Example 1:

Input: 2

Output: (1,2)

Explanation:

Initially, we have the team 1 and the team 2, placed like: 1,2.

Then we pair the team (1,2) together with '(', ')' and ',', which is the final answer.

Example 2:

Input: 4

Output: ((1,4),(2,3))

Explanation:

In the first round, we pair the team 1 and 4, the team 2 and 3 together, as we need to make the strong team and weak team together.

And we got (1,4),(2,3).

In the second round, the winners of (1,4) and (2,3) need to play again to generate the final winner, so you need to add the paratheses outside them.

And we got the final answer ((1,4),(2,3)).

Example 3:

Input: 8

Output: (((1,8),(4,5)),((2,7),(3,6)))

Explanation:

First round: (1,8),(2,7),(3,6),(4,5)

Second round: ((1,8),(4,5)),((2,7),(3,6))

Third round: (((1,8),(4,5)),((2,7),(3,6)))

Since the third round will generate the final winner, you need to output the answer (((1,8),(4,5)),((2,7),(3,6))).

Note:

The n is in range [2, 212].

We ensure that the input n can be converted into the form 2k, where k is a positive integer.

Method 1:

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public String findContestMatch(int n) {

String result = "";

List<String> list = new ArrayList<>();

for (int i = 0; i < n; i++){

list.add(String.valueOf(i+1));

}

while (n > 1){

for (int i = 0; i < n/2; i++){

String s = "(" + list.get(i) + "," + list.get(n-1-i) + ")";

list.set(i, s);

}

n /= 2;

}

return list.get(0);

}

}

class Solution {

public String findContestMatch(int n) {

String[] team = new String[n];

for (int i = 0; i < n; ++i)

team[i] = "" + i+1;

int k = n;

while (k > 1){

for (int j = 0; j < k/2; j++){

team[j] = "(" + team[j] + "," + team[k-1-j] + ")";

}

}

return team[0];

}

}

Method 2:

Let's try to solve the problem in linear time. We can treat this problem as two separate problems:

outputting the correct sequence of parentheses and commas, and outputting the correct team number.

With a little effort, one can be convinced that a linear time solution probably exists.

Algorithm

Let's focus on the parentheses first. We can use recursion to find the answer. For example, when N = 8, let R = log\_2(N) = 3

be the number of rounds. The parentheses and commas look like this:

(((x,x),(x,x)),((x,x),(x,x)))

But this is just recursively

"(" + (sequence for R = 2) + "," + (sequence for R = 2) + ")"

= "(" + "((x,x),(x,x))" + "," + "((x,x),(x,x))" + ")"

Now let's look at the team numbers. For N = 16, the team numbers are:

team = [1, 16, 8, 9, 4, 13, 5, 12, 2, 15, 7, 10, 3, 14, 6, 11]

One thing we might notice is that adjacent numbers sum to 17. More specifically, indices that are 0 and 1 (mod 2) sum to 17.

Also, indices 0 and 2 (mod 4) sum to 9, indices 0 and 4 (mod 8) sum to 5, and so on.

The pattern in general is: indices 0 and 2\*\*r (mod 2\*\*(r+1)) sum to N \* 2\*\*-r + 1.

If we want to find the next team[i], then the lowest bit of i will help determine it's lower neighbor. For example,

team[12] = team[0b1100] has lower bit w = 4 = 0b100, so 12 has lower neighbor 12 - w = 8, and

also those team numbers sum to N / w + 1.

class Solution {

int[] team;

int t;

StringBuilder ans;

public String findContestMatch(int n) {

team = new int[n];

t = 0;

ans = new StringBuilder();

write(n, Integer.numberOfTrailingZeros(n));

return ans.toString();

}

public void write(int n, int round) {

if (round == 0) {

int w = Integer.lowestOneBit(t);

team[t] = w > 0 ? n / w + 1 - team[t - w] : 1;

ans.append("" + team[t++]);

} else {

ans.append("(");

write(n, round - 1);

ans.append(",");

write(n, round - 1);

ans.append(")");

}

}

}

## 547\_FriendCircles.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are N students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature.

For example, if A is a direct friend of B, and B is a direct friend of C, then A is an indirect friend of C. And we

defined a friend circle is a group of students who are direct or indirect friends.

Given a N\*N matrix M representing the friend relationship between students in the class. If M[i][j] = 1, then the ith

and jth students are direct friends with each other, otherwise not. And you have to output the total number of friend

circles among all the students.

Example 1:

Input:

[[1,1,0],

[1,1,0],

[0,0,1]]

Output: 2

Explanation:The 0th and 1st students are direct friends, so they are in a friend circle.

The 2nd student himself is in a friend circle. So return 2.

Example 2:

Input:

[[1,1,0],

[1,1,1],

[0,1,1]]

Output: 1

Explanation:The 0th and 1st students are direct friends, the 1st and 2nd students are direct friends,

so the 0th and 2nd students are indirect friends. All of them are in the same friend circle, so return 1.

Note:

N is in range [1,200].

M[i][i] = 1 for all students.

If M[i][j] = 1, then M[j][i] = 1.

Method 1: BFS -- similar to connected components

class Solution {

public int findCircleNum(int[][] M) {

int count = 0;

for (int i = 0; i < M.length; i++){

if (M[i][i] == 1){

count++;

bfs(i, M);

}

}

return count;

}

private void bfs(int student, int[][] M){

Queue<Integer> queue = new LinkedList<>();

queue.offer(student);

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

int st = queue.poll();

M[st][st] = 2; //indicate the student is visited

for (int j = 0; j < M[0].length; j++){

if (M[st][j] == 1 && M[j][j] == 1){

queue.offer(j);

}

}

}

}

}

}

Method 2: DFS

class Solution {

public int findCircleNum(int[][] M) {

int[] visited = new int[M.length];

int count = 0;

for (int i = 0; i < M.length; i++){

if (visited[i] == 0){

count++;

dfs(M, visited, i);

}

}

return count;

}

private void dfs(int[][] M, int[] visited, int st){

for (int i = 0; i < M[0].length; i++){

if (M[st][i] == 1 && visited[i] == 0){

visited[i] = 1;

dfs(M, visited, i);

}

}

}

}

Method 3: Union Find Best solution

class Solution {

class UF {

int[] parent;

int[] size;

int count;

public UF (int N){

parent = new int[N];

size = new int[N];

count = N;

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (parent[x] == x){

return x;

}

return parent[x] = find(parent[x]);

}

public void union (int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

count--;

}

}

}

public int findCircleNum(int[][] M) {

int m = M.length;

int n = M[0].length;

UF uf = new UF(n);

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (i == j){

continue;

}

if (M[i][j] == 1){

uf.union(i, j);

}

}

}

return uf.count;

}

}

## 548\_Split.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array with n integers, you need to find if there are triplets (i, j, k) which satisfies following conditions:

0 < i, i + 1 < j, j + 1 < k < n - 1

Sum of subarrays (0, i - 1), (i + 1, j - 1), (j + 1, k - 1) and (k + 1, n - 1) should be equal.

where we define that subarray (L, R) represents a slice of the original array starting from the element indexed L to the element

indexed R.

Example:

Input: [1,2,1,2,1,2,1]

Output: True

Explanation:

i = 1, j = 3, k = 5.

sum(0, i - 1) = sum(0, 0) = 1

sum(i + 1, j - 1) = sum(2, 2) = 1

sum(j + 1, k - 1) = sum(4, 4) = 1

sum(k + 1, n - 1) = sum(6, 6) = 1

Note:

1 <= n <= 2000.

Elements in the given array will be in range [-1,000,000, 1,000,000].

Method 1:

Time complexity: O(n^2)

Here j is used for middle cut, i for left cut and k for right cut.

Iterate middle cuts and then find left cuts which divides the first half into two equal quarters,

store that quarter sums in the hashset. Then find right cuts which divides the second half into two equal

quarters and check if quarter sum is present in the hashset. If yes return true.

class Solution {

public boolean splitArray(int[] nums) {

int n = nums.length;

if (n < 7){

return false;

}

int[] sum = new int[n];

sum[0] = nums[0];

for (int i = 1; i < n; i++){

sum[i] = sum[i-1] + nums[i];

}

for (int j = 3; j < n - 3; j++){

Set<Integer> set = new HashSet<>();

for (int i = 1; i < j - 1; i++){

if (sum[i-1] == sum[j-1] - sum[i]){

set.add(sum[i-1]);

}

}

for (int k = j + 2; k < n -1; k++){

if (sum[n-1] - sum[k] == sum[k-1] - sum[j]){

if (set.contains(sum[n-1] - sum[k])){

return true;

}

}

}

}

return false;

}

}

class Solution {

public boolean splitArray(int[] nums) {

int n = nums.length;

if (n < 7){

return false;

}

int[] preSum = new int[n+1];

preSum[0] = 0;

for (int i = 1; i <= n; i++){

preSum[i] = preSum[i-1] + nums[i-1];

}

for (int j = 3; j < n - 3; j++){

Set<Integer> set = new HashSet<>();

for (int i = 1; i < j-1; i++){

if (preSum[j] - preSum[i+1] == preSum[i]){

set.add(preSum[i]);

}

}

for (int k = j + 1; k < n-2; k++){

if (preSum[n] - preSum[k+1] == preSum[k] - preSum[j+1]){

if (set.contains(preSum[n] - preSum[k+1]){

return true;

}

}

}

}

return false;

}

}

Methodd 2: DFS TLE

class Solution {

public boolean splitArray(int[] nums) {

int n = nums.length;

if (n < 7){

return false;

}

int total = 0;

for (int i : nums){

total += i;

}

int sum = 0;

int splitCount = 3;

for (int i = 0; i < n - (2\*splitCount-1); i ++){

sum += nums[i];

if (dfs(nums, i+1, sum, splitCount - 1, total - sum - nums[i+1])){

return true;

}

}

return false;

}

private boolean dfs(int[] nums, int start, int sum, int splitCount, int left){

if (splitCount == 0){

if (left == sum){

return true;

}

return false;

}

int sumSum = 0;

for (int i = start + 1; i < nums.length - (2\*splitCount-1); i++){

sumSum += nums[i];

if (sumSum == sum){

if (dfs(nums, i+1, sum, splitCount - 1, left - sum - nums[i+1])){

return true;

}

}

}

return false;

}

}

# 551\_Student.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a string representing an attendance record for a student. The record only contains the following three characters:

'A' : Absent.

'L' : Late.

'P' : Present.

A student could be rewarded if his attendance record doesn't contain more than one 'A' (absent) or more than two continuous 'L' (late).

You need to return whether the student could be rewarded according to his attendance record.

Example 1:

Input: "PPALLP"

Output: True

Example 2:

Input: "PPALLL"

Output: False

class Solution {

public boolean checkRecord(String s) {

int countA = 0;

int countL = 0;

int numL = 0;

for (int i = 0; i < s.length(); i++){

if (s.charAt(i) == 'A'){

countA++;

numL = 0;

}else if (s.charAt(i) == 'L'){

numL++;

if (numL > 2){

countL++;

}

}else{

numL = 0;

}

}

return countA <= 1 && countL == 0;

}

}

## 552\_Student.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer n, return the number of all possible attendance records with length n, which will be regarded as rewardable. The answer may be very large, return it after mod 109 + 7.

A student attendance record is a string that only contains the following three characters:

'A' : Absent.

'L' : Late.

'P' : Present.

A record is regarded as rewardable if it doesn't contain more than one 'A' (absent) or more than two continuous 'L' (late).

Example 1:

Input: n = 2

Output: 8

Explanation:

There are 8 records with length 2 will be regarded as rewardable:

"PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"

Only "AA" won't be regarded as rewardable owing to more than one absent times.

Note: The value of n won't exceed 100,000.

Best solution:

https://leetcode.com/problems/student-attendance-record-ii/discuss/101638/Simple-Java-O(n)-solution

class Solution {

public int checkRecord(int n) {

int M = (int) Math.pow(10, 9) + 7;

long[] P = new long[n+1];//ending with P and no A

long[] L = new long[n+1];//ending with L and no A

P[0] = 1;

P[1] = 1;

L[0] = 0;

L[1] = 1;

for (int i = 2; i <= n; i++){

P[i] = (P[i-1] + L[i-1]) % M;

L[i] = (P[i-1] + P[i-2]) % M;

}

long res = (P[n] + L[n]) % M;

//insert A into (n-1) length

for (int i = 0; i < n; i++){

long num = ((P[i] + L[i])%M \* (P[n-1-i] + L[n-1-i])%M)%M;

res = (res + num) % M;

}

return (int)res;

}

}

https://leetcode.com/problems/student-attendance-record-ii/discuss/101643/Share-my-O(n)-C++-DP-solution-with-thinking-process-and-explanation?page=2

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int checkRecord(int n) {

int M = (int) Math.pow(10, 9) + 7;

int[] A = new int[n+1];

int[] P = new int[n+1];

int[] L = new int[n+1];

P[1] = 1;

A[1] = 1;

L[1] = 1;

if (n == 1){

return A[1] + L[1] + P[1];

}

P[2] = 3;

A[2] = 2;

L[2] = 3;

if (n == 2){

return A[2] + L[2] + P[2];

}

P[3] = 8;

A[3] = 4;

L[3] = 7;

if (n == 3){

return A[3] + L[3] + P[3];

}

for (int i = 4; i <= n; i++){

P[i-1] %= M;

A[i-1] %= M;

L[i-1] %= M;

P[i] = ((P[i-1] + A[i-1])%M + L[i-1])%M;

L[i] = ((P[i-1] + A[i-1])%M + (P[i-2] + A[i-2])%M)%M;

A[i] = ((A[i-1] + A[i-2])%M + A[i-3])%M;

}

return (((A[n]%M) + (P[n]%M))%M + (L[n]%M))%M;

}

}

Finally, the recursive formula group becomes

P(n) = A(n - 1) + P(n - 1) + L(n - 1), n ≥ 2.

L(n) = A(n - 1) + P(n - 1) + A(n - 2) + P(n - 2), n ≥ 3.

A(n) = A(n - 1) + A(n - 2) + A(n - 3), n ≥ 4.

How to deduce

When n ≥ 4, the 3 formulas

A(n) = noAP(n - 1) + noAL(n - 1), n ≥ 2.

noAP(n) = noAP(n - 1) + noAL(n - 1), n ≥ 2.

noAL(n) = noAP(n - 1) + noAP(n - 2), n ≥ 3.

can be simplified to

A(n) = A(n - 1) + A(n - 2) + A(n - 3), n ≥ 4.

Answer:

A(n) = noAP(n - 1) + noAL(n - 1), n ≥ 2.

noAP(n) = noAP(n - 1) + noAL(n - 1), n ≥ 2.

which means

A(n) = noAP(n), n ≥ 2.

As

noAL(n) = noAP(n - 1) + noAP(n - 2), n ≥ 3.

Then

noAL(n) = A(n - 1) + A(n - 2), n ≥ 3.

As

A(n) = noAP(n - 1) + noAL(n - 1), n ≥ 2.

Then I get

A(n) = A(n - 1) + A(n - 2) + A(n - 3), n ≥ 4.

# 553\_OptimalDivision.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of positive integers, the adjacent integers will perform the float division. For example, [2,3,4] -> 2 / 3 / 4.

However, you can add any number of parenthesis at any position to change the priority of operations. You should find out how to add parenthesis to get the maximum result, and return the corresponding expression in string format. Your expression should NOT contain redundant parenthesis.

Example:

Input: [1000,100,10,2]

Output: "1000/(100/10/2)"

Explanation:

1000/(100/10/2) = 1000/((100/10)/2) = 200

However, the bold parenthesis in "1000/((100/10)/2)" are redundant,

since they don't influence the operation priority. So you should return "1000/(100/10/2)".

Other cases:

1000/(100/10)/2 = 50

1000/(100/(10/2)) = 50

1000/100/10/2 = 0.5

1000/100/(10/2) = 2

Note:

The length of the input array is [1, 10].

Elements in the given array will be in range [2, 1000].

There is only one optimal division for each test case.

X1/X2/X3/…/Xn will always be equal to (X1/X2) \* Y, no matter how you place parentheses. i.e no matter how you

place parentheses, X1 always goes to the numerator and X2 always goes to the denominator. Hence you just need to maximize Y.

And Y is maximized when it is equal to X3 \*…\*Xn. So the answer is always X1/(X2/X3/…/Xn) = (X1 \*X3 \*…\*Xn)/X2

class Solution {

public String optimalDivision(int[] nums) {

StringBuilder sb = new StringBuilder();

if (nums.length == 1){

return sb.append(String.valueOf(nums[0])).toString();

}

if (nums.length == 2){

return sb.append(String.valueOf(nums[0]) + "/" + String.valueOf(nums[1])).toString();

}

sb.append(String.valueOf(nums[0]));

sb.append("/(" + String.valueOf(nums[1]));

for (int i = 2; i < nums.length; i++){

sb.append("/" + String.valueOf(nums[i]));

}

sb.append(")");

return sb.toString();

}

}

## 554\_Brick.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a brick wall in front of you. The wall is rectangular and has several rows of bricks. The bricks have the same height but different width. You want to draw a vertical line from the top to the bottom and cross the least bricks.

The brick wall is represented by a list of rows. Each row is a list of integers representing the width of each brick in this row from left to right.

If your line go through the edge of a brick, then the brick is not considered as crossed. You need to find out how to draw the line to cross the least bricks and return the number of crossed bricks.

You cannot draw a line just along one of the two vertical edges of the wall, in which case the line will obviously cross no bricks.

Example:

Input:

[[1,2,2,1],

[3,1,2],

[1,3,2],

[2,4],

[3,1,2],

[1,3,1,1]]

Output: 2

Explanation:

class Solution {

public int leastBricks(List<List<Integer>> wall) {

Map<Integer, Integer> map = new HashMap<>();

int max = 0;

for (List<Integer> list : wall){

int preSum = 0;

for (int i = 0; i < list.size() - 1; i++){

preSum += list.get(i);

map.put(preSum, map.getOrDefault(preSum, 0) + 1);

max = Math.max(max, map.get(preSum));

}

}

return wall.size() - max;

}

}

class Solution {

public int leastBricks(List<List<Integer>> wall) {

int sum = 0;

Map<Integer, Integer> map = new HashMap<>();

int res = wall.size();

for (List<Integer> list : wall){

sum = 0;

for(int i = 0; i < list.size() - 1; i++){

sum += list.get(i);

map.put(sum, map.getOrDefault(sum, 0) + 1);

res = Math.min(res, wall.size() - map.get(sum));

}

}

return res;

}

}

## 555\_Split.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of strings, you could concatenate these strings together into a loop, where for each string you could choose to reverse

it or not. Among all the possible loops, you need to find the lexicographically biggest string after cutting the loop, which will

make the looped string into a regular one.

Specifically, to find the lexicographically biggest string, you need to experience two phases:

Concatenate all the strings into a loop, where you can reverse some strings or not and connect them in the same order as given.

Cut and make one breakpoint in any place of the loop, which will make the looped string into a regular one starting from the

character at the cutpoint.

And your job is to find the lexicographically biggest one among all the possible regular strings.

Example:

Input: "abc", "xyz"

Output: "zyxcba"

Explanation: You can get the looped string "-abcxyz-", "-abczyx-", "-cbaxyz-", "-cbazyx-",

where '-' represents the looped status.

The answer string came from the fourth looped one,

where you could cut from the middle character 'a' and get "zyxcba".

Note:

The input strings will only contain lowercase letters.

The total length of all the strings will not over 1,000.

Key idea:

1. find the bigger one of each str: normal or reverse

2. try every possibiity to for each str

Brute force:

Time complexity: O(n^2), n is the sum length of all lengths of strs

class Solution {

String res = "";

public String splitLoopedString(String[] strs) {

if (strs == null || strs.length == 0){

return "";

}

int n = strs.length;

//step 1: find the bigger one for each str

for (int i = 0; i < n; i++){

String strR = reverse(strs[i]);

if (strR.compareTo(strs[i]) > 0){

strs[i] = strR;

}

}

//step 2: check normal or reverse for each str

for (int i = 0; i < n; i++){

solve(strs, i, true);

solve(strs, i, false);

}

return res;

}

private String reverse(String str){

StringBuilder sb = new StringBuilder();

for (int i = str.length() - 1; i >= 0; i--){

sb.append(str.charAt(i));

}

return sb.toString();

}

private void solve(String[] strs, int i, boolean rev){

String str = rev ? reverse(strs[i]) : strs[i];

StringBuilder sb1 = new StringBuilder();

for (int j = 0; j < i; j++){

sb1.append(strs[j]);

}

StringBuilder sb2 = new StringBuilder();

for (int j = i + 1; j < strs.length; j++){

sb2.append(strs[j]);

}

for (int k = 0; k < str.length(); k++){

StringBuilder sb = new StringBuilder();

sb.append(str.substring(k));

sb.append(sb2);

sb.append(sb1);

sb.append(str.substring(0, k));

String newStr = sb.toString();

if (newStr.compareTo(res) > 0){

res = newStr;

}

}

}

}

## 556\_Next.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive 32-bit integer n, you need to find the smallest 32-bit integer which has exactly the same

digits existing in the integer n and is greater in value than n. If no such positive 32-bit integer exists,

you need to return -1.

Example 1:

Input: 12

Output: 21

Example 2:

Input: 21

Output: -1

The same as next permutation

class Solution {

public int nextGreaterElement(int n) {

//step 1: find the index of the first digit is smaller than previous one from rightmost digit

String s = Integer.toString(n);

char[] charArray = s.toCharArray();

int index1 = 0;

int i;

for (i = s.length() - 2; i >= 0; i--){

if (charArray[i] < charArray[i+1]){

index1 = i;

break;

}

}

if (i < 0){

return -1;

}

//step 2: find the first digit that is greater than the digit from the rightmost digit

int index2 = 0;

for (i = s.length() - 1; i >= 0; i--){

if (charArray[i] > s.charAt(index1)){

index2 = i;

break;

}

}

//step 3: swap index1 and index2

char temp = charArray[index1];

charArray[index1] = charArray[index2];

charArray[index2] = temp;

//step 4: reverse the order from index1 + 1 to rightmost

int start = index1 + 1;

int end = s.length() - 1;

while (start < end){

temp = charArray[start];

charArray[start] = charArray[end];

charArray[end] = temp;

start++;

end--;

}

long ans = Long.parseLong(String.valueOf(charArray));

if (ans > Integer.MAX\_VALUE){

return -1;

}

return (int) ans;

}

}

## 559\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a n-ary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

For example, given a 3-ary tree:

/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> children;

public Node() {}

public Node(int \_val,List<Node> \_children) {

val = \_val;

children = \_children;

}

};

\*/

class Solution {

public int maxDepth(Node root) {

return dfs(root);

}

private int dfs(Node root){

if (root == null){

return 0;

}

int max = 0;

for (Node node : root.children){

max = Math.max(max, dfs(node));

}

return max + 1;

}

}

class Solution {

public int maxDepth(Node root) {

if (root == null){

return 0;

}

int max = 0;

for (Node node : root.children){

max = Math.max(max, maxDepth(node));

}

return max + 1;

}

}

## 560\_SubarraySumEqualsK.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers and an integer k, you need to find the total number of continuous subarrays whose sum equals to k.

Example 1:

Input:nums = [1,1,1], k = 2

Output: 2

Note:

The length of the array is in range [1, 20,000].

The range of numbers in the array is [-1000, 1000] and the range of the integer k is [-1e7, 1e7].

This is the same as Question 930

https://github.com/optimisea/Leetcode/blob/master/Java/974\_Subarray.java

Method 1: Best solution

class Solution {

public int subarraySum(int[] nums, int k) {

int count = 0;

int preSum = 0;

Map<Integer, Integer> map = new HashMap<>();

map.put(0, 1);

for (int i = 0; i < nums.length; i++){

preSum += nums[i];

if (map.containsKey(preSum-k)){

count += map.get(preSum-k);

}

map.put(preSum, map.getOrDefault(preSum, 0) + 1);

}

return count;

}

}

Method 2:

Time complexity: O(n)

class Solution {

public int subarraySum(int[] nums, int k) {

int[] prefixSum = new int[nums.length + 1];

prefixSum[0] = 0;

for (int i = 1; i <= nums.length; i++){

prefixSum[i] = prefixSum[i-1] + nums[i-1];

}

Map<Integer, Integer> map = new HashMap<>();

int count = 0;

for (int i = 0; i <= nums.length; i++){

if (map.containsKey(prefixSum[i] - k)){

count += map.get(prefixSum[i] - k);

}

map.put(prefixSum[i], map.getOrDefault(prefixSum[i], 0) + 1);

}

return count;

}

}

## 561\_ArrayPartitionI.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of 2n integers, your task is to group these integers into n pairs of integer, say

(a1, b1), (a2, b2), ..., (an, bn) which makes sum of min(ai, bi) for all i from 1 to n as large as possible.

Example 1:

Input: [1,4,3,2]

Output: 4

Explanation: n is 2, and the maximum sum of pairs is 4 = min(1, 2) + min(3, 4).

Note:

n is a positive integer, which is in the range of [1, 10000].

All the integers in the array will be in the range of [-10000, 10000].

Method 1:

Time complexity: O(nlogn)

class Solution {

public int arrayPairSum(int[] nums) {

Arrays.sort(nums);

int sum = 0;

for (int i = 0 ; i < nums.length; i +=2){

sum += nums[i];

}

return sum;

}

}

## 562\_Longest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 01 matrix M, find the longest line of consecutive one in the matrix. The line could be horizontal, vertical, diagonal or

anti-diagonal.

Example:

Input:

[[0,1,1,0],

[0,1,1,0],

[0,0,0,1]]

Output: 3

Hint: The number of elements in the given matrix will not exceed 10,000.

class Solution {

public int longestLine(int[][] M) {

if (M == null || M.length == 0){

return 0;

}

int m = M.length;

int n = M[0].length;

int max = 0;

//horizontal

for (int i = 0; i < m; i++){

int count = 0;

for (int j = 0; j < n; j++){

if (M[i][j] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//vertical

for (int i = 0; i < n; i++){

int count = 0;

for (int j = 0; j < m; j++){

if (M[j][i] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//upper diagonal

for (int j = 0; j < n; j++){

int count = 0;

for (int x = 0, y = j; x < m && y < n; x++, y++){

if (M[x][y] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//lower diagonal

for (int i = 0; i < m; i++){

int count = 0;

for (int x = i, y = 0; x < m && y < n; x++, y++){

if (M[x][y] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//upper anti-diagonal

for (int j = 0; j < n; j++){

int count = 0;

for (int x = 0, y = n-1-j; x < m && y >= 0; x++, y--){

if (M[x][y] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//lower anti-diagonal

for (int i = 0; i < m; i++){

int count = 0;

for (int x = i, y = n-1; x < m && y >= 0; x++, y--){

if (M[x][y] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

return max;

}

}

class Solution {

public int longestLine(int[][] M) {

if (M == null || M.length == 0){

return 0;

}

int m = M.length;

int n = M[0].length;

int max = 0;

//horizontal

for (int i = 0; i < m; i++){

int count = 0;

for (int j = 0; j < n; j++){

if (M[i][j] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//vertical

for (int i = 0; i < n; i++){

int count = 0;

for (int j = 0; j < m; j++){

if (M[j][i] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//upper diagonal

for (int j = 0; j < n; j++){

int count = 0;

for (int x = 0, y = j; x < m && y < n; x++, y++){

if (M[x][y] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//lower diagonal

for (int i = 0; i < m; i++){

int count = 0;

for (int x = i, y = 0; x < m && y < n; x++, y++){

if (M[x][y] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//upper anti-diagonal

for (int j = n-1; j >= 0 ; j--){

int count = 0;

for (int x = 0, y = j; x < m && y >= 0; x++, y--){

if (M[x][y] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

//lower anti-diagonal

for (int i = m-1; i >= 0; i--){

int count = 0;

for (int x = i, y = n-1; x < m && y >= 0; x++, y--){

if (M[x][y] == 1){

count++;

max = Math.max(max, count);

}else{

count = 0;

}

}

}

return max;

}

}

## 563\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the tilt of the whole tree.

The tilt of a tree node is defined as the absolute difference between the sum of all left subtree node values and the sum of

all right subtree node values. Null node has tilt 0.

The tilt of the whole tree is defined as the sum of all nodes' tilt.

Example:

Input:

1

/ \

2 3

Output: 1

Explanation:

Tilt of node 2 : 0

Tilt of node 3 : 0

Tilt of node 1 : |2-3| = 1

Tilt of binary tree : 0 + 0 + 1 = 1

Note:

The sum of node values in any subtree won't exceed the range of 32-bit integer.

All the tilt values won't exceed the range of 32-bit integer.

Method 1:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

int tilt = 0;

public int findTilt(TreeNode root) {

sum(root);

return tilt;

}

private int sum(TreeNode root){

if (root == null){

return 0;

}

int left = sum(root.left);

int right = sum(root.right);

tilt += Math.abs(left - right);

return left + right + root.val;

}

}

Method 2: without global variable

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int findTilt(TreeNode root) {

int[] tilt = new int[1];

sum(root, tilt);

return tilt[0];

}

private int sum(TreeNode root, int[] tilt){

if (root == null){

return 0;

}

int left = sum(root.left, tilt);

int right = sum(root.right, tilt);

tilt[0] += Math.abs(left - right);

return left + right + root.val;

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

class Pair {

int sum;

int tilt;

public Pair (int sum, int tilt){

this.sum = sum;

this.tilt = tilt;

}

}

public int findTilt(TreeNode root) {

if (root == null){

return 0;

}

Pair p = dfs(root);

return p.tilt;

}

private Pair dfs(TreeNode root){

if (root == null){

return new Pair(0, 0);

}

Pair left = dfs(root.left);

Pair right = dfs(root.right);

int tilt = Math.abs(left.sum - right.sum) + left.tilt + right.tilt;

int sum = root.val + left.sum + right.sum;

return new Pair(sum, tilt);

}

}

## 565\_Array.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A zero-indexed array A of length N contains all integers from 0 to N-1. Find and return the longest length of set S,

where S[i] = {A[i], A[A[i]], A[A[A[i]]], ... } subjected to the rule below.

Suppose the first element in S starts with the selection of element A[i] of index = i, the next element in S should be A[A[i]],

and then A[A[A[i]]]… By that analogy, we stop adding right before a duplicate element occurs in S.

Example 1:

Input: A = [5,4,0,3,1,6,2]

Output: 4

Explanation:

A[0] = 5, A[1] = 4, A[2] = 0, A[3] = 3, A[4] = 1, A[5] = 6, A[6] = 2.

One of the longest S[K]:

S[0] = {A[0], A[5], A[6], A[2]} = {5, 6, 2, 0}

Note:

N is an integer within the range [1, 20,000].

The elements of A are all distinct.

Each element of A is an integer within the range [0, N-1].

Method 1: Brute force (TLE)

class Solution {

public int arrayNesting(int[] nums) {

Set<Integer> set = new HashSet<>();

int max = 0;

for (int i = 0; i < nums.length; i++){

while (!set.contains(nums[i])){

set.add(nums[i]);

i = nums[i];

}

max = Math.max(max, set.size());

set.clear();

}

return max;

}

}

class Solution {

public int arrayNesting(int[] nums) {

int n = nums.length;

int max = 0;

for (int i = 0; i < n; i++){

Set<Integer> set = new HashSet<>();

int index = i;

int count = 0;

while (!set.contains(nums[index])){

set.add(nums[index]);

count++;

index = nums[index];

}

max = Math.max(max, count);

}

return max;

}

}

class Solution {

public int arrayNesting(int[] nums) {

int n = nums.length;

int max = 0;

for (int i = 0; i < n; i++){

boolean[] visited = new boolean[n];

int index = i;

int count = 0;

while (!visited[nums[index]]){

visited[nums[index]] = true;

count++;

index = nums[index];

}

max = Math.max(max, count);

}

return max;

}

}

Method 2:

class Solution {

public int arrayNesting(int[] nums) {

int max = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] < 0){

continue;

}

int count = 1;

int val = nums[i];

while (Math.abs(val) != i){

count++;

val = nums[Math.abs(val)];

nums[Math.abs(val)] \*= -1;

}

max = Math.max(max, count);

}

return max;

}

}

## 566\_Reshape.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In MATLAB, there is a very useful function called 'reshape', which can reshape a matrix into a new one with different size but keep its original data.

You're given a matrix represented by a two-dimensional array, and two positive integers r and c representing the row number and column number of the wanted reshaped matrix, respectively.

The reshaped matrix need to be filled with all the elements of the original matrix in the same row-traversing order as they were.

If the 'reshape' operation with given parameters is possible and legal, output the new reshaped matrix; Otherwise, output the original matrix.

Example 1:

Input:

nums =

[[1,2],

[3,4]]

r = 1, c = 4

Output:

[[1,2,3,4]]

Explanation:

The row-traversing of nums is [1,2,3,4]. The new reshaped matrix is a 1 \* 4 matrix, fill it row by row by using the previous list.

Example 2:

Input:

nums =

[[1,2],

[3,4]]

r = 2, c = 4

Output:

[[1,2],

[3,4]]

Explanation:

There is no way to reshape a 2 \* 2 matrix to a 2 \* 4 matrix. So output the original matrix.

Note:

The height and width of the given matrix is in range [1, 100].

The given r and c are all positive.

Note that this implementation is different from Matlab implementation because Matlab implementation is column traversing order

Method 1:

Space complexity O(r\*c)

class Solution {

public int[][] matrixReshape(int[][] nums, int r, int c) {

if (nums == null || nums.length == 0){

return nums;

}

int rows = nums.length;

int cols = nums[0].length;

if (rows \* cols != r \* c){

return nums;

}

int[] count = new int[rows \* cols];

int k = 0;

for (int i = 0; i < rows; i++){

for (int j = 0; j < cols; j++){

count[k++] = nums[i][j];

}

}

int[][] result = new int[r][c];

k = 0;

for (int i = 0; i < r; i++){

for (int j = 0; j < c; j++){

result[i][j] = count[k++];

}

}

return result;

}

}

Method 2: Best solution

Space complexity: O(1)

class Solution {

public int[][] matrixReshape(int[][] nums, int r, int c) {

if (nums == null || nums.length == 0){

return nums;

}

int rows = nums.length;

int cols = nums[0].length;

if (rows \* cols != r \* c){

return nums;

}

int[][] result = new int[r][c];

for (int i = 0; i < r\*c; i++){

result[i/c][i%c] = nums[i/cols][i%cols];

}

return result;

}

}

## 567\_Permutation.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings s1 and s2, write a function to return true if s2 contains the permutation of s1.

In other words, one of the first string's permutations is the substring of the second string.

Example 1:

Input:s1 = "ab" s2 = "eidbaooo"

Output:True

Explanation: s2 contains one permutation of s1 ("ba").

Example 2:

Input:s1= "ab" s2 = "eidboaoo"

Output: False

Note:

The input strings only contain lower case letters.

The length of both given strings is in range [1, 10,000].

Method: slidng window, the same as 438. Find All Anagrams in a String and 76. Minimum Window Substring

class Solution {

public boolean checkInclusion(String s1, String s2) {

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i< s1.length(); i++){

char ch = s1.charAt(i);

map.put(ch, map.getOrDefault(ch, 0) + 1);

}

int count = map.size();

int start = 0;

int end = 0;

while (end < s2.length()){

char charEnd = s2.charAt(end);

if (map.containsKey(charEnd)){

map.put(charEnd, map.get(charEnd) - 1);

if (map.get(charEnd) == 0){

count--;

}

}

end++;

while (count == 0){

if (end - start == s1.length()){

return true;

}

char charStart = s2.charAt(start);

if (map.containsKey(charStart)){

if (map.get(charStart) == 0){

count++;

}

map.put(charStart, map.get(charStart) + 1);

}

start++;

}

}

return false;

}

}

class Solution {

public boolean checkInclusion(String s1, String s2) {

Map<Character, Integer> map = new HashMap<>();

for (char c : s1.toCharArray()){

map.put(c, map.getOrDefault(c, 0) + 1);

}

int start = 0;

int end = 0;

int count = map.size();

while (end < s2.length()){

char cEnd = s2.charAt(end);

map.put(cEnd, map.getOrDefault(cEnd, 0) - 1);

if (map.get(cEnd) == 0){

count--;

}

end++;

while (count == 0){

if (end - start == s1.length()){

return true;

}

char cStart = s2.charAt(start);

if (map.get(cStart) == 0){

count++;

}

map.put(cStart, map.getOrDefault(cStart, 0) + 1);

start++;

}

}

return false;

}

}

## 568\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LeetCode wants to give one of its best employees the option to travel among N cities to collect algorithm problems. But all work

and no play makes Jack a dull boy, you could take vacations in some particular cities and weeks. Your job is to schedule the

traveling to maximize the number of vacation days you could take, but there are certain rules and restrictions you need to follow.

Rules and restrictions:

You can only travel among N cities, represented by indexes from 0 to N-1. Initially, you are in the city indexed 0 on Monday.

The cities are connected by flights. The flights are represented as a N\*N matrix (not necessary symmetrical), called flights

representing the airline status from the city i to the city j. If there is no flight from the city i to the city j,

flights[i][j] = 0; Otherwise, flights[i][j] = 1. Also, flights[i][i] = 0 for all i. You totally have K weeks (each week has 7 days)

to travel. You can only take flights at most once per day and can only take flights on each week's Monday morning. Since flight time

is so short, we don't consider the impact of flight time. For each city, you can only have restricted vacation days in different

weeks, given an N\*K matrix called days representing this relationship. For the value of days[i][j], it represents the maximum days

you could take vacation in the city i in the week j.

You're given the flights matrix and days matrix, and you need to output the maximum vacation days you could take during K weeks.

Example 1:

Input:flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[1,3,1],[6,0,3],[3,3,3]]

Output: 12

Explanation:

Ans = 6 + 3 + 3 = 12.

One of the best strategies is:

1st week : fly from city 0 to city 1 on Monday, and play 6 days and work 1 day.

(Although you start at city 0, we could also fly to and start at other cities since it is Monday.)

2nd week : fly from city 1 to city 2 on Monday, and play 3 days and work 4 days.

3rd week : stay at city 2, and play 3 days and work 4 days.

Example 2:

Input:flights = [[0,0,0],[0,0,0],[0,0,0]], days = [[1,1,1],[7,7,7],[7,7,7]]

Output: 3

Explanation:

Ans = 1 + 1 + 1 = 3.

Since there is no flights enable you to move to another city, you have to stay at city 0 for the whole 3 weeks.

For each week, you only have one day to play and six days to work.

So the maximum number of vacation days is 3.

Example 3:

Input:flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[7,0,0],[0,7,0],[0,0,7]]

Output: 21

Explanation:

Ans = 7 + 7 + 7 = 21

One of the best strategies is:

1st week : stay at city 0, and play 7 days.

2nd week : fly from city 0 to city 1 on Monday, and play 7 days.

3rd week : fly from city 1 to city 2 on Monday, and play 7 days.

Note:

N and K are positive integers, which are in the range of [1, 100].

In the matrix flights, all the values are integers in the range of [0, 1].

In the matrix days, all the values are integers in the range [0, 7].

You could stay at a city beyond the number of vacation days, but you should work on the extra days, which won't be counted as

vacation days.

If you fly from the city A to the city B and take the vacation on that day, the deduction towards vacation days will count

towards the vacation days of city B in that week.

We don't consider the impact of flight hours towards the calculation of vacation days.

Method 1:

Time complexity: O(N^2\*K)

Space compleixty: O(N\*K)

dp[i][j] represents the max vacation days at index i city and at index j week.

class Solution {

public int maxVacationDays(int[][] flights, int[][] days) {

int N = days.length;

int K = days[0].length;

int[][] dp = new int[N][K];

dp[0][0] = days[0][0];

for (int i = 1 ; i < N; i++){

if (flights[0][i] == 0){

dp[i][0] = Integer.MIN\_VALUE; // if can't go through, set to Integer.MIN\_VALUE instead of 0

}else{

dp[i][0] = days[i][0];

}

}

for (int j = 1; j < K; j++){

for (int i = 0; i < N; i++){

dp[i][j] = Integer.MIN\_VALUE;

for (int k = 0; k < N; k++){

if (i == k || flights[k][i] == 1){

dp[i][j] = Math.max(dp[i][j], dp[k][j-1] + days[i][j]) ;

}

}

}

}

int res = 0;

for (int i = 0; i < N; i++){

res = Math.max(res, dp[i][K-1]);

}

return res;

}

}

Method 2: because j only replies on j-1, so we can compress one dimension

Time complexity: O(N^2\*K)

Space complexity: O(N)

class Solution {

public int maxVacationDays(int[][] flights, int[][] days) {

int N = days.length;

int K = days[0].length;

int[][] dp = new int[N][2];

for (int i = 0 ; i < N; i++){

Arrays.fill(dp[i], Integer.MIN\_VALUE); // if can't go through, set to Integer.MIN\_VALUE instead of 0

if (i == 0 || flights[0][i] == 1){

dp[i][0] = days[i][0];

}

}

for (int j = 1; j < K; j++){

for (int i = 0; i < N; i++){

for (int k = 0; k < N; k++){

if (i == k || flights[k][i] == 1){

dp[i][j%2] = Math.max(dp[i][j%2], dp[k][(j-1)%2] + days[i][j]) ;

}

}

}

}

int res = 0;

for (int i = 0; i < N; i++){

res = Math.max(res, dp[i][(K-1)%2]);

}

return res;

}

}

## 57\_ThreeSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all

unique triplets in the array which gives the sum of zero.

Notice

Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c)

The solution set must not contain duplicate triplets.

Have you met this question in a real interview? Yes

Example

For example, given array S = {-1 0 1 2 -1 -4}, A solution set is:

(-1, 0, 1)

(-1, -1, 2)

public class Solution {

/\*

\* @param numbers: Give an array numbers of n integer

\* @return: Find all unique triplets in the array which gives the sum of zero.

\*/

public List<List<Integer>> threeSum(int[] numbers) {

List<List<Integer>> result = new ArrayList<>();

Arrays.sort(numbers);

for(int i = 0; i < numbers.length - 2; i++){

if (i > 0 && numbers[i] == numbers[i - 1]){

continue;

}

int target = -numbers[i];

int start = i + 1;

int end = numbers.length - 1;

twoSum(result, numbers, target, start, end);

}

return result;

}

private void twoSum(List<List<Integer>> result,

int[] numbers, int target, int start, int end){

while (start < end){

List<Integer> set = new ArrayList<>();

if (numbers[start] + numbers[end] == target){

set.add(-target);

set.add(numbers[start]);

set.add(numbers[end]);

result.add(set);

start++;

end--;

while (start < end && numbers[start] == numbers[start - 1]){

start++;

}

while (start < end && numbers[end] == numbers[end + 1]){

end--;

}

}else if (numbers[start] + numbers[end] > target){

end--;

}else{

start++;

}

}

}

}

## 572\_SubtreeofAnotherTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two non-empty binary trees s and t, check whether tree t has exactly the same structure and node values with

a subtree of s. A subtree of s is a tree consists of a node in s and all of this node's descendants.

The tree s could also be considered as a subtree of itself.

Example 1:

Given tree s:

3

/ \

4 5

/ \

1 2

Given tree t:

4

/ \

1 2

Return true, because t has the same structure and node values with a subtree of s.

Example 2:

Given tree s:

3

/ \

4 5

/ \

1 2

/

0

Given tree t:

4

/ \

1 2

Return false.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isSubtree(TreeNode s, TreeNode t) {

if (t == null){

return true;

}

if (s == null){

return false;

}

return isSameTree(s, t) || isSubtree(s.left, t) || isSubtree(s.right, t);

}

private boolean isSameTree(TreeNode s, TreeNode t){

if (s == null && t == null){

return true;

}

if (s != null && t == null){

return false;

}

if (s == null && t != null){

return false;

}

if (s.val != t.val){

return false;

}

return isSameTree(s.left, t.left) && isSameTree(s.right, t.right);

}

}

class Solution {

public boolean isSubtree(TreeNode s, TreeNode t) {

if (s == null && t == null){

return true;

}

if (s == null && t != null){

return false;

}

if (s != null && t == null){

return false;

}

if (s.val != t.val){

return isSubtree(s.left, t) || isSubtree(s.right, t);

}

return isSame(s, t) || isSubtree(s.left, t) || isSubtree(s.right, t);

}

private boolean isSame(TreeNode s, TreeNode t){

if (s == null && t == null){

return true;

}

if (s == null && t != null){

return false;

}

if (s != null && t == null){

return false;

}

if (s.val != t.val){

return false;

}

return isSame(s.left, t.left) && isSame(s.right, t.right);

}

}

## 575\_Distribute.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array with even length, where different numbers in this array represent different kinds of candies. Each number means one candy of the corresponding kind. You need to distribute these candies equally in number to brother and sister. Return the maximum number of kinds of candies the sister could gain.

Example 1:

Input: candies = [1,1,2,2,3,3]

Output: 3

Explanation:

There are three different kinds of candies (1, 2 and 3), and two candies for each kind.

Optimal distribution: The sister has candies [1,2,3] and the brother has candies [1,2,3], too.

The sister has three different kinds of candies.

Example 2:

Input: candies = [1,1,2,3]

Output: 2

Explanation: For example, the sister has candies [2,3] and the brother has candies [1,1].

The sister has two different kinds of candies, the brother has only one kind of candies.

Note:

The length of the given array is in range [2, 10,000], and will be even.

The number in given array is in range [-100,000, 100,000].

Method 1:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int distributeCandies(int[] candies) {

Set<Integer> set = new HashSet<>();

for (int candy : candies){

if (!set.contains(candy)){

set.add(candy);

}

}

return set.size() < candies.length / 2 ? set.size() : candies.length / 2;

}

}

class Solution {

public int distributeCandies(int[] candies) {

Set<Integer> set = new HashSet<>();

for (int c : candies){

set.add(c);

}

return Math.min(candies.length/2, set.size());

}

}

Method 2:

Time complexity: O(nlogn)

Space complexity: O(1)

class Solution {

public int distributeCandies(int[] candies) {

Arrays.sort(candies);

int count = 1;

for (int i = 1; i < candies.length; i++){

if (candies[i] != candies[i-1]){

count++;

}

}

return Math.min(count, candies.length / 2);

}

}

## 575\_ExpressionExpand.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an expression s includes numbers, letters and brackets. Number represents the number of

repetitions inside the brackets(can be a string or another expression)．Please expand expression to be a string.

Have you met this question in a real interview?

Example

s = abc3[a] return abcaaa

s = 3[abc] return abcabcabc

s = 4[ac]dy, return acacacacdy

s = 3[2[ad]3[pf]]xyz, return adadpfpfpfadadpfpfpfadadpfpfpfxyz

Challenge

Can you do it without recursion?

Method 1: Stack O(n)

public class Solution {

/\*\*

\* @param s: an expression includes numbers, letters and brackets

\* @return: a string

\*/

public String expressionExpand(String s) {

Stack<Object> stack = new Stack<>();

int num = 0;

for (char c : s.toCharArray()){

if (Character.isDigit(c)){

num = num \* 10 + c - '0';

}else if (c == '['){

stack.push(Integer.valueOf(num));

num = 0;

}else if (c == ']'){

String str = popStack(stack);

Integer count = (Integer) stack.pop();

for (int i = 0; i < count; i++){

stack.push(str);

}

}else{

stack.push(String.valueOf(c));

}

}

return popStack(stack);

}

private String popStack(Stack<Object> stack){

StringBuilder sb = new StringBuilder();

while (!stack.isEmpty() && (stack.peek() instanceof String)){

sb.insert(0, (String) stack.pop());

}

return sb.toString();

}

}

Method 2: DFS O(n^2)

## 576\_Out.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is an m by n grid with a ball. Given the start coordinate (i,j) of the ball, you can move the ball to adjacent cell or cross

the grid boundary in four directions (up, down, left, right). However, you can at most move N times. Find out the number of paths

to move the ball out of grid boundary. The answer may be very large, return it after mod 109 + 7.

Example 1:

Input:m = 2, n = 2, N = 2, i = 0, j = 0

Output: 6

Explanation:

Example 2:

Input:m = 1, n = 3, N = 3, i = 0, j = 1

Output: 12

Explanation:

Note:

Once you move the ball out of boundary, you cannot move it back.

The length and height of the grid is in range [1,50].

N is in range [0,50].

https://leetcode.com/problems/out-of-boundary-paths/solution/

Method 1: Brute Force (TLE)

Time complexity : O(4^n) Size of recursion tree will be 4^n. Here, n refers to the number of moves allowed.

Space complexity : O(n). The depth of the recursion tree can go upto n.

class Solution {

public int findPaths(int m, int n, int N, int i, int j) {

if (i == m || j == n || i < 0 || j < 0){

return 1;

}

if (N == 0){

return 0;

}

return findPaths(m, n, N-1, i-1, j) + findPaths(m, n, N-1, i+1, j) + findPaths(m, n, N-1, i, j-1) + findPaths(m, n, N-1, i, j+1);

}

}

Method 2: Memo

Time complexity : O(m\*n\*N)O(m∗n∗N). We need to fill the memomemo array once with dimensions mmxnnxNN. Here, mm, nn refer to the number of rows and columns of the given grid respectively. NN refers to the total number of allowed moves.

Space complexity : O(m\*n\*N)O(m∗n∗N). memomemo array of size m\*n\*Nm∗n∗N is used.

class Solution {

int M=1000000007;

public int findPaths(int m, int n, int N, int i, int j) {

int[][][] memo = new int[m][n][N+1];

for (int ii = 0; ii < m; ii++){

for (int jj = 0; jj < n; jj++){

Arrays.fill(memo[ii][jj], -1);

}

}

return dfs(m, n, N, i, j, memo);

}

private int dfs(int m, int n, int N, int i, int j, int[][][] memo){

if (i == m || j == n || i < 0 || j < 0){

return 1;

}

if (N == 0){

return 0;

}

if (memo[i][j][N] >= 0){

return memo[i][j][N];

}

// memo[i][j][N]=((dfs(m,n,N-1,i-1,j,memo)+dfs(m,n,N-1,i+1,j,memo))%M+(dfs(m,n,N-1,i,j-1,memo)+dfs(m,n,N-1,i,j+1,memo))%M)%M;

memo[i][j][N] = ((dfs(m, n, N-1, i-1, j, memo)%M + dfs(m, n, N-1, i+1, j, memo)%M)%M + (dfs(m, n, N-1, i, j-1, memo)%M

+ dfs(m, n, N-1, i, j+1, memo)%M)%M)%M;

return memo[i][j][N];

}

}

Method 3: Dynamic Programming Best solution Reduce space dimension

DP[i][j][k] stands for how many possible ways to walk into cell j,k in step i, DP[i][j][k] only depends on DP[i - 1][j][k],

so we can compress 3 dimensional dp array to 2 dimensional.

https://leetcode.com/problems/out-of-boundary-paths/discuss/102967/Java-Solution-DP-with-space-compression

Time complexity : O(N\*m\*n). We need to fill the dp array with dimensions mn N times.

Here mn refers to the size of the grid and N refers to the number of moves available.

Space complexity : O(m\*n). dp and temp array of size mn are used.

dp[i][j] refers to the number of ways the position corresponding to the indices (i,j)

can be reached given some particular number of moves.

dp[i][j]=dp[i−1][j]+dp[i+1][j]+dp[i][j−1]+dp[i][j+1]

initial position, dp[i][j] = 1 since there is only one way to reach i, j which is zero move

class Solution {

public int findPaths(int m, int n, int N, int i, int j) {

int MOD = (int)Math.pow(10, 9) + 7;

int[][] dp = new int[m][n];//record the number of ways to reach to row, col index

dp[i][j] = 1; //reach to i, j in zero move, so the number of ways is 1

int count = 0;

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

for (int move = 1; move <= N; move++){

int[][] temp = new int[m][n];

for (int r = 0; r < m; r++){

for (int c = 0; c < n; c++){

for (int k = 0; k < dx.length; k++){

int nx = r + dx[k];

int ny = c + dy[k];

if (nx < 0 || nx >= m || ny < 0 || ny >= n){

count = (count + dp[r][c]) % MOD;

}else{

temp[nx][ny] = (temp[nx][ny] + dp[r][c]) % MOD;

}

}

}

}

dp = temp;

}

return count;

}

}

Similar as Leetcode 688

https://github.com/optimisea/Leetcode/edit/master/Java/688\_KnightProbabilityinChessboard.java

class Solution {

public int findPaths(int m, int n, int N, int i, int j) {

int mod = (int) Math.pow(10, 9) + 7;

int[][] dp = new int[m][n];

dp[i][j] = 1;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

int count = 0;

for (int k = 1; k <= N; k++){

int[][] temp = new int[m][n];

for (int r = 0; r < m; r++){

for (int c = 0; c < n; c++){

for (int[] dir : dirs){

int nx = r + dir[0];

int ny = c + dir[1];

if (nx < 0 || nx == m || ny < 0 || ny == n){

count = (count + dp[r][c]) % mod;

}else{

temp[nx][ny] = (temp[nx][ny] + dp[r][c]) % mod;

}

}

}

}

dp = temp;

}

return count;

}

}

## 581\_Shortest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array, you need to find one continuous subarray that if you only sort this subarray in ascending order,

then the whole array will be sorted in ascending order, too.

You need to find the shortest such subarray and output its length.

https://leetcode.com/articles/shortest-unsorted-continous-subarray/

Example 1:

Input: [2, 6, 4, 8, 10, 9, 15]

Output: 5

Explanation: You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole array sorted in ascending order.

Note:

Then length of the input array is in range [1, 10,000].

The input array may contain duplicates, so ascending order here means <=.

Method 1: select sorting idea TLE

Time complexity: O(n^2)

Space complexity: O(1)

class Solution {

public int findUnsortedSubarray(int[] nums) {

int left = nums.length;

int right = 0;

for (int i = 0; i < nums.length; i++){

for (int j = i + 1; j < nums.length; j++){

if (nums[j] < nums[i]){

left = Math.min(left, i);

right = Math.max(right, j);

}

}

}

return right - left < 0 ? 0 : right - left + 1;

}

}

Method 2:

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public int findUnsortedSubarray(int[] nums) {

int[] numsClone = nums.clone();

Arrays.sort(nums);

int left = nums.length - 1;

int right = 0;

for (int i = 0; i < nums.length; i++){

if (nums[i] != numsClone[i]){

left = Math.min(left, i);

right = Math.max(right, i);

}

}

return right - left <= 0 ? 0 : right - left + 1;

}

}

Method 3: use stack to reduce time complexity and select sorting idea

Time complexity: O(n) two traverses

Space complexity: O(n)

class Solution {

public int findUnsortedSubarray(int[] nums) {

Stack<Integer> stack = new Stack<>();

int left = nums.length - 1;

int right = 0;

for (int i = 0; i < nums.length; i++){

while (!stack.isEmpty() && nums[i] < nums[stack.peek()]){

left = Math.min(left, stack.pop());

}

stack.push(i);

}

stack.clear();

for (int i = nums.length - 1; i >= 0; i--){

while (!stack.isEmpty() && nums[i] > nums[stack.peek()]){

right = Math.max(right, stack.pop());

}

stack.push(i);

}

return right - left <= 0 ? 0 : right - left + 1;

}

}

Method 4: use one traverse to find min within the unsorted range and use another traverse to find the pivot index

do the same thing from right to left

Time complexity: O(n) four traverses

Space complexity: O(1)

class Solution {

public int findUnsortedSubarray(int[] nums) {

int max = Integer.MIN\_VALUE;

int min = Integer.MAX\_VALUE;

boolean flag = false;

for (int i = 1; i < nums.length; i++){

if (nums[i] < nums[i-1]){

flag = true;

}

if (flag){

min = Math.min(min, nums[i]);

}

}

flag = false;

for (int i = nums.length - 2; i >= 0; i--){

if (nums[i] > nums[i+1]){

flag = true;

}

if (flag){

max = Math.max(max, nums[i]);

}

}

int left = 0;

while (left < nums.length){

if (nums[left] > min){

break;

}

left++;

}

int right = nums.length - 1;

while (right >= 0){

if (nums[right] < max){

break;

}

right--;

}

return right - left <= 0 ? 0 : right - left + 1;

}

}

Method 5: Better solution

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public int findUnsortedSubarray(int[] nums) {

int[] copy = Arrays.copyOf(nums, nums.length);

Arrays.sort(copy);

int i = 0;

int j = nums.length - 1;

while (i < j && nums[i] == copy[i]){

i++;

}

while (i < j && nums[j] == copy[j]){

j--;

}

return i == j ? 0 : j - i + 1;

}

}

Method 6: Best solution

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public int findUnsortedSubarray(int[] nums) {

if (nums == null || nums.length == 0 || nums.length == 1){

return 0;

}

int left = 0;

int right = nums.length - 1;

while (left < right && nums[left+1] >= nums[left]){

left++;

}

if (left == right){

return 0;

}

while (left < right && nums[right-1] <= nums[right]){

right--;

}

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;

for (int i = left; i <= right; i++){

min = Math.min(min, nums[i]);

max = Math.max(max, nums[i]);

}

while (left >= 0 && nums[left] > min){

left--;

}

while (right < nums.length && nums[right] < max){

right++;

}

return right - left - 1;

}

}

## 582\_Kill.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n processes, each process has a unique PID (process id) and its PPID (parent process id).

Each process only has one parent process, but may have one or more children processes. This is just like a tree structure.

Only one process has PPID that is 0, which means this process has no parent process. All the PIDs will be distinct positive integers.

We use two list of integers to represent a list of processes, where the first list contains PID for each process and the second

list contains the corresponding PPID.

Now given the two lists, and a PID representing a process you want to kill, return a list of PIDs of processes that will be

killed in the end. You should assume that when a process is killed, all its children processes will be killed. No order

is required for the final answer.

Example 1:

Input:

pid = [1, 3, 10, 5]

ppid = [3, 0, 5, 3]

kill = 5

Output: [5,10]

Explanation:

3

/ \

1 5

/

10

Kill 5 will also kill 10.

Note:

The given kill id is guaranteed to be one of the given PIDs.

n >= 1.

Method 1: HashMap + BFS

class Solution {

public List<Integer> killProcess(List<Integer> pid, List<Integer> ppid, int kill) {

List<Integer> res = new ArrayList<>();

int n = ppid.size();

Map<Integer, List<Integer>> map = new HashMap<>();

for (int i = 0; i < n; i++){

if (!map.containsKey(ppid.get(i))){

map.put(ppid.get(i), new ArrayList<>());

}

map.get(ppid.get(i)).add(i);

}

Queue<Integer> queue = new LinkedList<>();

queue.offer(kill);

while (!queue.isEmpty()){

int id = queue.poll();

res.add(id);

if (map.containsKey(id)){

List<Integer> list = map.get(id);

for (int i : list){

queue.offer(pid.get(i));

}

}

}

return res;

}

}

Method 2: HashMap + DFS

class Solution {

public List<Integer> killProcess(List<Integer> pid, List<Integer> ppid, int kill) {

List<Integer> res = new ArrayList<>();

int n = ppid.size();

Map<Integer, List<Integer>> map = new HashMap<>();

for (int i = 0; i < n; i++){

if (!map.containsKey(ppid.get(i))){

map.put(ppid.get(i), new ArrayList<>());

}

map.get(ppid.get(i)).add(i);

}

dfs(pid, kill, res, map);

return res;

}

private void dfs(List<Integer> pid, int kill, List<Integer> res, Map<Integer, List<Integer>> map){

res.add(kill);

if (map.containsKey(kill)){

List<Integer> list = map.get(kill);

for (int i : list){

dfs(pid, pid.get(i), res, map);

}

}

}

}

## 583\_Delete.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two words word1 and word2, find the minimum number of steps required to make word1 and word2 the same,

where in each step you can delete one character in either string.

Example 1:

Input: "sea", "eat"

Output: 2

Explanation: You need one step to make "sea" to "ea" and another step to make "eat" to "ea".

Note:

The length of given words won't exceed 500.

Characters in given words can only be lower-case letters.

The same as Edit Distance except that there is no replace function

class Solution {

public int minDistance(String word1, String word2) {

int m = word1.length();

int n = word2.length();

int[][] dp = new int[m+1][n+1];

for (int i = 0; i <= m; i++){

dp[i][0] = i;

}

for (int j = 0; j <= n; j++){

dp[0][j] = j;

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (word1.charAt(i-1) == word2.charAt(j-1)){

dp[i][j] = dp[i-1][j-1];

}else{

dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1]) + 1;

}

}

}

return dp[m][n];

}

}

## 588\_Design.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design an in-memory file system to simulate the following functions:

ls: Given a path in string format. If it is a file path, return a list that only contains this file's name. If it is a directory path,

return the list of file and directory names in this directory. Your output (file and directory names together) should in lexicographic

order.

mkdir: Given a directory path that does not exist, you should make a new directory according to the path. If the middle directories

in the path don't exist either, you should create them as well. This function has void return type.

addContentToFile: Given a file path and file content in string format. If the file doesn't exist, you need to create that file

containing given content. If the file already exists, you need to append given content to original content. This function has void

return type.

readContentFromFile: Given a file path, return its content in string format.

Example:

Input:

["FileSystem","ls","mkdir","addContentToFile","ls","readContentFromFile"]

[[],["/"],["/a/b/c"],["/a/b/c/d","hello"],["/"],["/a/b/c/d"]]

Output:

[null,[],null,null,["a"],"hello"]

Explanation:

filesystem

Note:

You can assume all file or directory paths are absolute paths which begin with / and do not end with / except that the path is

just "/".

You can assume that all operations will be passed valid parameters and users will not attempt to retrieve file content or

list a directory or file that does not exist.

You can assume that all directory names and file names only contain lower-case letters, and same names won't exist in the same

directory.

Similar as Trie Node

note that for split function

"/".split == > [];

"/a".split == > ["", a]

"/a/".split == > ["", a]

class FileSystem {

class Node{

boolean isFile;

String name;

String content;

Map<String, Node> fileAndFolder; // key: current folder name, value: subfolder or files names

public Node (boolean isFile, String name, String content, Map<String, Node> fileAndFolder){

this.isFile = isFile;

this.name = name;

this.content = content;

this.fileAndFolder = fileAndFolder;

}

}

Node root;

public FileSystem() {

root = new Node(false, "", "", new TreeMap<String, Node>());

}

public List<String> ls(String path) {

List<String> res = new ArrayList<>();

Node node = root;

if (!path.equals("/")){

String[] strs = path.split("/");

for (int i = 1; i < strs.length; i++){

node = node.fileAndFolder.get(strs[i]);

}

if (node.isFile){

res.add(node.name);

return res;

}

}

for (String key : node.fileAndFolder.keySet()){

res.add(key);

}

return res;

}

public void mkdir(String path) {

Node node = root;

String[] strs = path.split("/");

for (int i = 1; i < strs.length; i++){

Map<String, Node> map = node.fileAndFolder;

if (!map.containsKey(strs[i])){

map.put(strs[i], new Node(false, strs[i], "", new TreeMap<String, Node>()));

}

node = map.get(strs[i]);

}

}

public void addContentToFile(String filePath, String content) {

Node node = root;

String[] strs = filePath.split("/");

for (int i = 1; i < strs.length - 1; i++){

Map<String, Node> map = node.fileAndFolder;

node = map.get(strs[i]);

}

Map<String, Node> currMap = node.fileAndFolder;

String name = strs[strs.length-1];

if (!currMap.containsKey(name)){

currMap.put(name, new Node(true, name, content, null));

}else{

String str = currMap.get(name).content + content;

currMap.put(name, new Node(true, name, str, null));

}

}

public String readContentFromFile(String filePath) {

Node node = root;

String[] strs = filePath.split("/");

for (int i = 1; i < strs.length - 1; i++){

Map<String, Node> map = node.fileAndFolder;

node = map.get(strs[i]);

}

Map<String, Node> currMap = node.fileAndFolder;

String name = strs[strs.length-1];

return currMap.get(name).content;

}

}

/\*\*

\* Your FileSystem object will be instantiated and called as such:

\* FileSystem obj = new FileSystem();

\* List<String> param\_1 = obj.ls(path);

\* obj.mkdir(path);

\* obj.addContentToFile(filePath,content);

\* String param\_4 = obj.readContentFromFile(filePath);

\*/

After remove name which is not necessary

class FileSystem {

class Node{

boolean isFile;

String content;

Map<String, Node> fileAndFolder; // key: current subfolder or file name, value: node for subfolder or files

public Node (boolean isFile, String content, Map<String, Node> fileAndFolder){

this.isFile = isFile;

this.content = content;

this.fileAndFolder = fileAndFolder;

}

}

Node root;

public FileSystem() {

root = new Node(false, "", new TreeMap<String, Node>());

}

public List<String> ls(String path) {

List<String> res = new ArrayList<>();

Node node = root;

if (!path.equals("/")){

String[] strs = path.split("/");

for (int i = 1; i < strs.length; i++){

node = node.fileAndFolder.get(strs[i]);

}

if (node.isFile){

res.add(strs[strs.length-1]);

return res;

}

}

for (String key : node.fileAndFolder.keySet()){

res.add(key);

}

return res;

}

public void mkdir(String path) {

Node node = root;

String[] strs = path.split("/");

for (int i = 1; i < strs.length; i++){

Map<String, Node> map = node.fileAndFolder;

if (!map.containsKey(strs[i])){

map.put(strs[i], new Node(false, "", new TreeMap<String, Node>()));

}

node = map.get(strs[i]);

}

}

public void addContentToFile(String filePath, String content) {

Node node = root;

String[] strs = filePath.split("/");

for (int i = 1; i < strs.length - 1; i++){

Map<String, Node> map = node.fileAndFolder;

node = map.get(strs[i]);

}

Map<String, Node> currMap = node.fileAndFolder;

String name = strs[strs.length-1];

if (!currMap.containsKey(name)){

currMap.put(name, new Node(true, content, null));

}else{

String str = currMap.get(name).content + content;

currMap.put(name, new Node(true, str, null));

}

}

public String readContentFromFile(String filePath) {

Node node = root;

String[] strs = filePath.split("/");

for (int i = 1; i < strs.length - 1; i++){

Map<String, Node> map = node.fileAndFolder;

node = map.get(strs[i]);

}

Map<String, Node> currMap = node.fileAndFolder;

String name = strs[strs.length-1];

return currMap.get(name).content;

}

}

/\*\*

\* Your FileSystem object will be instantiated and called as such:

\* FileSystem obj = new FileSystem();

\* List<String> param\_1 = obj.ls(path);

\* obj.mkdir(path);

\* obj.addContentToFile(filePath,content);

\* String param\_4 = obj.readContentFromFile(filePath);

\*/

## 589\_N-ary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an n-ary tree, return the preorder traversal of its nodes' values.

Method 1: recursion

/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> children;

public Node() {}

public Node(int \_val,List<Node> \_children) {

val = \_val;

children = \_children;

}

};

\*/

class Solution {

public List<Integer> preorder(Node root) {

List<Integer> res = new ArrayList<>();

dfs(root, res);

return res;

}

private void dfs(Node root, List<Integer> res){

if (root == null){

return;

}

res.add(root.val);

for (Node node : root.children){

dfs(node, res);

}

}

}

Method 2: Iteration

/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> children;

public Node() {}

public Node(int \_val,List<Node> \_children) {

val = \_val;

children = \_children;

}

};

\*/

class Solution {

public List<Integer> preorder(Node root) {

List<Integer> res = new ArrayList<>();

if (root == null){

return res;

}

Stack<Node> stack = new Stack<>();

stack.push(root);

while (!stack.isEmpty()){

Node node = stack.pop();

res.add(node.val);

for (int i = node.children.size() - 1; i >= 0; i--){

stack.push(node.children.get(i));

}

}

return res;

}

}

# 59\_SpiralMatrixII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer n, generate a square matrix filled with elements from 1 to n2 in spiral order.

For example,

Given n = 3,

You should return the following matrix:

[

[ 1, 2, 3 ],

[ 8, 9, 4 ],

[ 7, 6, 5 ]

]

class Solution {

public int[][] generateMatrix(int n) {

int[][] result = new int[n][n];

int top = 0;

int bottom = n-1;

int left = 0;

int right = n-1;

int count = 1;

while (top <= bottom && left <= right){

for (int i = left; i <= right; i++){

result[top][i] = count;

count++;

}

top++;

for (int i = top; i <= bottom; i++){

result[i][right] = count;

count++;

}

right--;

if (top > bottom || left > right){

break;

}

for (int i = right; i>= left; i--){

result[bottom][i] = count;

count++;

}

bottom--;

for (int i = bottom; i>= top; i--){

result[i][left] = count;

count++;

}

left++;

}

return result;

}

}

## 590\_N-ary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an n-ary tree, return the postorder traversal of its nodes' values.

Method 1: recursion

/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> children;

public Node() {}

public Node(int \_val,List<Node> \_children) {

val = \_val;

children = \_children;

}

};

\*/

class Solution {

public List<Integer> postorder(Node root) {

List<Integer> res = new ArrayList<>();

if (root == null){

return res;

}

dfs(root, res);

return res;

}

private void dfs(Node root, List<Integer> res){

if (root == null){

return;

}

for (Node node : root.children){

dfs(node, res);

}

res.add(root.val);

}

}

Method 2: Iteration

/\*

// Definition for a Node.

class Node {

public int val;

public List<Node> children;

public Node() {}

public Node(int \_val,List<Node> \_children) {

val = \_val;

children = \_children;

}

};

\*/

class Solution {

public List<Integer> postorder(Node root) {

List<Integer> res = new ArrayList<>();

if (root == null){

return res;

}

Stack<Node> stack = new Stack<>();

stack.push(root);

while (!stack.isEmpty()){

Node node = stack.pop();

res.add(node.val);

for (Node n : node.children){

stack.push(n);

}

}

Collections.reverse(res);

return res;

}

}

## 594\_Longest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We define a harmonious array is an array where the difference between its maximum value and its minimum value is exactly 1.

Now, given an integer array, you need to find the length of its longest harmonious subsequence among all its possible subsequences.

Example 1:

Input: [1,3,2,2,5,2,3,7]

Output: 5

Explanation: The longest harmonious subsequence is [3,2,2,2,3].

Note: The length of the input array will not exceed 20,000.

class Solution {

public int findLHS(int[] nums) {

Map<Integer, Integer> map = new HashMap<>(); //store value and occurance

for (int i = 0; i < nums.length; i++){

map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);

}

int max = 0;

for (int key : map.keySet()){

if (map.containsKey(key - 1)){

max = Math.max(max, map.get(key-1) + map.get(key));

}

}

return max;

}

}

## 599\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Suppose Andy and Doris want to choose a restaurant for dinner, and they both have a list of favorite restaurants represented by strings.

You need to help them find out their common interest with the least list index sum. If there is a choice tie between answers, output

all of them with no order requirement. You could assume there always exists an answer.

Example 1:

Input:

["Shogun", "Tapioca Express", "Burger King", "KFC"]

["Piatti", "The Grill at Torrey Pines", "Hungry Hunter Steakhouse", "Shogun"]

Output: ["Shogun"]

Explanation: The only restaurant they both like is "Shogun".

Example 2:

Input:

["Shogun", "Tapioca Express", "Burger King", "KFC"]

["KFC", "Shogun", "Burger King"]

Output: ["Shogun"]

Explanation: The restaurant they both like and have the least index sum is "Shogun" with index sum 1 (0+1).

Note:

The length of both lists will be in the range of [1, 1000].

The length of strings in both lists will be in the range of [1, 30].

The index is starting from 0 to the list length minus 1.

No duplicates in both lists.

class Solution {

public String[] findRestaurant(String[] list1, String[] list2) {

List<String> list = new ArrayList<>();

Map<String, Integer> map = new HashMap<>();

for (int i = 0; i < list1.length; i++){

map.put(list1[i], i);

}

int min = Integer.MAX\_VALUE;

for (int i = 0; i < list2.length; i++){

if (map.containsKey(list2[i])){

min = Math.min(min, map.get(list2[i]) + i);

}

}

for (int i = 0; i < list2.length; i++){

if (map.containsKey(list2[i]) && map.get(list2[i]) + i == min){

list.add(list2[i]);

}

}

String[] res = new String[list.size()];

int i = 0;

for (String s: list){

res[i++] = s;

}

return res;

}

}

public String[] findRestaurant(String[] list1, String[] list2) {

Map<String, Integer> map = new HashMap<>();

List<String> res = new LinkedList<>();

int minSum = Integer.MAX\_VALUE;

for (int i=0;i<list1.length;i++) map.put(list1[i], i);

for (int i=0;i<list2.length;i++) {

Integer j = map.get(list2[i]);

if (j != null && i + j <= minSum) {

if (i + j < minSum) { res.clear(); minSum = i+j; }

res.add(list2[i]);

}

}

return res.toArray(new String[res.size()]);

}

class Solution {

public String[] findRestaurant(String[] list1, String[] list2) {

Map<String, Integer> map = new HashMap<>();

for (int i = 0; i < list1.length; i++){

map.put(list1[i], i);

}

int min = Integer.MAX\_VALUE;

List<String> res = new ArrayList<>();

for (int i = 0; i < list2.length; i++){

if (map.containsKey(list2[i])){

if (map.get(list2[i]) + i < min){

min = map.get(list2[i]) + i;

res.clear();

res.add(list2[i]);

}else if (map.get(list2[i]) + i == min){

res.add(list2[i]);

}

}

}

String[] result = new String[res.size()];

for (int i = 0; i < res.size(); i++){

result[i] = res.get(i);

}

return result;

}

}

## 6\_MergeTwoSortedArrays.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Merge two given sorted integer array A and B into a new sorted integer array.

Have you met this question in a real interview? Yes

Example

A=[1,2,3,4]

B=[2,4,5,6]

return [1,2,2,3,4,4,5,6]

public class Solution {

/\*

\* @param A: sorted integer array A

\* @param B: sorted integer array B

\* @return: A new sorted integer array

\*/

public int[] mergeSortedArray(int[] A, int[] B) {

if (A == null || B == null){

return null;

}

int[] result = new int[A.length + B.length];

int i = 0, j = 0, index = 0;

while (i < A.length && j < B.length){

if (A[i] < B[j]){

result[index++] = A[i++];

}else{

result[index++] = B[j++];

}

}

while (i < A.length){

result[index++] = A[i++];

}

while (j < B.length){

result[index++] = B[j++];

}

return result;

}

}

## 6\_ZigZagConversion.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

P A H N

A P L S I I G

Y I R

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

string convert(string text, int nRows);

convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

class Solution {

public String convert(String s, int numRows) {

if (s == null || s.length() == 0 || numRows == 1){

return s;

}

String ans = "";

for (int i = 0; i < numRows; i++){

int j = i;

String row = "";

int step = 2 \* numRows - 2;

while (j < s.length()){

row += s.charAt(j);

j += step;

int between = j - 2\*i ;

if (i != 0 && i != numRows - 1 && between < s.length()){

row += s.charAt(between);

}

}

ans += row;

}

return ans;

}

}

Best solution:

class Solution {

public String convert(String s, int numRows) {

if (numRows == 1){

return s;

}

StringBuilder res = new StringBuilder();

int steps = 2 \* numRows - 2;

int n = s.length();

for (int i = 0; i < numRows; i++){

for (int j = 0; j + i < n; j += steps){

res.append(s.charAt(j+i));

if (i!= 0 && i != numRows - 1 && j + steps - i < n){

res.append(s.charAt(j + steps - i));

}

}

}

return res.toString();

}

}

## 60\_Permutation.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

By listing and labeling all of the permutations in order,

We get the following sequence (ie, for n = 3):

"123"

"132"

"213"

"231"

"312"

"321"

Given n and k, return the kth permutation sequence.

Note: Given n will be between 1 and 9 inclusive.

Method 1:

https://leetcode.com/problems/permutation-sequence/discuss/22507/%22Explain-like-I'm-five%22-Java-Solution-in-O(n)

class Solution {

public String getPermutation(int n, int k) {

StringBuilder sb = new StringBuilder();

List<Integer> list = new ArrayList<>();

int fact = 1;

for (int i = 1; i <= n; i++){

fact \*= i;

list.add(i);

}

k--;

for (int i = 0; i < n; i++){

fact /= (n-i);

int index = k / fact;

sb.append(list.remove(index));

k -= index \* fact;

}

return sb.toString();

}

}

class Solution {

public String getPermutation(int n, int k) {

int[] nums = new int[n];

for (int i = 1; i <= n; i++){

nums[i-1] = i;

}

List<List<Integer>> list = permute(nums);

StringBuilder sb = new StringBuilder();

for (int i : list.get(k-1)){

sb.append(i);

}

return sb.toString();

}

public List<List<Integer>> permute(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

Set<Integer> seen = new HashSet<>();

dfs(res, new ArrayList<Integer>(), seen, nums);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, Set<Integer> seen, int[] nums){

if(item.size() == nums.length){

res.add(new ArrayList<Integer>(item));

return;

}

for (int i = 0; i < nums.length; i++){

if (seen.contains(nums[i])){

continue;

}

item.add(nums[i]);

seen.add(nums[i]);

dfs(res, item, seen, nums);

seen.remove(nums[i]);

item.remove(item.size() - 1);

}

}

}

Method 2: DFS TLS

class Solution {

public String getPermutation(int n, int k) {

List<List<String>> result = new ArrayList<>();

List<String> list = new ArrayList<>();

dfs(result, list, n);

StringBuilder sb = new StringBuilder();

for (String str : result.get(k-1)){

sb.append(str);

}

return sb.toString();

}

private void dfs(List<List<String>> result, List<String> list, int n){

if (list.size() == n){

result.add(new ArrayList<String>(list));

return;

}

for (int i = 1; i <= n; i++){

String str = String.valueOf(i);

if (list.contains(str)){

continue;

}

list.add(str);

dfs(result, list, n);

list.remove(list.size() - 1);

}

}

}

## 600\_Non-negative.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer n, find the number of non-negative integers less than or equal to n, whose binary representations do NOT contain consecutive ones.

Example 1:

Input: 5

Output: 5

Explanation:

Here are the non-negative integers <= 5 with their corresponding binary representations:

0 : 0

1 : 1

2 : 10

3 : 11

4 : 100

5 : 101

Among them, only integer 3 disobeys the rule (two consecutive ones) and the other 5 satisfy the rule.

Note: 1 <= n <= 109

Method 1: Brute Force TLE

Time complexity: O(32 \* n)

class Solution {

public int findIntegers(int num) {

int count = 0;

for (int i = 0; i <= num; i++){

if (check(i)){

count++;

}

}

return count;

}

private boolean check(int n){

int i = 31;

while (i > 0){

if ((n & (1 << i)) != 0 && (n & (1 << (i-1))) != 0){

return false;

}

i--;

}

return true;

}

}

Method 2: DP

Time complexity: O(32)

Space complexity: O(32)

https://leetcode.com/problems/non-negative-integers-without-consecutive-ones/discuss/103749/Java-Solution-DP

https://www.geeksforgeeks.org/count-number-binary-strings-without-consecutive-1s/

class Solution {

public int findIntegers(int num) {

StringBuilder sb = new StringBuilder(Integer.toBinaryString(num));

int len = sb.length();

int[] zeros = new int[len];

int[] ones = new int[len];

zeros[0] = 1;

ones[0] = 1;

for (int i = 1; i < len; i++){

zeros[i] = zeros[i-1] + ones[i-1];

ones[i] = zeros[i-1];

}

int ans = zeros[len-1] + ones[len-1];

//remove cases which are greater than num

for (int i = 1; i < len; i++){

if (sb.charAt(i) == '1' && sb.charAt(i-1) == '1'){

break;

}

if (sb.charAt(i) == '0' && sb.charAt(i-1) == '0'){

ans -= ones[len-i-1];

}

}

return ans;

}

}

Why break the loop when seeing a pair of 11, and minus b[i] when seeing a pair of 00? I list out the n=5 sequence. X means containing 11.

0 00000

1 00001

2 00010

3 00011 X

4 00100

5 00101

6 00110 X

7 00111 X

8 01000

9 01001

10 01010

11 01100 X

12 01100 X

13 01101 X

14 01110 X

15 01111 X

16 10000

17 10001

18 10010

19 10011 X

20 10100

21 10101

22 10110 X

23 10111 X

24 11000 X

25 11001 X

26 11010 X

27 11011 X

28 11100 X

29 11101 X

30 11110 X

31 11111 X

Now, say num = 16 (i.e. 10000)

From the most significant digit onward, look at each pair of the digits (i.e. 10, 00, 00, 00)

The first pair of 00 ==> delete 10100 & 10101

The second pair of 00 ==> delete 10010

The third pair of 00 ==> delete 10001

## 604\_Design.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design and implement a data structure for a compressed string iterator. It should support the following operations: next and hasNext.

The given compressed string will be in the form of each letter followed by a positive integer representing the number of this letter

existing in the original uncompressed string.

next() - if the original string still has uncompressed characters, return the next letter; Otherwise return a white space.

hasNext() - Judge whether there is any letter needs to be uncompressed.

Note:

Please remember to RESET your class variables declared in StringIterator, as static/class variables are persisted across multiple test

cases. Please see here for more details.

Example:

StringIterator iterator = new StringIterator("L1e2t1C1o1d1e1");

iterator.next(); // return 'L'

iterator.next(); // return 'e'

iterator.next(); // return 'e'

iterator.next(); // return 't'

iterator.next(); // return 'C'

iterator.next(); // return 'o'

iterator.next(); // return 'd'

iterator.hasNext(); // return true

iterator.next(); // return 'e'

iterator.hasNext(); // return false

iterator.next(); // return ' '

class StringIterator {

int ptr;

int num;

String str;

char nextChar;

public StringIterator(String compressedString) {

str = compressedString;

ptr = 0;

num = 0;

nextChar= ' ';

}

public char next() {

if (hasNext()){

if (num == 0){

nextChar = str.charAt(ptr++);

while (ptr < str.length() && Character.isDigit(str.charAt(ptr))){

num = 10 \* num + (int) (str.charAt(ptr) - '0');

ptr++;

}

}

num--;

return nextChar;

}

return ' ';

}

public boolean hasNext() {

if (num == 0 && ptr == str.length()){

return false;

}

return true;

}

}

/\*\*

\* Your StringIterator object will be instantiated and called as such:

\* StringIterator obj = new StringIterator(compressedString);

\* char param\_1 = obj.next();

\* boolean param\_2 = obj.hasNext();

\*/

## 604\_WindowSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of n integer, and a moving window(size k), move the window at each iteration

from the start of the array, find the sum of the element inside the window at each moving.

Have you met this question in a real interview?

Example

For array [1,2,7,8,5], moving window size k = 3.

1 + 2 + 7 = 10

2 + 7 + 8 = 17

7 + 8 + 5 = 20

return [10,17,20]

public class Solution {

/\*

\* @param nums: a list of integers.

\* @param k: length of window.

\* @return: the sum of the element inside the window at each moving.

\*/

public int[] winSum(int[] nums, int k) {

// write your code here

int n = nums.length;

if (k > n || k == 0){

return new int[0];

}

int[] res = new int[n-k+1];

int sum = 0;

for (int i = 0; i < n; i++){

sum += nums[i];

if (i >= k - 1){

res[i-k+1)] = sum;

sum -= nums[i-k+1];

}

}

return res;

}

}

## 605\_Can.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Suppose you have a long flowerbed in which some of the plots are planted and some are not. However, flowers cannot be planted in

adjacent plots - they would compete for water and both would die.

Given a flowerbed (represented as an array containing 0 and 1, where 0 means empty and 1 means not empty), and a number n, return

if n new flowers can be planted in it without violating the no-adjacent-flowers rule.

Example 1:

Input: flowerbed = [1,0,0,0,1], n = 1

Output: True

Example 2:

Input: flowerbed = [1,0,0,0,1], n = 2

Output: False

Method 1:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public boolean canPlaceFlowers(int[] flowerbed, int n) {

if (n == 0){

return true;

}

int count = 0;

int[] buff = new int[flowerbed.length+2];

for (int i = 1; i < buff.length - 1; i++){

buff[i] = flowerbed[i-1];

}

for (int i = 0; i < buff.length; i++){

if(buff[i] == 0){

count++;

if (count == 3){

n--;

if (n == 0){

return true;

}

count = 1;

}

}else{

count = 0;

}

}

return false;

}

}

Method 2: Best solution

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public boolean canPlaceFlowers(int[] flowerbed, int n) {

int count = 0;

for (int i = 0; i < flowerbed.length; i++){

if(flowerbed[i] == 0 && (i == 0 || flowerbed[i-1] == 0) && (i == flowerbed.length - 1 || flowerbed[i+1] == 0)){

flowerbed[i] = 1;

count++;

}

}

return count >= n;

}

}

## 606\_ConstructStringfromBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You need to construct a string consists of parenthesis and integers from a binary tree with the preorder traversing way.

The null node needs to be represented by empty parenthesis pair "()". And you need to omit

all the empty parenthesis pairs that don't affect the one-to-one mapping relationship between the string and the

original binary tree.

Example 1:

Input: Binary tree: [1,2,3,4]

1

/ \

2 3

/

4

Output: "1(2(4))(3)"

Explanation: Originallay it needs to be "1(2(4)())(3()())",

but you need to omit all the unnecessary empty parenthesis pairs.

And it will be "1(2(4))(3)".

Example 2:

Input: Binary tree: [1,2,3,null,4]

1

/ \

2 3

\

4

Output: "1(2()(4))(3)"

Explanation: Almost the same as the first example,

except we can't omit the first parenthesis pair to break the one-to-one mapping relationship between

the input and the output.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public String tree2str(TreeNode t) {

if (t == null){

return "";

}

StringBuilder sb = new StringBuilder();

dfs(t, sb);

return sb.toString();

}

private void dfs(TreeNode root, StringBuilder sb){

if (root == null){

return;

}

sb.append(String.valueOf(root.val));

if (root.left != null){

sb.append("(");

dfs(root.left, sb);

sb.append(")");

}

if (root.right != null){

if (root.left == null){

sb.append("()");

}

sb.append("(");

dfs(root.right, sb);

sb.append(")");

}

}

}

## 609\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of directory info including directory path, and all the files with contents in this directory, you need to find out all the groups of duplicate files in the file system in terms of their paths.

A group of duplicate files consists of at least two files that have exactly the same content.

A single directory info string in the input list has the following format:

"root/d1/d2/.../dm f1.txt(f1\_content) f2.txt(f2\_content) ... fn.txt(fn\_content)"

It means there are n files (f1.txt, f2.txt ... fn.txt with content f1\_content, f2\_content ... fn\_content, respectively) in directory root/d1/d2/.../dm. Note that n >= 1 and m >= 0. If m = 0, it means the directory is just the root directory.

The output is a list of group of duplicate file paths. For each group, it contains all the file paths of the files that have the same content. A file path is a string that has the following format:

"directory\_path/file\_name.txt"

Example 1:

Input:

["root/a 1.txt(abcd) 2.txt(efgh)", "root/c 3.txt(abcd)", "root/c/d 4.txt(efgh)", "root 4.txt(efgh)"]

Output:

[["root/a/2.txt","root/c/d/4.txt","root/4.txt"],["root/a/1.txt","root/c/3.txt"]]

Note:

No order is required for the final output.

You may assume the directory name, file name and file content only has letters and digits, and the length of file content is in the range of [1,50].

The number of files given is in the range of [1,20000].

You may assume no files or directories share the same name in the same directory.

You may assume each given directory info represents a unique directory. Directory path and file info are separated by a single blank space.

Follow-up beyond contest:

Imagine you are given a real file system, how will you search files? DFS or BFS?

If the file content is very large (GB level), how will you modify your solution?

If you can only read the file by 1kb each time, how will you modify your solution?

What is the time complexity of your modified solution? What is the most time-consuming part and memory consuming part of it? How to optimize?

How to make sure the duplicated files you find are not false positive?

https://leetcode.com/problems/find-duplicate-file-in-system/discuss/104123/C++-clean-solution-answers-to-follow-up

class Solution {

public List<List<String>> findDuplicate(String[] paths) {

List<List<String>> res = new ArrayList<>();

Map<String, List<String>> map = new HashMap<>();

for (String path : paths){

String[] strs = path.split("\\s+");

for (int i = 1; i < strs.length; i++){

String str = strs[i];

int start = str.indexOf("(");

String key = str.substring(start + 1, str.length() - 1);

if (!map.containsKey(key)){

map.put(key, new ArrayList<String>());

}

List<String> list = map.get(key);

String file = str.substring(0, start);

list.add(strs[0] + "/" + file);

}

}

for (String key : map.keySet()){

List<String> list = map.get(key);

if (list.size() > 1){

res.add(map.get(key));

}

}

return res;

}

}

## 61\_SearchforaRange.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted array of n integers, find the starting and ending position of a given target value.

If the target is not found in the array, return [-1, -1].

Have you met this question in a real interview? Yes

Example

Given [5, 7, 7, 8, 8, 10] and target value 8,

return [3, 4].

public class Solution {

/\*

\* @param A: an integer sorted array

\* @param target: an integer to be inserted

\* @return: a list of length 2, [index1, index2]

\*/

public int[] searchRange(int[] A, int target) {

int[] results = new int[2];

results[0] = -1;

results[1] = -1;

if (A == null || A.length == 0){

return results;

}

int start = 0;

int end = A.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (A[mid] == target){

end = mid;

}else if (A[mid] < target){

start = mid;

}else{

end = mid;

}

}

if (A[start] == target){

results[0] = start;

}else if (A[end] == target){

results[0] = end;

}

start = 0;

end = A.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (A[mid] == target){

start = mid;

}else if (A[mid] < target){

start = mid;

}else{

end = mid;

}

}

if (A[end] == target){

results[1] = end;

}else if (A[start] == target){

results[1] = start;

}

return results;

}

}

## 611\_ValidTriangleNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array consists of non-negative integers, your task is to count the number of triplets chosen from the array that

can make triangles if we take them as side lengths of a triangle.

Example 1:

Input: [2,2,3,4]

Output: 3

Explanation:

Valid combinations are:

2,3,4 (using the first 2)

2,3,4 (using the second 2)

2,2,3

Note:

The length of the given array won't exceed 1000.

The integers in the given array are in the range of [0, 1000].

class Solution {

public int triangleNumber(int[] nums) {

int ans = 0;

Arrays.sort(nums);

for (int i = nums.length - 1; i >= 2; i--){

int left = 0;

int right = i - 1;

while (left < right){

if (nums[right] + nums[left] > nums[i]){

ans += right - left;

right--;

}else{

left++;

}

}

}

return ans;

}

}

## 613\_HighFive.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are two properties in the node student id and scores, to ensure that each student will have at least 5 points,

find the average of 5 highest scores for each person.

Have you met this question in a real interview?

Example

Given results = [[1,91],[1,92],[2,93],[2,99],[2,98],[2,97],[1,60],[1,58],[2,100],[1,61]]

Return

public Map<Integer, Double> highFive(Record[] results) {

Map<Integer, Double> result = new HashMap<>();

Map<Integer, PriorityQueue<Integer>> map = new HashMap<>();

if (results == null || results.length == 0) {

return result;

}

for (int i = 0; i < results.length; i++) {

if (!map.containsKey(results[i].id)) {

map.put(results[i].id, new PriorityQueue<Integer>(5));

}

// map.get(results[i].id).offer(results[i].score);

// if (map.get(results[i].id).size() > 5) {

// map.get(results[i].id).poll();

// }

if (map.get(results[i].id).size() < 5) {

map.get(results[i].id).offer(results[i].score);

} else {

if (results[i].score > map.get(results[i].id).peek()) {

map.get(results[i].id).poll();

map.get(results[i].id).offer(results[i].score);

}

}

}

for (int key: map.keySet()) {

int sum = 0;

for (int val: map.get(key)) {

sum += val;

}

result.put(key, sum / 5.0);

}

return result;

}

}

## 616\_Add.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s and a list of strings dict, you need to add a closed pair of bold tag <b> and </b> to wrap the substrings in s

that exist in dict. If two such substrings overlap, you need to wrap them together by only one pair of closed bold tag. Also, if

two substrings wrapped by bold tags are consecutive, you need to combine them.

Example 1:

Input:

s = "abcxyz123"

dict = ["abc","123"]

Output:

"<b>abc</b>xyz<b>123</b>"

Example 2:

Input:

s = "aaabbcc"

dict = ["aaa","aab","bc"]

Output:

"<b>aaabbc</b>c"

Note:

The given dict won't contain duplicates, and its length won't exceed 100.

All the strings in input have length in range [1, 1000].

Method 1: based on merge interval

Consider you have string

s = "aaabbcc"

dict = ["aaa","aab","bc"]

you find the index of each string in dict, conver to an interval, you will get

[[0, 3], [1, 4], [4, 6]]

aaa aab bc

then combine these intervals

Ater merged, we got [0,6], so we know "aaabbc" needs to be surrounded by tag.

class Solution {

class Interval{

int start;

int end;

Interval(int start, int end){

this.start = start;

this.end = end;

}

public String toString() {

return "[" + start + ", " + end + "]" ;

}

}

public List<Interval> merge(List<Interval> intervals){

List<Interval> result = new ArrayList<>();

if (intervals == null || intervals.size() == 0){

return result;

}

Collections.sort(intervals, new Comparator<Interval>(){

public int compare (Interval a, Interval b){

if (a.start == b.start){

return a.end - b.end;

}

return a.start - b.start;

}

});

Interval last = null;

for (Interval interval : intervals){

if (last == null || last.end < interval.start){

result.add(interval);

last = interval;

}else{

last.end = Math.max(last.end, interval.end);

}

}

return result;

}

public String addBoldTag(String s, String[] dict) {

List<Interval> intervals = new ArrayList<>();

for (String str : dict){

int index = -1;

index = s.indexOf(str, index);

while (index != -1){

intervals.add(new Interval(index, index + str.length() ));

index++;

index = s.indexOf(str, index);

}

}

System.out.println(Arrays.toString(intervals.toArray()));

intervals = merge(intervals);

System.out.println(Arrays.toString(intervals.toArray()));

StringBuilder sb = new StringBuilder();

int prev = 0;

for (Interval interval : intervals){

sb.append(s.substring(prev, interval.start));

sb.append("<b>");

sb.append(s.substring(interval.start, interval.end));

sb.append("</b>");

prev = interval.end;

}

if (prev < s.length()){

sb.append(s.substring(prev));

}

return sb.toString();

}

}

## 621\_Task.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a char array representing tasks CPU need to do. It contains capital letters A to Z

where different letters represent different tasks.Tasks could be done without original order.

Each task could be done in one interval. For each interval, CPU could finish one task or just be idle.

However, there is a non-negative cooling interval n that means between two same tasks, there must

be at least n intervals that CPU are doing different tasks or just be idle.

You need to return the least number of intervals the CPU will take to finish all the given tasks.

Example 1:

Input: tasks = ["A","A","A","B","B","B"], n = 2

Output: 8

Explanation: A -> B -> idle -> A -> B -> idle -> A -> B.

Note:

The number of tasks is in the range [1, 10000].

The integer n is in the range [0, 100].

Method 1:

Complexity Analysis

Time complexity : O(time). Number of iterations will be equal to resultant time.

Space complexity : O(1). Constant size array map is used.

class Solution {

public int leastInterval(char[] tasks, int n) {

int[] map = new int[26];

for (int i = 0; i < tasks.length; i++){

map[tasks[i] - 'A']++;

}

Arrays.sort(map);

int idle = 0;

while (map[25] > 0){

int i = 0;

while(i <= n){

if (map[25] == 0){

break;

}

if (i <= 25 && map[25 - i] > 0){

map[25 - i]--;

}else{

idle++;

}

i++;

}

Arrays.sort(map);

}

return tasks.length + idle;

}

}

Method 2:

class Solution {

public int leastInterval(char[] tasks, int n) {

int[] map = new int[26];

for (char c : tasks){

map[c - 'A']++;

}

Arrays.sort(map);

int count = 0;

int i = 25;

while (i>=0 && map[i] == map[25]){

i--;

count++;

}

return Math.max(tasks.length, (map[25]-1)\*(n+1) + count);

}

}

https://leetcode.com/problems/task-scheduler/discuss/104496/concise-Java-Solution-O(N)-time-O(26)-space

Method 3: best solution

class Solution {

public int leastInterval(char[] tasks, int n) {

int[] map = new int[26];

for (char c : tasks){

map[c - 'A']++;

}

int maxCount = 0;

int count = 0;

for (int i = 0; i < 26; i++){

if (map[i] > maxCount){

maxCount = map[i];

count = 1;

}else if (map[i] == maxCount){

count++;

}

}

return Math.max(tasks.length, (maxCount-1)\*(n+1) + count);

}

}

better:

Corner case: when n == 1

AAABBB 0 , n there is 0, it means same task can put together,

class Solution {

public int leastInterval(char[] tasks, int n) {

int maxCount = 0;

Map<Character, Integer> map = new HashMap<>();

for (char c : tasks){

map.put(c, map.getOrDefault(c, 0) + 1);

}

int extra = 0;

for (char key : map.keySet()){

if (maxCount == map.get(key)){

extra++;

}else if (maxCount < map.get(key)){

maxCount = map.get(key);

extra = 1;

}

}

return Math.max(tasks.length, (maxCount-1) \* (n+1) + extra);

}

}

## 622\_GuessNumberGame.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are playing the Guess Game. The game is as follows:

I pick a number from 1 to n. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number is higher or lower.

You call a pre-defined API guess(int num) which returns 3 possible results (-1, 1, or 0):

Have you met this question in a real interview? Yes

Example

n = 10, I pick 4 (but you don't know)

Return 4. Correct !

/\* The guess API is defined in the parent class GuessGame.

@param num, your guess

@return -1 if my number is lower, 1 if my number is higher, otherwise return 0

int guess(int num); \*/

public class Solution extends GuessGame {

/\*\*

\* @param n an integer

\* @return the number you guess

\*/

public int guessNumber(int n) {

int l = 1;

int r = n;

while (l + 1 < r){

int mid = l + (r - l) / 2;

int res = guess(mid);

if (res == 0){

return mid;

}else if (res == -1){

r = mid;

}else{

l = mid;

}

}

if (guess(l) == 0){

return l;

}

if (guess(r) == 0){

return r;

}

return 0;

}

}

## 624\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given m arrays, and each array is sorted in ascending order. Now you can pick up two integers from two different arrays

(each array picks one) and calculate the distance. We define the distance between two integers a and b to be their absolute

difference |a-b|. Your task is to find the maximum distance.

Example 1:

Input:

[[1,2,3],

[4,5],

[1,2,3]]

Output: 4

Explanation:

One way to reach the maximum distance 4 is to pick 1 in the first or third array and pick 5 in the second array.

Note:

Each given array will have at least 1 number. There will be at least two non-empty arrays.

The total number of the integers in all the m arrays will be in the range of [2, 10000].

The integers in the m arrays will be in the range of [-10000, 10000].

Method 1:

Template: keep three varialbles, max (or min) value, max (or min) index and 2nd max(or min) value

Similar concept as Paint House II https://github.com/optimisea/Leetcode/blob/master/Java/265\_Paint.java

to find the best and keep 2nd best

class Solution {

public int maxDistance(List<List<Integer>> arrays) {

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;

int minInd = 0;

int maxInd = 0;

int minSec = Integer.MAX\_VALUE;

int maxSec = Integer.MIN\_VALUE;

for (int i = 0; i < arrays.size(); i++){

int small = arrays.get(i).get(0);

int large = arrays.get(i).get(arrays.get(i).size() - 1);

if (small < min){

minSec = min;

min = small;

minInd = i;

}else if (small < minSec){

minSec = small;

}

if (large > max){

maxSec = max;

max = large;

maxInd = i;

}else if (max > maxSec){

maxSec = large;

}

}

return minInd != maxInd ? max - min : Math.max(maxSec - min, max - minSec);

}

}

Method 2: Very Good like DP

class Solution {

public int maxDistance(List<List<Integer>> arrays) {

int ans = 0;

int min = arrays.get(0).get(0);

int max = arrays.get(0).get(arrays.get(0).size() - 1);

for (int i = 1; i < arrays.size(); i++){

int small = arrays.get(i).get(0);

int large = arrays.get(i).get(arrays.get(i).size() - 1);

ans = Math.max(ans, Math.max(Math.abs(max - small), Math.abs(large - min)));

min = Math.min(min, small);

max = Math.max(max, large);

}

return ans;

}

}

## 625\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer a, find the smallest positive integer b whose multiplication of each digit equals to a.

If there is no answer or the answer is not fit in 32-bit signed integer, then return 0.

Example 1

Input:

48

Output:

68

Example 2

Input:

15

Output:

35

Factorization:

We know that the final number generated, resresres, should be such that its digits should have a product equal to the given number aaa.

In other words, the digits of resresres will be the factors of the given number aaa. Thus, our problem reduces to finding the factors

(not necessarily prime) of aaa and finding their smallest possible arrangement. Thus, we start with trying with the largest possible

factor 999, obtain as many such counts of this factor as possible in resresres and place such factors obtianed at its least significant

positions. Then, we go on decrementing the number currently considered as the possible factor and if it is a factor, we keep on placing

it at relatively more significant positions in resresres. We go on getting such factors till we are done considering all the numbers

from 9 to 2. At the end, resresres gives the required result.

class Solution {

public int smallestFactorization(int a) {

if (a < 2){

return a;

}

long res = 0;

long multi = 1;

for (int i = 9; i >= 2; i--){

while (a % i == 0){

a = a / i;

res += i \* multi;

multi = multi \* 10;

}

}

return a < 10 && res <= Integer.MAX\_VALUE ? (int)res : 0;

}

}

## 626\_RectangleOverlap.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two rectangles, find if the given two rectangles overlap or not.

Notice

l1: Top Left coordinate of first rectangle.

r1: Bottom Right coordinate of first rectangle.

l2: Top Left coordinate of second rectangle.

r2: Bottom Right coordinate of second rectangle.

l1 != r2 and l2 != r2

Have you met this question in a real interview? Yes

Example

Given l1 = [0, 8], r1 = [8, 0], l2 = [6, 6], r2 = [10, 0], return true

Given l1 = [0, 8], r1 = [8, 0], l2 = [9, 6], r2 = [10, 0], return `false

思路:

• 最基础的想法分类讨论

• 假设一个固定,另一个从左往右移,那么在一个维度上分别是:

• 不重叠 重叠重叠重叠 不重叠 • 两个维度上都重叠矩形才重叠

• 分类讨论有点麻烦,有更简单的方法吗?

• 正着想麻烦就反着来

• 考虑下不重叠的时候是什么情况? • 要么上下、要么左右

public class Solution {

/\*\*

\* @param l1 top-left coordinate of first rectangle

\* @param r1 bottom-right coordinate of first rectangle

\* @param l2 top-left coordinate of second rectangle

\* @param r2 bottom-right coordinate of second rectangle

\* @return true if they are overlap or false

\*/

public boolean doOverlap(Point l1, Point r1, Point l2, Point r2) {

if (l1.x > r2.x || l2.x > r1.x){

return false;

}

if (l1.y < r2.y || l2.y < r1.y){

return false;

}

return true;

}

}

class Solution {

public boolean isRectangleOverlap(int[] rec1, int[] rec2) {

return rec1[2] > rec2[0] && rec1[3] > rec2[1] && rec1[0] < rec2[2] && rec1[1] < rec2[3];

}

}

## 627\_LongestPalindrome.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string which consists of lowercase or uppercase letters, find the length of the longest palindromes

that can be built with those letters.

This is case sensitive, for example "Aa" is not considered a palindrome here.

Notice

Assume the length of given string will not exceed 1010.

Have you met this question in a real interview? Yes

Example

Given s = "abccccdd" return 7

One longest palindrome that can be built is "dccaccd", whose length is 7.

public class Solution {

/\*

\* @param s: a string which consists of lowercase or uppercase letters

\* @return: the length of the longest palindromes that can be built

\*/

public int longestPalindrome(String s) {

if (s == null || s.length() == 0){

return 0;

}

int[] hash = new int[256];

int count = 0;

boolean single = false;

for (int i = 0; i < s.length(); i++){

hash[s.charAt(i)]++;

}

for (int i = 0; i < 256; i++){

if (hash[i] % 2 == 0){

count += hash[i];

}else{

count += hash[i] - 1;

single = true;

}

}

if (single){

return count + 1;

}

return count;

}

}

## 628\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array, find three numbers whose product is maximum and output the maximum product.

Example 1:

Input: [1,2,3]

Output: 6

Example 2:

Input: [1,2,3,4]

Output: 24

Note:

The length of the given array will be in range [3,104] and all elements are in the range [-1000, 1000].

Multiplication of any three numbers in the input won't exceed the range of 32-bit signed integer.

Method 1:

Time complexity: O(nlogn)

class Solution {

public int maximumProduct(int[] nums) {

int n = nums.length;

Arrays.sort(nums);

return Math.max(nums[n-1] \* nums[n-2] \* nums[n-3], nums[0] \* nums[1] \* nums[n-1]);

}

}

Method 2:

Time complexity: O(n)

Simply find out the three largest numbers and the two smallest numbers using one pass.

class Solution {

public int maximumProduct(int[] nums) {

int n = nums.length;

int max1 = Integer.MIN\_VALUE;

int max2 = Integer.MIN\_VALUE;

int max3 = Integer.MIN\_VALUE;

int min1 = Integer.MAX\_VALUE;

int min2 = Integer.MAX\_VALUE;

for (int i = 0; i < n; i++){

if (nums[i] > max1){

max3 = max2;

max2 = max1;

max1 = nums[i];

}else if (nums[i] > max2){

max3 = max2;

max2 = nums[i];

}else if (nums[i] > max3){

max3 = nums[i];

}

}

for (int i = 0; i < n; i++){

if (nums[i] < min1){

min2 = min1;

min1 = nums[i];

}else if (nums[i] < min2){

min2 = nums[i];

}

}

return Math.max(max1 \* max2 \* max3, min1 \* min2 \* max1);

}

}

class Solution {

public int maximumProduct(int[] nums) {

int max = Integer.MIN\_VALUE;

int max2 = Integer.MIN\_VALUE;

int max3 = Integer.MIN\_VALUE;

int min = Integer.MAX\_VALUE;

int min2 = Integer.MAX\_VALUE;

for (int i : nums){

if (i > max){

max3 = max2;

max2 = max;

max = i;

}else if (i > max2){

max3 = max2;

max2 = i;

}else if (i > max3){

max3 = i;

}

if (i < min){

min2 = min;

min = i;

}else if (i < min2){

min2 = i;

}

}

return Math.max(max \* max2 \* max3, max \* min \* min2);

}

}

## 628\_MaximumSubtree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, find the subtree with maximum sum. Return the root of the subtree.

Notice

LintCode will print the subtree which root is your return node.

It's guaranteed that there is only one subtree with maximum sum and the given binary tree is not an empty tree.

Have you met this question in a real interview? Yes

Example

Given a binary tree:

1

/ \

-5 2

/ \ / \

0 3 -4 -5

return the node with value 3.

public class Solution {

/\*

\* @param root: the root of binary tree

\* @return: the maximum weight node

\*/

int max = Integer.MIN\_VALUE;

TreeNode node = null;

public TreeNode findSubtree(TreeNode root) {

int sum = dfs(root);

return node;

}

private int dfs(TreeNode cur){

if (cur == null){

return 0;

}

int leftSum = dfs(cur.left);

int rightSum = dfs(cur.right);

int sum = leftSum + rightSum + cur.val;

if (max < sum){

max = sum;

node = cur;

}

return sum;

}

}

## 629\_MinimumSpanningTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of Connections, which is the Connection class (the city name at both ends of the edge and a cost between them), find some

edges, connect all the cities and spend the least amount.

Return the connects if can connect all the cities, otherwise return empty list.

Notice

Return the connections sorted by the cost, or sorted city1 name if their cost is same, or sorted city2 if their city1

name is also same.

Have you met this question in a real interview? Yes

Example

Gievn the connections = ["Acity","Bcity",1], ["Acity","Ccity",2], ["Bcity","Ccity",3]

Return ["Acity","Bcity",1], ["Acity","Ccity",2]

Method: Union Find + Kruskal

/\*\*

\* Definition for a Connection.

\* public class Connection {

\* public String city1, city2;

\* public int cost;

\* public Connection(String city1, String city2, int cost) {

\* this.city1 = city1;

\* this.city2 = city2;

\* this.cost = cost;

\* }

\* }

\*/

public class Solution {

/\*\*

\* @param connections given a list of connections include two cities and cost

\* @return a list of connections from results

\*/

class UFS{

int[] f;

public UFS(int n){

f = new int[n];

for (int i = 0; i < n; i++){

f[i] = -1;

}

}

public int find(int x){

if (f[x] < 0){

return x;

}

f[x] = find(f[x]);

return f[x];

}

public void union(int a, int b){

a = find(a);

b = find(b);

if (f[a] < f[b]){

f[a] += f[b];

f[b] = a;

}else{

f[b] += f[a];

f[a] = b;

}

}

}

Map<String, Integer> name2ID = new HashMap<>();

int n = 0;

public int getID(String name){

if (name2ID.containsKey(name)){

return name2ID.get(name);

}else{

name2ID.put(name, n++);

return n - 1;

}

}

public List<Connection> lowestCost(List<Connection> connections) {

Collections.sort(connections, new Comparator<Connection>(){

public int compare(Connection a, Connection b){

if (a.cost != b.cost){

return a.cost - b.cost;

}else if (!a.city1.equals(b.city1)){

return a.city1.compareTo(b.city1);

}

return a.city2.compareTo(b.city2);

}

});

List<Connection> result = new ArrayList<>();

UFS ufs = new UFS(connections.size() \* 2);

for (Connection conn : connections){

int id1 = getID(conn.city1);

int id2 = getID(conn.city2);

if (ufs.find(id1) != ufs.find(id2)){

result.add(conn);

ufs.union(id1, id2);

}

}

if (result.size() == n - 1){

return result;

}

return new ArrayList<>();

}

}

## 63\_UniquePathsII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

[

[0,0,0],

[0,1,0],

[0,0,0]

]

The total number of unique paths is 2.

Note: m and n will be at most 100.

class Solution {

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

if (obstacleGrid == null || obstacleGrid.length == 0 || obstacleGrid[0].length == 0){

return 0;

}

int n = obstacleGrid.length;

int m = obstacleGrid[0].length;

int[][] f= new int[n][m];

if (obstacleGrid[0][0] == 1){

return 0;

}else{

f[0][0] = 1;

}

for (int i = 1; i < n; i++){

if (obstacleGrid[i][0] == 1){

f[i][0] = 0;

}else{

f[i][0] = f[i-1][0];

}

}

for (int i = 1; i < m; i++){

if (obstacleGrid[0][i] == 1){

f[0][i] = 0;

}else{

f[0][i] = f[0][i-1];

}

}

for (int i = 1; i < n; i++){

for (int j = 1; j < m; j++){

if (obstacleGrid[i][j] == 1){

f[i][j] = 0;

}else{

f[i][j] = f[i-1][j] + f[i][j-1];

}

}

}

return f[n-1][m-1];

}

}

Better version:

class Solution {

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

int m = obstacleGrid.length;

int n = obstacleGrid[0].length;

int[][] dp = new int[m][n];

if (obstacleGrid[0][0] == 1){

return 0;

}

dp[0][0] = 1;

for (int i = 1; i < m; i++){

if (obstacleGrid[i][0] == 1){

dp[i][0] = 0;

}else{

dp[i][0] = dp[i-1][0];

}

}

for (int j = 1; j < n; j++){

if (obstacleGrid[0][j] == 1){

dp[0][j] = 0;

}else{

dp[0][j] = dp[0][j-1];

}

}

for (int i = 1; i < m; i++){

for (int j = 1; j < n; j++){

if (obstacleGrid[i][j] == 1){

dp[i][j] = 0;

}else{

dp[i][j] = dp[i-1][j] + dp[i][j-1];

}

}

}

return dp[m-1][n-1];

}

}

## 630\_Course.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are n different online courses numbered from 1 to n. Each course has some duration(course length) t and closed on dth day. A course should be taken continuously for t days and must be finished before or on the dth day. You will start at the 1st day.

Given n online courses represented by pairs (t,d), your task is to find the maximal number of courses that can be taken.

Example:

Input: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]

Output: 3

Explanation:

There're totally 4 courses, but you can take 3 courses at most:

First, take the 1st course, it costs 100 days so you will finish it on the 100th day, and ready to take the next course on the 101st day.

Second, take the 3rd course, it costs 1000 days so you will finish it on the 1100th day, and ready to take the next course on the 1101st day.

Third, take the 2nd course, it costs 200 days so you will finish it on the 1300th day.

The 4th course cannot be taken now, since you will finish it on the 3300th day, which exceeds the closed date.

Note:

The integer 1 <= d, t, n <= 10,000.

You can't take two courses simultaneously.

class Solution {

public int scheduleCourse(int[][] courses) {

//Sort the courses by their deadlines (Greedy! We have to deal with courses with early deadlines first)

Arrays.sort(courses, (a, b)->a[1] - b[1]);

Queue<Integer> pq = new PriorityQueue<Integer>((a,b)->b-a);

int time = 0;

for (int[] course : courses){

time += course[0];

pq.offer(course[0]);

if (time > course[1]){

//If time exceeds, drop the previous course which costs the most time. (That must be the best choice!)

time -= pq.poll();

}

}

return pq.size();

}

}

class Solution {

public int scheduleCourse(int[][] courses) {

Arrays.sort(courses, new Comparator<int[]>(){

public int compare(int[] a1, int[] a2){

return a1[1] - a2[1];

}

});

Queue<Integer> pq = new PriorityQueue<>(new Comparator<Integer>(){

public int compare(Integer i1, Integer i2){

return (int)(i2 - i1);

}

});

int time = 0;

for (int[] c : courses){

time += c[0];

pq.offer(c[0]);

if (time > c[1]){

time -= pq.poll();

}

}

return pq.size();

}

}

## 632\_Smallest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have k lists of sorted integers in ascending order. Find the smallest range that includes at least one number from each of the

k lists.

We define the range [a,b] is smaller than range [c,d] if b-a < d-c or a < c if b-a == d-c.

Example 1:

Input:[[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]

Output: [20,24]

Explanation:

List 1: [4, 10, 15, 24,26], 24 is in range [20,24].

List 2: [0, 9, 12, 20], 20 is in range [20,24].

List 3: [5, 18, 22, 30], 22 is in range [20,24].

Note:

The given list may contain duplicates, so ascending order means >= here.

1 <= k <= 3500

-105 <= value of elements <= 105.

Method: use priority queue similar as Merge K array

class Solution {

class Pair {

int val;

int list;

int idx;

public Pair (int val, int list, int idx){

this.val = val;

this.list = list;

this.idx = idx;

}

}

public int[] smallestRange(List<List<Integer>> nums) {

int[] res = new int[2];

Queue<Pair> pq = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

if (p1.val == p2.val){

return p1.idx - p2.idx;

}

return p1.val - p2.val;

}

});

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;//need keep tracking the max when enqueue in order to calculate the min window

for (int i = 0; i < nums.size(); i++){

pq.offer(new Pair(nums.get(i).get(0), i, 0));

max = Math.max(max, nums.get(i).get(0));

}

while (pq.size() == nums.size()){

Pair p = pq.poll();

if (min > max - p.val){

min = max - p.val;

res[0] = p.val;

res[1] = max;

}

List<Integer> l = nums.get(p.list);

int index = p.idx + 1;

if (index == l.size()){

break;

}

max = Math.max(max, l.get(index));

pq.offer(new Pair(l.get(index), p.list, index));

}

return res;

}

}

Better solution:

class Solution {

class Point{

int val;

int index;

int listIdx;

public Point (int val, int index, int listIdx){

this.val = val;

this.index = index;

this.listIdx = listIdx;

}

}

public int[] smallestRange(List<List<Integer>> nums) {

Queue<Point> pq = new PriorityQueue<Point>(new Comparator<Point>(){

public int compare (Point p1, Point p2){

if (p1.val == p2.val){

return p1.index - p2.index;

}

return p1.val - p2.val;

}

});

int minLen = Integer.MAX\_VALUE;//track the minimum length

int maxVal = Integer.MIN\_VALUE;//track the maximum value

for (int i = 0; i < nums.size(); i++){

List<Integer> list = nums.get(i);

int val = list.get(0);

pq.offer(new Point(val, 0, i));

maxVal = Math.max(maxVal, val);

}

int[] res = new int[2];

while (!pq.isEmpty()){

Point p = pq.poll();

if (minLen > maxVal - p.val){

minLen = maxVal - p.val;

res[0] = p.val;

res[1] = maxVal;

}

if (p.index + 1 < nums.get(p.listIdx).size()){

int nextVal = nums.get(p.listIdx).get(p.index + 1);

pq.offer(new Point(nextVal, p.index + 1, p.listIdx));

maxVal = Math.max(maxVal, nextVal);

}else{

break;

}

}

return res;

}

}

## 633\_Sum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-negative integer c, your task is to decide whether there're two integers a and b such that a2 + b2 = c.

Example 1:

Input: 5

Output: True

Explanation: 1 \* 1 + 2 \* 2 = 5

Example 2:

Input: 3

Output: False

Method 1: Two points (Best solution)

Time compleixty: O(sqrt(n))

Space compleixty: O(1)

class Solution {

public boolean judgeSquareSum(int c) {

int left = 0;

int right = (int)Math.sqrt(c);

while (left <= right){

int val = left \* left + right \* right;

if (val == c){

return true;

}else if (val < c){

left++;

}else{

right--;

}

}

return false;

}

}

Method 2: HashSet

Time complexity: O(sqrt(n))

Space complexity: O(sqrt(n))

class Solution {

public boolean judgeSquareSum(int c) {

Set<Integer> set = new HashSet<>();

for (int i = 0; i <= Math.sqrt(c); i++){

set.add(i \* i);

if (set.contains(c - i \* i)){

return true;

}

}

return false;

}

}

## 634\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In combinatorial mathematics, a derangement is a permutation of the elements of a set, such that no element appears

in its original position.

There's originally an array consisting of n integers from 1 to n in ascending order, you need to find the number of

derangement it can generate.

Also, since the answer may be very large, you should return the output mod 109 + 7.

Example 1:

Input: 3

Output: 2

Explanation: The original array is [1,2,3]. The two derangements are [2,3,1] and [3,1,2].

Note:

n is in the range of [1, 106].

http://www.cnblogs.com/grandyang/p/7210929.html

Put elment i at index i -1, then it will generate two cases:

1) put i - 1 at index i => dp[i-2];

2) put i -1 not at index i => dp[i-1]

i could be put at (i-1) different locations. so,

dp[i]=(i−1)∗(dp[i−1]+dp[i−2])

class Solution {

public int findDerangement(int n) {

if (n == 0){

return 1;

}

if (n == 1){

return 0;

}

int mod = (int) Math.pow(10, 9) + 7;

int[] dp = new int[n+1];

dp[0] = 1;

dp[1] = 0;

for (int i = 2; i <= n; i++){

dp[i] = (int)(((i-1L) \* (dp[i-1] + dp[i-2])) % mod);

}

return dp[n];

}

}

## 634\_WordSquares.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a set of words without duplicates, find all word squares you can build from them.

A sequence of words forms a valid word square if the kth row and column read the exact same string,

where 0 ≤ k < max(numRows, numColumns).

For example, the word sequence ["ball","area","lead","lady"] forms a word square because each word reads

the same both horizontally and vertically.

b a l l

a r e a

l e a d

l a d y

Notice

There are at least 1 and at most 1000 words.

All words will have the exact same length.

Word length is at least 1 and at most 5.

Each word contains only lowercase English alphabet a-z.

Have you met this question in a real interview? Yes

Example

Given a set ["area","lead","wall","lady","ball"]

return [["wall","area","lead","lady"],["ball","area","lead","lady"]]

Explanation:

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

Given a set ["abat","baba","atan","atal"]

return [["baba","abat","baba","atan"],["baba","abat","baba","atal"]]

Explanation:

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

思路:

• 裸写DFS很简单:

– 枚举每一行选哪个单词

– 枚举完后check是否满足条件 – 但是时间上过不了最坏1000^5

• 所以怎办? – DFS+剪枝

• DFS中,什么是剪枝?

– 剪枝就是去掉搜索过程中的冗余

• 什么是冗余?

– 就是搜到某个情况下,明显往后继续搜是搜不出结果的

• 这一题有哪些冗余?

• 冗余一:

– 第一个词填了ball后,第二个词必须以a开头 – 第二个词填了area后,第三个词必须以le开头 – 以其他开头的就没必要搜下去了

– 怎么实现?

• 用Hash or Trie树记录下以某个前缀开头的有哪些单词

• 比如以l开头的有lead lady,以le开头的有lead,以lea开头的有lead • 每次只用从特定开头的单词中继续往后搜

• 冗余二:

– 第一个词填了ball

– 第二个词想填area的话

– 字典中必须有以le la开头的单词,否则没有的话就不能填area

– 怎么实现?

• 直接利用冗余一中的Hash or Trie树

public class Solution {

/\*

\* @param words: a set of words without duplicates

\* @return: all word squares

\*/

Map<String, List<String>> map = new HashMap<>();

public List<List<String>> wordSquares(String[] words) {

List<List<String>> result = new ArrayList<>();

if (words == null || words.length == 0){

return result;

}

List<String> square = new ArrayList<>();

int len = words[0].length();

initPrefix(words);

dfs(result, square, words, len, 0);

return result;

}

private void dfs(List<List<String>> result, List<String> square,

String[] words, int len, int start){

if (start == len){

result.add(new ArrayList<String>(square));

return;

}

String pre = "";

for (int i = 0; i < start; i++){

pre += square.get(i).charAt(start);

}

List<String> candidate = map.get(pre);

for (String can : candidate){

if (!check(can, start, square)){

continue;

}

square.add(can);

dfs(result, square, words, len, start + 1);

square.remove(square.size() - 1);

}

}

private void initPrefix(String[] words){

for (String str : words){

if (!map.containsKey("")){

map.put("", new ArrayList<String>());

}

map.get("").add(str);

String pre = "";

for (int i = 0; i < str.length(); i++){

pre += str.charAt(i);

if (!map.containsKey(pre)){

map.put(pre, new ArrayList<String>());

}

map.get(pre).add(str);

}

}

}

private boolean check(String str, int start, List<String> square){

for (int i = start + 1; i < str.length(); i++){

String pre = "";

for (int j = 0; j < start; j++){

pre += square.get(j).charAt(i);

}

pre += str.charAt(i);

if (!map.containsKey(pre)){

return false;

}

}

return true;

}

}

## 635\_Design.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given several logs that each log contains a unique id and timestamp. Timestamp is a string that has the following format:

Year:Month:Day:Hour:Minute:Second, for example, 2017:01:01:23:59:59. All domains are zero-padded decimal numbers.

Design a log storage system to implement the following functions:

void Put(int id, string timestamp): Given a log's unique id and timestamp, store the log in your storage system.

int[] Retrieve(String start, String end, String granularity): Return the id of logs whose timestamps are within the range

from start to end. Start and end all have the same format as timestamp. However, granularity means the time level for consideration.

For example, start = "2017:01:01:23:59:59", end = "2017:01:02:23:59:59", granularity = "Day", it means that we need to find the

logs within the range from Jan. 1st 2017 to Jan. 2nd 2017.

Example 1:

put(1, "2017:01:01:23:59:59");

put(2, "2017:01:01:22:59:59");

put(3, "2016:01:01:00:00:00");

retrieve("2016:01:01:01:01:01","2017:01:01:23:00:00","Year"); // return [1,2,3], because you need to return all logs within 2016 and 2017.

retrieve("2016:01:01:01:01:01","2017:01:01:23:00:00","Hour"); // return [1,2], because you need to return all logs start from 2016:01:01:01 to 2017:01:01:23, where log 3 is left outside the range.

Note:

There will be at most 300 operations of Put or Retrieve.

Year ranges from [2000,2017]. Hour ranges from [00,23].

Output for Retrieve has no order required.

Key:

Arrays.asList()

IndexOf()

compareTo()

class LogSystem {

List<String[]> log;

List<String> units;

int[] indices;

public LogSystem() {

log = new ArrayList<String[]>();

units = Arrays.asList("Year", "Month", "Day", "Hour", "Minute", "Second");

indices = new int[]{4, 7, 10, 13, 16, 19};

}

public void put(int id, String timestamp) {

log.add(new String[]{Integer.toString(id), timestamp});

}

public List<Integer> retrieve(String s, String e, String gra) {

List<Integer> res = new ArrayList<>();

int index = units.indexOf(gra);

int pos = indices[index];

for (String[] strArray : log){

String str = strArray[1].substring(0, pos);

if (str.compareTo(s.substring(0, pos)) >= 0 && str.compareTo(e.substring(0, pos)) <= 0){

res.add(Integer.parseInt(strArray[0]));

}

}

return res;

}

}

/\*\*

\* Your LogSystem object will be instantiated and called as such:

\* LogSystem obj = new LogSystem();

\* obj.put(id,timestamp);

\* List<Integer> param\_2 = obj.retrieve(s,e,gra);

\*/

## 636\_Exclusive.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the running logs of n functions that are executed in a nonpreemptive single threaded CPU, find the exclusive time of these

functions.

Each function has a unique id, start from 0 to n-1. A function may be called recursively or by another function.

A log is a string has this format : function\_id:start\_or\_end:timestamp. For example, "0:start:0" means function 0 starts from the very

beginning of time 0. "0:end:0" means function 0 ends to the very end of time 0.

Exclusive time of a function is defined as the time spent within this function, the time spent by calling other functions should not

be considered as this function's exclusive time. You should return the exclusive time of each function sorted by their function id.

Example 1:

Input:

n = 2

logs =

["0:start:0",

"1:start:2",

"1:end:5",

"0:end:6"]

Output:[3, 4]

Explanation:

Function 0 starts at time 0, then it executes 2 units of time and reaches the end of time 1.

Now function 0 calls function 1, function 1 starts at time 2, executes 4 units of time and end at time 5.

Function 0 is running again at time 6, and also end at the time 6, thus executes 1 unit of time.

So function 0 totally execute 2 + 1 = 3 units of time, and function 1 totally execute 4 units of time.

Note:

Input logs will be sorted by timestamp, NOT log id.

Your output should be sorted by function id, which means the 0th element of your output corresponds to the exclusive time of function 0.

Two functions won't start or end at the same time.

Functions could be called recursively, and will always end.

1 <= n <= 100

class Solution {

public int[] exclusiveTime(int n, List<String> logs) {

Stack<Integer> stack = new Stack<>();

int[] res = new int[n];

String[] s = logs.get(0).split(":");

int prev = Integer.parseInt(s[2]);

int i = 0;

while (i < logs.size()){

String[] str = logs.get(i).split(":");

if (str[1].equals("start")){

if (!stack.isEmpty()){

res[stack.peek()] += Integer.parseInt(str[2]) - prev;

}

stack.push(Integer.parseInt(str[0]));

prev = Integer.parseInt(str[2]);

}else{

res[stack.pop()] += Integer.parseInt(str[2]) - prev + 1;

prev = Integer.parseInt(str[2]) + 1;

}

i++;

}

return res;

}

}

Better version:

Note that job id is stacked, so we created stack to store job id.

Then we need to calculate the duration, so we use prev to store previous timestamp

class Solution {

public int[] exclusiveTime(int n, List<String> logs) {

int[] res = new int[n];

Stack<Integer> stack = new Stack<>();//store id, not timestamp because id is being stacked not timestamp

int prev = 0;//store timestamp

for (String log : logs){

String[] strs = log.split(":");

int id = Integer.parseInt(strs[0]);

int curr = Integer.parseInt(strs[2]);

if (strs[1].equals("start")){

if (!stack.isEmpty()){

res[stack.peek()] += curr - prev;

}

stack.push(id);

prev = curr;

}else{

res[stack.pop()] += curr - prev + 1;

prev = curr + 1;

}

}

return res;

}

}

## 637\_Average.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty binary tree, return the average value of the nodes on each level in the form of an array.

Example 1:

Input:

3

/ \

9 20

/ \

15 7

Output: [3, 14.5, 11]

Explanation:

The average value of nodes on level 0 is 3, on level 1 is 14.5, and on level 2 is 11. Hence return [3, 14.5, 11].

Note:

The range of node's value is in the range of 32-bit signed integer.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

Method 1: BFS

class Solution {

public List<Double> averageOfLevels(TreeNode root) {

List<Double> res = new ArrayList<>();

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

int size = queue.size();

double sum = 0.0;

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

sum += node.val;

if (node.left != null){

queue.offer(node.left);

}

if (node.right != null){

queue.offer(node.right);

}

}

res.add(sum / size);

}

return res;

}

}

Method 2: DFS

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

class Node {

double sum;

int count;

public Node (double sum, int count){

this.sum = sum;

this.count = count;

}

}

public List<Double> averageOfLevels(TreeNode root) {

List<Double> res = new ArrayList<>();

List<Node> list = new ArrayList<>();

dfs(root, list, 0);

for (int i = 0; i < list.size(); i++){

res.add(list.get(i).sum / list.get(i).count);

}

return res;

}

private void dfs(TreeNode root, List<Node> list, int level){

if (root == null){

return;

}

if (list.size() == level){

list.add(new Node(root.val, 1));

}else{

list.get(level).sum += root.val;

list.get(level).count++;

}

dfs(root.left, list, level + 1);

dfs(root.right, list, level + 1);

}

}

## 637\_CheckWordAbbreviation.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty string word and an abbreviation abbr, return whether the string matches with the given abbreviation.

A string such as "word" contains only the following valid abbreviations:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Notice

Notice that only the above abbreviations are valid abbreviations of the string word. Any other string is not a

valid abbreviation of word.

Have you met this question in a real interview? Yes

Example

Example 1:

Given s = "internationalization", abbr = "i12iz4n":

Return true.

Example 2:

Given s = "apple", abbr = "a2e":

Return false.

https://github.com/optimisea/Leetcode/edit/master/Java/408\_Valid.java

public class Solution {

/\*

\* @param word: a non-empty string

\* @param abbr: an abbreviation

\* @return: true if string matches with the given abbr or false

\*/

public boolean validWordAbbreviation(String word, String abbr) {

char[] wordChar = word.toCharArray();

char[] abbrChar = abbr.toCharArray();

int i = 0, j = 0;

while (i < wordChar.length && j < abbrChar.length){

if (Character.isDigit(abbrChar[j])){

if (abbrChar[j] == '0'){

return false;

}

int val = 0;

while (j < abbrChar.length && Character.isDigit(abbrChar[j])){

val = val \* 10 + abbrChar[j] - '0';

j++;

}

i += val;

}else{

if (wordChar[i++] != abbrChar[j++]){

return false;

}

}

}

return i == wordChar.length;

}

}

## 638\_Shopping.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In LeetCode Store, there are some kinds of items to sell. Each item has a price.

However, there are some special offers, and a special offer consists of one or more different kinds of items with a sale price.

You are given the each item's price, a set of special offers, and the number we need to buy for each item. The job is to output the lowest price you have to pay for exactly certain items as given, where you could make optimal use of the special offers.

Each special offer is represented in the form of an array, the last number represents the price you need to pay for this special offer, other numbers represents how many specific items you could get if you buy this offer.

You could use any of special offers as many times as you want.

Example 1:

Input: [2,5], [[3,0,5],[1,2,10]], [3,2]

Output: 14

Explanation:

There are two kinds of items, A and B. Their prices are $2 and $5 respectively.

In special offer 1, you can pay $5 for 3A and 0B

In special offer 2, you can pay $10 for 1A and 2B.

You need to buy 3A and 2B, so you may pay $10 for 1A and 2B (special offer #2), and $4 for 2A.

Example 2:

Input: [2,3,4], [[1,1,0,4],[2,2,1,9]], [1,2,1]

Output: 11

Explanation:

The price of A is $2, and $3 for B, $4 for C.

You may pay $4 for 1A and 1B, and $9 for 2A ,2B and 1C.

You need to buy 1A ,2B and 1C, so you may pay $4 for 1A and 1B (special offer #1), and $3 for 1B, $4 for 1C.

You cannot add more items, though only $9 for 2A ,2B and 1C.

Note:

There are at most 6 kinds of items, 100 special offers.

For each item, you need to buy at most 6 of them.

You are not allowed to buy more items than you want, even if that would lower the overall price.

Method: best solution:

class Solution {

public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List<Integer> needs) {

int n = price.size();

int res = Integer.MAX\_VALUE;

for (List<Integer> list : special){

boolean pass = true;

for (int i = 0; i < n; i++){

if (list.get(i) > needs.get(i)){

pass = false;

break;

}

}

if (pass){

List<Integer> newNeed = new ArrayList<>();

for (int i = 0; i < n; i++){

newNeed.add(needs.get(i) - list.get(i));

}

res = Math.min(res, shoppingOffers(price, special, newNeed) + list.get(list.size() - 1));

}

}

int sum = 0;

for (int i = 0; i < n; i++){

sum += price.get(i) \* needs.get(i);

}

res = Math.min(res, sum);

return res;

}

}

With memo

class Solution {

Map<String, Integer> map = new HashMap<>();

public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List<Integer> needs) {

StringBuilder sb = new StringBuilder();

for (int i : needs){

sb.append(i + ":");

}

String key = sb.toString();

if (map.containsKey(key)){

return map.get(key);

}

int n = price.size();

int res = Integer.MAX\_VALUE;

for (List<Integer> list : special){

boolean pass = true;

for (int i = 0; i < n; i++){

if (list.get(i) > needs.get(i)){

pass = false;

break;

}

}

if (pass){

List<Integer> newNeed = new ArrayList<>();

for (int i = 0; i < n; i++){

newNeed.add(needs.get(i) - list.get(i));

}

res = Math.min(res, shoppingOffers(price, special, newNeed) + list.get(list.size() - 1));

}

}

int sum = 0;

for (int i = 0; i < n; i++){

sum += price.get(i) \* needs.get(i);

}

res = Math.min(res, sum);

map.put(key, res);

return res;

}

}

Method: DFS + memo

class Solution {

public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List<Integer> needs) {

Map<List<Integer>, Integer> map = new HashMap<>();

return dfs(price, special, needs, map);

}

private int dfs(List<Integer> price, List<List<Integer>> special, List<Integer> needs, Map<List<Integer>, Integer> map){

if (map.containsKey(needs)){

return map.get(needs);

}

int res = dot(price, needs);

for (int i = 0; i < special.size(); i++){

int j = 0;

List<Integer> clone = new ArrayList<Integer>(needs);

for (j = 0; j < needs.size(); j++){

if (clone.get(j) < special.get(i).get(j)){

break;

}

clone.set(j, clone.get(j) - special.get(i).get(j));

}

if (j == needs.size()){

res = Math.min(res, special.get(i).get(j) + dfs(price, special, clone, map));

}

}

map.put(needs, res);

return res;

}

private int dot(List<Integer> price, List<Integer> needs){

int sum = 0;

for (int i = 0; i < price.size(); i++){

sum += price.get(i) \* needs.get(i);

}

return sum;

}

}

Method

backtracking: TLE

class Solution {

int min = Integer.MAX\_VALUE;

public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List<Integer> needs) {

for (int i = 0; i < price.size(); i++){

List<Integer> temp = new ArrayList<>();

for (int j = 0; j < price.size() + 1; j++){

if (i == j){

temp.add(1);

}else if (j == price.size()){

temp.add(price.get(i));

}else{

temp.add(0);

}

}

special.add(temp);

}

dfs(special, needs, new ArrayList<>(), 0);

return min;

}

private void dfs(List<List<Integer>> special, List<Integer> needs, List<Integer> candidate, int sum){

if (match(special, needs, candidate)){

min = Math.min(min, sum);

return;

}

for (int i = 0; i < special.size(); i++){

boolean useSpecial = true;

for (int j = 0; j < needs.size(); j++){

if (needs.get(j) < special.get(i).get(j)){

useSpecial = false;

break;

}

}

if (useSpecial){

candidate.add(i);

dfs(special, needs, candidate, sum + special.get(i).get(needs.size()));

candidate.remove(candidate.size() - 1);

}

}

}

private boolean match(List<List<Integer>> special, List<Integer> needs, List<Integer> candidate){

for (int i = 0; i < needs.size(); i++){

int sum = 0;

for (int j = 0; j < candidate.size(); j++){

sum += special.get(candidate.get(j)).get(i);

}

if (needs.get(i) != sum){

return false;

}

}

return true;

}

}

## 638\_StringsHomomorphism.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings s and t, determine if they are isomorphic.

Two strings are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order

of characters. No two characters may map to the same character but a character may map to itself.

Notice

You may assume both s and t have the same length.

Have you met this question in a real interview? Yes

Example

Given s = "egg", t = "add", return true.

Given s = "foo", t = "bar", return false.

Given s = "paper", t = "title", return true.

Method 1: Array HashMap

public class Solution {

/\*

\* @param s: a string

\* @param t: a string

\* @return: true if the characters in s can be replaced to get t or false

\*/

public boolean isIsomorphic(String s, String t) {

final int MAX\_CHAR = 256;

int[] map1 = new int[MAX\_CHAR];

char[] sc = s.toCharArray();

char[] tc = t.toCharArray();

for (int i = 0; i < s.length(); i++){

if (map1[sc[i]] == 0){

map1[sc[i]] = tc[i];

}else{

if (map1[sc[i]] != tc[i]){

return false;

}

}

}

int[] map2 = new int[MAX\_CHAR];

for (int i = 0; i < t.length(); i++){

if (map2[tc[i]] == 0){

map2[tc[i]] = sc[i];

}else{

if (map2[tc[i]] != sc[i]){

return false;

}

}

}

return true;

}

}

Method 2: HashMap

public class Solution {

/\*

\* @param s: a string

\* @param t: a string

\* @return: true if the characters in s can be replaced to get t or false

\*/

public boolean isIsomorphic(String s, String t) {

HashMap<Character, Character> map1 = new HashMap<>();

HashMap<Character, Character> map2 = new HashMap<>();

for (int i = 0; i < s.length(); i++){

if (!map1.containsKey(s.charAt(i))){

map1.put(s.charAt(i), t.charAt(i));

}else{

if (map1.get(s.charAt(i)) != t.charAt(i)){

return false;

}

}

}

for (int i = 0; i < t.length(); i++){

if (!map2.containsKey(t.charAt(i))){

map2.put(t.charAt(i), s.charAt(i));

}else{

if (map2.get(t.charAt(i)) != s.charAt(i)){

return false;

}

}

}

return true;

}

}

## 639\_Decode.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A message containing letters from A-Z is being encoded to numbers using the following mapping way:

'A' -> 1

'B' -> 2

...

'Z' -> 26

Beyond that, now the encoded string can also contain the character '\*', which can be treated as one of the numbers from 1 to 9.

Given the encoded message containing digits and the character '\*', return the total number of ways to decode it.

Also, since the answer may be very large, you should return the output mod 109 + 7.

Example 1:

Input: "\*"

Output: 9

Explanation: The encoded message can be decoded to the string: "A", "B", "C", "D", "E", "F", "G", "H", "I".

Example 2:

Input: "1\*"

Output: 9 + 9 = 18

Note:

The length of the input string will fit in range [1, 105].

The input string will only contain the character '\*' and digits '0' - '9'.

class Solution {

public int numDecodings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int n = s.length();

long[] dp = new long[n+1];

dp[0] = 1;

if (s.charAt(0) == '0'){

dp[1] = 0;

}else if (s.charAt(0) == '\*'){

dp[1] = 9;

}else{

dp[1] = 1;

}

for (int i = 2; i <= n; i++){

char first = s.charAt(i-2);

char second = s.charAt(i-1);

if (second != '0'){

if (second == '\*'){

dp[i] += dp[i-1] \* 9;

}else{

dp[i] += dp[i-1];

}

}

if (first != '\*' && second != '\*'){

int twoDigits = (first - '0') \* 10 + (second - '0');

if (twoDigits <= 26 && twoDigits >= 10){

dp[i] += dp[i-2];

}

}else if (first == '\*' && second != '\*'){

if (second - '0' <= 6){

dp[i] += dp[i-2] \* 2;

}else{

dp[i] += dp[i-2];

}

}else if (first != '\*' && second == '\*'){

if (first - '0' == 2){

dp[i] += dp[i-2] \* 6;

}else if (first -'0' == 1){

dp[i] += dp[i-2] \* 9;

}

}else{

dp[i] += dp[i-2] \* 15;

}

dp[i] %= 1000000007;

}

return (int)(dp[n]);

}

}

https://leetcode.com/problems/decode-ways-ii/discuss/105258/Java-O(N)-by-General-Solution-for-all-DP-problems

class Solution {

public int numDecodings(String s) {

int mod = (int)Math.pow(10, 9) + 7;

int n = s.length();

if (n == 0){

return 0;

}

if (n == 1){

if (s.charAt(0) == '0'){

return 0;

}else if (s.charAt(0) == '\*'){

return 9;

}else{

return 1;

}

}

if (s.charAt(0) == '0'){

return 0;

}

long[] dp = new long[n+1];

dp[0] = 1;

dp[1] = s.charAt(0) == '\*' ? 9 : 1;

for (int i = 2; i <= n; i++){

char first = s.charAt(i-1);

char second = s.charAt(i-2);

if (first != '0'){

if (first == '\*'){

dp[i] += dp[i-1] \* 9;

}else{

dp[i] += dp[i-1];

}

}

if (first != '\*' && second != '\*'){

int curr = (int)(first - '0');

int prev = (int)(second - '0');

int num = prev \* 10 + curr;

if (num >= 10 && num <= 26){

dp[i] += dp[i-2];

}

}else if (first == '\*' && second != '\*'){

if (second - '0' == 2){

dp[i] += dp[i-2] \* 6;

}else if (second -'0' == 1){

dp[i] += dp[i-2] \* 9;

}

}else if (first != '\*' && second == '\*'){

if (first - '0' <= 6){

dp[i] += dp[i-2] \* 2;

}else{

dp[i] += dp[i-2];

}

}else{

dp[i] += dp[i-2] \* 15;//can't be 20 so 26-11+1-1 because \* can't be 0

}

dp[i] %= mod;

}

return (int)dp[n];

}

}

## 639\_WordsAbbreviation.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of n distinct non-empty strings, you need to generate minimal possible abbreviations for every

word following rules below.

Begin with the first character and then the number of characters abbreviated, which followed by the last character.

If there are any conflict, that is more than one words share the same abbreviation,

a longer prefix is used instead of only the first character until making the map from word to abbreviation become unique.

In other words, a final abbreviation cannot map to more than one original words.

If the abbreviation doesn't make the word shorter, then keep it as original.

Both n and the length of each word will not exceed 400.

The length of each word is greater than 1.

The words consist of lowercase English letters only.

The return answers should be in the same order as the original array.

Have you met this question in a real interview? Yes

Example

Given dict = ["like", "god", "internal", "me", "internet", "interval", "intension", "face", "intrusion"]

return ["l2e","god","internal","me","i6t","interval","inte4n","f2e","intr4n"]

public class Solution {

/\*

\* @param dict: an array of n distinct non-empty strings

\* @return: an array of minimal possible abbreviations for every word

\*/

public String[] wordsAbbreviation(String[] dict) {

int len = dict.length;

String[] result = new String[len];

int[] prefixNum = new int[len];

HashMap<String, Integer> map = new HashMap<>();

for (int i = 0; i < len; i++){

prefixNum[i] = 1;

result[i] = getAbbr(dict[i], 1);

if (!map.containsKey(result[i])){

map.put(result[i], 1);

}else{

map.put(result[i], map.get(result[i]) + 1);

}

}

boolean unique = false;

while (!unique){

unique = true;

for (int i = 0; i < len; i++){

if (map.get(result[i]) > 1){

prefixNum[i]++;

result[i] = getAbbr(dict[i], prefixNum[i]);

if (!map.containsKey(result[i])){

map.put(result[i], 1);

}else{

map.put(result[i], map.get(result[i]) + 1);

}

unique = false;

}

}

}

return result;

}

private String getAbbr(String str, int prefixNum){

if (prefixNum >= str.length() - 2){

return str;

}

String ans = str.substring(0, prefixNum) + (str.length() - prefixNum - 1)

+ str.charAt(str.length() - 1);

return ans;

}

}

## 64\_MinimumPathSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right

which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example 1:

[[1,3,1],

[1,5,1],

[4,2,1]]

Given the above grid map, return 7. Because the path 1→3→1→1→1 minimizes the sum.

class Solution {

public int minPathSum(int[][] grid) {

if (grid == null || grid.length == 0 || grid[0].length == 0){

return 0;

}

int n = grid.length;

int m = grid[0].length;

int[][] f = new int[n][m];

f[0][0] = grid[0][0];

for (int i = 1; i < n; i++){

f[i][0] = f[i-1][0] + grid[i][0];

}

for (int j = 1; j < m; j++){

f[0][j] = f[0][j-1] + grid[0][j];

}

for (int i = 1; i < n; i++){

for (int j = 1; j < m; j++){

f[i][j] = grid[i][j] + Math.min(f[i][j-1], f[i-1][j]);

}

}

return f[n-1][m-1];

}

}

## 640\_EditDistanceII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings S and T, determine if they are both one edit distance apart.

Have you met this question in a real interview? Yes

Example

Given s = "aDb", t = "adb"

return true

public class Solution {

/\*

\* @param s: a string

\* @param t: a string

\* @return: true if they are both one edit distance apart or false

\*/

public boolean isOneEditDistance(String s, String t) {

if (s == null || t == null){

return false;

}

if (s.length() > t.length()){

String temp = s;

s = t;

t = temp;

}

int slen = s.length();

int tlen = t.length();

int diff = tlen - slen;

if (diff > 1){

return false;

}

if (diff == 0){

int count = 0;

for (int i = 0; i < slen; i++){

if (s.charAt(i) != t.charAt(i)){

count++;

}

}

return (count == 1);

}

if (diff == 1){

for (int i = 0; i < slen; i++){

if (s.charAt(i) != t.charAt(i)){

return s.substring(i).equals(t.substring(i+1));

}

}

return true;

}

return true;

}

}

## 641\_DesignDeque.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design your implementation of the circular double-ended queue (deque).

Your implementation should support following operations:

MyCircularDeque(k): Constructor, set the size of the deque to be k.

insertFront(): Adds an item at the front of Deque. Return true if the operation is successful.

insertLast(): Adds an item at the rear of Deque. Return true if the operation is successful.

deleteFront(): Deletes an item from the front of Deque. Return true if the operation is successful.

deleteLast(): Deletes an item from the rear of Deque. Return true if the operation is successful.

getFront(): Gets the front item from the Deque. If the deque is empty, return -1.

getRear(): Gets the last item from Deque. If the deque is empty, return -1.

isEmpty(): Checks whether Deque is empty or not.

isFull(): Checks whether Deque is full or not.

Example:

MyCircularDeque circularDeque = new MycircularDeque(3); // set the size to be 3

circularDeque.insertLast(1); // return true

circularDeque.insertLast(2); // return true

circularDeque.insertFront(3); // return true

circularDeque.insertFront(4); // return false, the queue is full

circularDeque.getRear(); // return 32

circularDeque.isFull(); // return true

circularDeque.deleteLast(); // return true

circularDeque.insertFront(4); // return true

circularDeque.getFront(); // return 4

Note:

All values will be in the range of [0, 1000].

The number of operations will be in the range of [1, 1000].

Please do not use the built-in Deque library.

class MyCircularDeque {

private int[] queue;

private int next;

/\*\* Initialize your data structure here. Set the size of the deque to be k. \*/

public MyCircularDeque(int k) {

queue = new int[k];

next = 0;

}

/\*\* Adds an item at the front of Deque. Return true if the operation is successful. \*/

public boolean insertFront(int value) {

if (isFull()){

return false;

}

if (!isEmpty()){

for (int i = next; i > 0; i--){

queue[i] = queue[i-1];

}

}

queue[0] = value;

next++;

return true;

}

/\*\* Adds an item at the rear of Deque. Return true if the operation is successful. \*/

public boolean insertLast(int value) {

if (isFull()){

return false;

}

queue[next] = value;

next++;

return true;

}

/\*\* Deletes an item from the front of Deque. Return true if the operation is successful. \*/

public boolean deleteFront() {

if (isEmpty()){

return false;

}

for (int i = 0; i < next - 1; i++){

queue[i] = queue[i+1];

}

queue[next-1] = 0;

next--;

return true;

}

/\*\* Deletes an item from the rear of Deque. Return true if the operation is successful. \*/

public boolean deleteLast() {

if (isEmpty()){

return false;

}

queue[next-1] = 0;

next--;

return true;

}

/\*\* Get the front item from the deque. \*/

public int getFront() {

if (isEmpty()){

return -1;

}

return queue[0];

}

/\*\* Get the last item from the deque. \*/

public int getRear() {

if (isEmpty()){

return -1;

}

return queue[next-1];

}

/\*\* Checks whether the circular deque is empty or not. \*/

public boolean isEmpty() {

return next == 0;

}

/\*\* Checks whether the circular deque is full or not. \*/

public boolean isFull() {

return next == queue.length;

}

}

/\*\*

\* Your MyCircularDeque object will be instantiated and called as such:

\* MyCircularDeque obj = new MyCircularDeque(k);

\* boolean param\_1 = obj.insertFront(value);

\* boolean param\_2 = obj.insertLast(value);

\* boolean param\_3 = obj.deleteFront();

\* boolean param\_4 = obj.deleteLast();

\* int param\_5 = obj.getFront();

\* int param\_6 = obj.getRear();

\* boolean param\_7 = obj.isEmpty();

\* boolean param\_8 = obj.isFull();

\*/

# 641\_MissingInterval.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted integer array where the range of elements are in the inclusive range [lower, upper], return its missing ranges.

Have you met this question in a real interview? Yes

Example

Given nums = [0, 1, 3, 50, 75], lower = 0 and upper = 99

return ["2", "4->49", "51->74", "76->99"].

思路:

• 简单的模拟题

– 两端点和一头一尾形成的区间 + for循环扫描中间形成的区间 – 利用函数让自己的代码更简洁(见代码)

• 特殊输入?

– 实现时可能出现中间值超过int范围

public class Solution {

/\*

\* @param nums: a sorted integer array

\* @param lower: An integer

\* @param upper: An integer

\* @return: a list of its missing ranges

\*/

public List<String> findMissingRanges(int[] nums, int lower, int upper) {

List<String> result = new ArrayList<String>();

if (nums == null || nums.length == 0){

getRange(result, lower, upper);

return result;

}

if (nums[0] != Integer.MIN\_VALUE){

getRange(result, lower, nums[0] - 1);

}

for (int i = 1; i < nums.length; i++){

if (nums[i-1] != Integer.MAX\_VALUE && nums[i] != Integer.MIN\_VALUE){

getRange(result, nums[i-1] + 1, nums[i] - 1);

}

}

if (nums[nums.length - 1] != Integer.MAX\_VALUE){

getRange(result, nums[nums.length - 1] + 1, upper);

}

return result;

}

private void getRange(List<String> result, int lower, int upper){

if (lower > upper){

return;

}else if (lower == upper){

result.add(String.valueOf(lower));

}else{

result.add(lower + "->" + upper);

}

}

}

# 642\_Design.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a search autocomplete system for a search engine. Users may input a sentence (at least one word and end with a special character

'#'). For each character they type except '#', you need to return the top 3 historical hot sentences that have prefix the same as

the part of sentence already typed. Here are the specific rules:

The hot degree for a sentence is defined as the number of times a user typed the exactly same sentence before.

The returned top 3 hot sentences should be sorted by hot degree (The first is the hottest one). If several sentences have the

same degree of hot, you need to use ASCII-code order (smaller one appears first).

If less than 3 hot sentences exist, then just return as many as you can.

When the input is a special character, it means the sentence ends, and in this case, you need to return an empty list.

Your job is to implement the following functions:

The constructor function:

AutocompleteSystem(String[] sentences, int[] times): This is the constructor. The input is historical data. Sentences is a string

array consists of previously typed sentences. Times is the corresponding times a sentence has been typed. Your system should

record these historical data.

Now, the user wants to input a new sentence. The following function will provide the next character the user types:

List<String> input(char c): The input c is the next character typed by the user. The character will only be lower-case letters

('a' to 'z'), blank space (' ') or a special character ('#'). Also, the previously typed sentence should be recorded in your system.

The output will be the top 3 historical hot sentences that have prefix the same as the part of sentence already typed.

Example:

Operation: AutocompleteSystem(["i love you", "island","ironman", "i love leetcode"], [5,3,2,2])

The system have already tracked down the following sentences and their corresponding times:

"i love you" : 5 times

"island" : 3 times

"ironman" : 2 times

"i love leetcode" : 2 times

Now, the user begins another search:

Operation: input('i')

Output: ["i love you", "island","i love leetcode"]

Explanation:

There are four sentences that have prefix "i". Among them, "ironman" and "i love leetcode" have same hot degree.

Since ' ' has ASCII code 32 and 'r' has ASCII code 114, "i love leetcode" should be in front of "ironman". Also we only need

to output top 3 hot sentences, so "ironman" will be ignored.

Operation: input(' ')

Output: ["i love you","i love leetcode"]

Explanation:

There are only two sentences that have prefix "i ".

Operation: input('a')

Output: []

Explanation:

There are no sentences that have prefix "i a".

Operation: input('#')

Output: []

Explanation:

The user finished the input, the sentence "i a" should be saved as a historical sentence in system. And the following input

will be counted as a new search.

Note:

The input sentence will always start with a letter and end with '#', and only one blank space will exist between two words.

The number of complete sentences that to be searched won't exceed 100. The length of each sentence including those in the historical data won't exceed 100.

Please use double-quote instead of single-quote when you write test cases even for a character input.

Please remember to RESET your class variables declared in class AutocompleteSystem, as static/class variables are persisted across multiple test cases. Please see here for more details.

Idea:

Only thing more than a normal Trie is added a map of sentence to count in each of the

Trie node to facilitate process of getting top 3 results.

class AutocompleteSystem {

class TrieNode {

Map<Character, TrieNode> children;

Map<String, Integer> counts;

boolean isEnd;

public TrieNode(){

children = new HashMap<Character, TrieNode>();

counts = new HashMap<String, Integer>();

isEnd = false;

}

}

class Pair {

String s;

int count;

public Pair (String s, int count){

this.s = s;

this.count = count;

}

}

TrieNode root;

String prefix;

public AutocompleteSystem(String[] sentences, int[] times) {

root = new TrieNode();

prefix = "";

for (int i = 0; i < sentences.length; i++){

add(sentences[i], times[i]);

}

}

private void add(String sentence, int count){

TrieNode curr = root;

for (char c : sentence.toCharArray()){

TrieNode next = curr.children.get(c);

if (next == null){

next = new TrieNode();

curr.children.put(c, next);

}

curr = next;

curr.counts.put(sentence, curr.counts.getOrDefault(sentence, 0) + count);

}

curr.isEnd = true;

}

public List<String> input(char c) {

if (c == '#'){

add(prefix, 1);

prefix = "";

return new ArrayList<String>();

}

TrieNode curr = root;

prefix = prefix + c;

for (char ch : prefix.toCharArray()){

TrieNode next = curr.children.get(ch);

if (next == null){

return new ArrayList<String>();

}

curr = next;

}

Queue<Pair> pq = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

if (p1.count == p2.count){

return p1.s.compareTo(p2.s);// compareTo for String comparation

}

return p2.count - p1.count;

}

});

for (String str : curr.counts.keySet()){

pq.offer(new Pair(str, curr.counts.get(str)));

}

List<String> res = new ArrayList<>();

for (int i = 0; i < 3 && !pq.isEmpty(); i++){

res.add(pq.poll().s);

}

return res;

}

}

/\*\*

\* Your AutocompleteSystem object will be instantiated and called as such:

\* AutocompleteSystem obj = new AutocompleteSystem(sentences, times);

\* List<String> param\_1 = obj.input(c);

\*/

# 642\_SlidingWindowAveragefromDataStream.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

Have you met this question in a real interview? Yes

Example

MovingAverage m = new MovingAverage(3);

m.next(1) = 1 // return 1.00000

m.next(10) = (1 + 10) / 2 // return 5.50000

m.next(3) = (1 + 10 + 3) / 3 // return 4.66667

m.next(5) = (10 + 3 + 5) / 3 // return 6.00000

Method:

• 方便快速求一段的和

– a[k] + a[k + 1] +... + a[j] = s[j] - s[k -1] 时间复杂度o(1)

• 怎样快求s[i] ?

– s[i] = s[i - 1] + a[i] 时间复杂度o(1)

如何节省储存空间呢?(2种方法)

1. 链表保存sum

2. 数组滚动

• sum[4]-sum[1]时,sum[0]这个位置空出来了,sum[4]可以放到sum[0] • 这样算实际位置

– 逻辑位置取mod,这一题mod (size+1)

 小技巧总结:

• 如何快速求和? 前缀和数组(dummy 0)

• 如何节省储存空间呢? 滚动

• 写滚动的技巧 先写程序最后加滚动

public class MovingAverage {

private int size;

private double[] sum;

private int id;

/\*

\* @param size: An integer

\*/public MovingAverage(int size) {

this.size = size;

id = 0;

sum = new double[size + 1];

}

/\*

\* @param val: An integer

\* @return:

\*/

public double next(int val) {

id++;

sum[mod(id)] = sum[mod(id-1)] + val;

if (id >= size){

return (sum[mod(id)] - sum[mod(id - size)]) / size;

}else{

return sum[mod(id)] / id;

}

}

private int mod(int k){

return k % (size + 1);

}

}

# 643\_MaximumI.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array consisting of n integers, find the contiguous subarray of given length k

that has the maximum average value. And you need to output the maximum average value.

Example 1:

Input: [1,12,-5,-6,50,3], k = 4

Output: 12.75

Explanation: Maximum average is (12-5-6+50)/4 = 51/4 = 12.75

Note:

1 <= k <= n <= 30,000.

Elements of the given array will be in the range [-10,000, 10,000].

Method 1: Sliding window

class Solution {

public double findMaxAverage(int[] nums, int k) {

int sum = 0;

for (int i = 0; i < k; i++){

sum += nums[i];

}

int max = sum;

for (int i = k; i < nums.length; i++){

sum += nums[i] - nums[i-k];

max = Math.max(max, sum);

}

return (double) max / k;

}

}

Method 2: Prefix Sum

class Solution {

public double findMaxAverage(int[] nums, int k) {

int[] preSum = new int[nums.length+1];

preSum[0] = 0;

for(int i = 0; i < nums.length; i++){

preSum[i+1] = preSum[i] + nums[i];

}

int max = Integer.MIN\_VALUE;

for (int i = k; i < preSum.length; i++){

max = Math.max(max, preSum[i] - preSum[i-k]);

}

return (double)(max) / k;

}

}

class Solution {

public double findMaxAverage(int[] nums, int k) {

int sum = 0;

int max = Integer.MIN\_VALUE;

for (int i = 0; i < nums.length; i++){

sum += nums[i];

if (i - k >= 0){

sum -= nums[i-k];

}

if (i >= k -1){

max = Math.max(max, sum);

}

}

return (double)max / k;

}

}

# 643\_SystemLongestFilePath.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Suppose we abstract our file system by a string in the following manner:

The string "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" represents:

dir

subdir1

subdir2

file.ext

The directory dir contains an empty sub-directory subdir1 and a sub-directory subdir2 containing a file file.ext.

The string

"dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsubdir2\n\t\t\tfile2.ext"

represents:

dir

subdir1

file1.ext

subsubdir1

subdir2

subsubdir2

file2.ext

The directory dir contains two sub-directories subdir1 and subdir2. subdir1 contains a file file1.ext and an empty second-level sub-directory subsubdir1. subdir2 contains a second-level sub-directory subsubdir2 containing a file file2.ext.

We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the second example above, the longest absolute path is "dir/subdir2/subsubdir2/file2.ext", and its length is 32 (not including the double quotes).

Given a string representing the file system in the above format, return the length of the longest absolute path to file in the abstracted file system. If there is no file in the system, return 0.

Notice

The name of a file contains at least a . and an extension.

The name of a directory or sub-directory will not contain a ..

Time complexity required: O(n) where n is the size of the input string.

Notice that a/aa/aaa/file1.txt is not the longest file path, if there is another path aaaaaaaaaaaaaaaaaaaaa/sth.png.

Have you met this question in a real interview? Yes

Example

Give input = "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" return

思路:

• 有点小麻烦的纯模拟

• 技巧一:用split(‘\n’) 将原串分割开,相当于一次读一行

• 技巧二:利用’\t’的个数来当前目录/文件 在第几层

• 技巧三:从上到下一行一行读入的顺序像dfs的顺序,所以可以把前面 几层的字符串长度都记下来(画图模拟)

Note that consider "\t" as a whole, so the following output is 2.

import java.io.\*;

class MyCode {

public static void main (String[] args) {

String Str = new String("\t\t\t");

System.out.print("Found Last Index :" );

System.out.println(Str.lastIndexOf("\t")); }

}

public class Solution {

/\*

\* @param input: an abstract file system

\* @return: return the length of the longest absolute path to file

\*/

public int lengthLongestPath(String input) {

if (input == null || input.length() == 0){

return 0;

}

int ans = 0;

int[] level\_size = new int[input.length()+1];

for (String line : input.split("\n")){

int level = line.lastIndexOf("\t") + 2; //can't find, return -1; if found, "\t" together as one index

int len = line.length() - (level - 1);

if (line.contains(".")){

ans = Math.max(ans, level\_size[level - 1] + len);

}else{

level\_size[level] = level\_size[level - 1] + len + 1;

}

}

return ans;

}

}

# 644\_MaximumII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array consisting of n integers, find the contiguous subarray of given length k that

has the maximum average value. And you need to output the maximum average value.

Example 1:

Input: [1,12,-5,-6,50,3], k = 4

Output: 12.75

Explanation: Maximum average is (12-5-6+50)/4 = 51/4 = 12.75

Note:

1 <= k <= n <= 30,000.

Elements of the given array will be in the range [-10,000, 10,000].

Method 1: sliding window in check function

public class Solution {

/\*

\* @param nums: an array with positive and negative numbers

\* @param k: an integer

\* @return: the maximum average

\*/

public double maxAverage(int[] nums, int k) {

double max = Integer.MIN\_VALUE;

double min = Integer.MAX\_VALUE;

for (int i = 0; i < nums.length; i++){

max = Math.max(max, nums[i]);

min = Math.min(min, nums[i]);

}

double error = Integer.MAX\_VALUE;

while (error > 1e-6){

double mid = (max + min) / 2.0;

if (checkLargerThanMid(nums, mid, k)){

min = mid;

}else{

max = mid;

}

error = max - min;

}

return min;

}

private boolean checkLargerThanMid(int[] nums, double mid, int k){

int n = nums.length;

double sum = 0;

double min = 0.0;

double prev = 0.0;

for (int i = 0; i < k; i++){

sum += nums[i] - mid;

}

if (sum >= 0){

return true;

}

for (int i = k; i < n; i++){

sum += nums[i] - mid;

prev += nums[i-k] - mid;

min = Math.min(min, prev);

if (sum - min >= 0){

return true;

}

}

return false;

}

}

Method 2: prefix sum in check function

public class Solution {

/\*\*

\* @param nums an array with positive and negative numbers

\* @param k an integer

\* @return the maximum average

\*/

public double maxAverage(int[] nums, int k) {

// Write your code here

double l = Integer.MAX\_VALUE, r = Integer.MIN\_VALUE;

for (int i = 0; i < nums.length; ++i) {

if (nums[i] < l)

l = nums[i];

if (nums[i] > r)

r = nums[i];

}

while (r - l >= 1e-6) {

double mid = (l + r) / 2.0;

if (check\_valid(nums, mid, k)) {

l = mid;

}

else {

r = mid;

}

}

return l;

}

private boolean check\_valid(int nums[], double mid, int k) {

int n = nums.length;

double min\_pre = 0;

double[] sum = new double[n + 1];

sum[0] = 0;

for (int i = 1; i <= n; ++i) {

sum[i] = sum[i - 1] + nums[i - 1] - mid;

if (i >= k && sum[i] - min\_pre >= 0) {

return true;

}

if (i >= k)

min\_pre = Math.min(min\_pre, sum[i - k + 1]);

}

return false;

}

}

To understand the idea behind this method, let's look at the following points.

Firstly, we know that the value of the average could lie between the range (min,max)(min, max)(min,max).

Here, minminmin and maxmaxmax refer to the minimum and the maximum values out of the given numsnumsnums array.

This is because, the average can't be lesser than the minimum value and can't be larger than the maximum value.

But, in this case, we need to find the maximum average of a subarray with atleast kkk elements. The idea in this

method is to try to approximate(guess) the solution and to try to find if this solution really exists.

If it exists, we can continue trying to approximate the solution even to a further precise value, but choosing a

larger number as the next approximation. But, if the initial guess is wrong, and the initial maximum average

value(guessed) isn't possible, we need to try with a smaller number as the next approximate.

Now, instead of doing the guesses randomly, we can make use of Binary Search. With minminmin and maxmaxmax

as the initial numbers to begin with, we can find out the midmidmid of these two numbers given by

(min+max)/2(min+max)/2(min+max)/2. Now, we need to find if a subarray with length greater than or equal

to kkk is possible with an average sum greater than this midmidmid value.

To determine if this is possible in a single scan, let's look at an observation. Suppose,

there exist jjj elements, a1,a2,a3...,aja\_1, a\_2, a\_3..., a\_ja​1​​,a​2​​,a​3​​...,a​j​​ in a subarray

within numsnumsnums such that their average is greater than midmidmid. In this case, we can say that

(a1+a2+a3...+aj)/j≥mid

or

(a1+a2+a3...+aj)≥j∗mid

or

(a1−mid)+(a2−mid)+(a3−mid)...+(aj−mid)≥0

Thus, we can see that if after subtracting the midmidmid number from the elements of a subarray with

more than k−1k-1k−1 elements, within numsnumsnums, if the sum of elements of this reduced subarray is

greater than 0, we can achieve an average value greater than midmidmid. Thus, in this case, we need to

set the midmidmid as the new minimum element and continue the process.

Otherwise, if this reduced sum is lesser than 0 for all subarrays with greater than or equal to kkk elements,

we can't achieve midmidmid as the average. Thus, we need to set midmidmid as the new maximum element and

continue the process.

In order to determine if such a subarray exists in a linear manner, we keep on adding nums[i]−midnums[i]-midnums[i]−mid

to the sumsumsum obtained till the ithi^{th}i​th​​ element while traversing over the numsnumsnums array.

If on traversing the first kkk elements, the sumsumsum becomes greater than or equal to 0, we can directly

determine that we can increase the average beyond midmidmid. Otherwise, we continue making additions to sumsumsum

for elements beyond the kthk^{th}k​th​​ element, making use of the following idea.

If we know the cumulative sum upto indices iii and jjj, say sumisum\_isum​i​​ and sumjsum\_jsum​j​​ respectively,

we can determine the sum of the subarray between these indices(including jjj) as sumj−sumisum\_j - sum\_isum​j​​−sum​i​​.

In our case, we want this difference between the cumulative sums to be greater than or equal to 0 as discusssed above.

Further, for sumisum\_isum​i​​ as the cumulative sum upto the current(ithi^{th}i​th​​) index, all we need is sumj−sumi≥0

such that j−i≥k

.

To achive this, instead of checking with all possible values of sumisum\_isum​i​​, we can just

onsider the minimum cumulative sum upto the index j−kj - kj−k. This is because if the required

condition can't be sastisfied with the minimum sumisum\_isum​i​​, it can never be satisfied with a larger value.

To fulfil this, we make use of a prevprevprev variable which again stores the cumulative sums but,

its current index(for cumulative sum) lies behind the current index for sumsumsum at an offset of kkk units.

Thus, by finding the minimum out of prevprevprev and the last minimum value, we can easily find out the required minimum sum value.

Every time after checking the possiblility with a new midmidmid value, at the end, we need to settle at some

value as the average. But, we can observe that eventually, we'll reach a point, where we'll keep moving near

some same value with very small changes. In order to keep our precision in control, we limit this process to

10−510^-510​−​​5 precision, by making use of errorerrorerror and continuing the process till errorerrorerror

becomes lesser than 0.00001 .

# 644\_MirrorNumbers.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A mirror number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to determine if a number is mirror. The number is represented as a string.

Have you met this question in a real interview? Yes

Example

For example, the numbers "69", "88", and "818" are all mirror numbers.

Given num = "69" return true

Given num = "68" return false

public class Solution {

/\*

\* @param num: a string

\* @return: true if a number is strobogrammatic or false

\*/

public boolean isStrobogrammatic(String num) {

if (num == null || num.length() == 0){

return false;

}

for (int i = 0; i < num.length(); i++){

if (num.charAt(i) == '2' || num.charAt(i) == '3' || num.charAt(i) == '4' || num.charAt(i) == '5' || num.charAt(i) == '7'){

return false;

}

}

StringBuilder sb = new StringBuilder();

for (int i = num.length() - 1; i >= 0; i--){

if (num.charAt(i) == '6'){

sb.append('9');

}else if (num.charAt(i) == '9'){

sb.append('6');

}else{

sb.append(num.charAt(i));

}

}

return num.equals(sb.toString());

}

}

## 645\_IdentifyCelebrity.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Suppose you are at a party with n people (labeled from 0 to n - 1) and among them, there may exist one celebrity. The definition of a

celebrity is that all the other n - 1 people know him/her but he/she does not know any of them.

Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask

questions like: "Hi, A. Do you know B?" to get information of whether A knows B. You need to find out the celebrity

(or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given a helper function bool knows(a, b) which tells you whether A knows B. Implement a function int findCelebrity(n),

your function should minimize the number of calls to knows.

Notice

There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party.

If there is no celebrity, return -1.

Have you met this question in a real interview? Yes

Example

Given n = 2

2 // next n \* (n - 1) lines

0 knows 1

1 does not know 0

return 1 // 1 is celebrity

 小技巧总结:

• 降时间复杂度 - > 找冗余

• 思维上双向:true时候,false的时候?

• 我们询问一次的时候只利用答案为true的情况,如果为false呢?

• 一次询问knows(a, b): true a认识 b a一定不是名人

￼•

• 所以一次询问就可以排除一个人,n-1询问后剩下一个人,再对这个做

false a不认识b b一定不是名人 个名人检验就能确定他是否为名人

• 所以实现上就是从左到右扫一遍,每次都是保留下的人和新的人做一次 询问,最开始保留的人设为第1个人

/\* The knows API is defined in the parent class Relation.

boolean knows(int a, int b); \*/

public class Solution extends Relation {

/\*\*

\* @param n a party with n people

\* @return the celebrity's label or -1

\*/

public int findCelebrity(int n) {

if (n <= 0){

return -1;

}

int ans = 0;

for (int i = 0; i < n; i++){

if (knows(ans, i)){

ans = i;

}

}

for (int i = 0; i < n; i++){

if (ans != i && knows(ans, i)){

return -1;

}

if (ans != i && !knows(i, ans)){

return -1;

}

}

return ans;

}

}

## 645\_SetMismatch.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The set S originally contains numbers from 1 to n. But unfortunately, due to the data error,

one of the numbers in the set got duplicated to another number in the set, which results in

repetition of one number and loss of another number.

Given an array nums representing the data status of this set after the error. Your task is to

firstly find the number occurs twice and then find the number that is missing. Return them in the form of an array.

Example 1:

Input: nums = [1,2,2,4]

Output: [2,3]

Note:

The given array size will in the range [2, 10000].

The given array's numbers won't have any order.

class Solution {

public int[] findErrorNums(int[] nums) {

int[] result = new int[2];

int[] hash = new int[nums.length];

for (int i = 0; i < nums.length; i++){

hash[nums[i]-1]++;

}

for (int i = 0; i < nums.length; i++){

if (hash[i] == 2){

result[0] = i + 1 ;

}

if (hash[i] == 0){

result[1] = i + 1;

}

}

return result;

}

}

class Solution {

public int[] findErrorNums(int[] nums) {

int[] res = new int[2];

Set<Integer> set = new HashSet<>();

for (int num : nums){

if (!set.add(num)){

res[0] = num;

}

}

for (int i = 1; i <= nums.length; i++){

if (!set.contains(i)){

res[1] = i;

}

}

return res;

}

}

## 646\_FirstPositionUniqueCharacter.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, find the first non-repeating character in it and return it's index. If it doesn't exist, return -1.

Have you met this question in a real interview? Yes

Example

Given s = "lintcode", return 0.

Given s = "lovelintcode", return 2.

Method: use char as hash array

public class Solution {

/\*

\* @param s: a string

\* @return: it's index

\*/

public int firstUniqChar(String s) {

final int MAX\_CHAR = 256;

char[] count = new char[MAX\_CHAR];

for (int i = 0 ; i < s.length(); i++){

count[s.charAt(i)]++;

}

for (int i = 0; i < s.length(); i++){

if (count[s.charAt(i)] == 1){

return i;

}

}

return -1;

}

}

## 646\_MaximumLengthofPairChain.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.

Now, we define a pair (c, d) can follow another pair (a, b) if and only if b < c. Chain of pairs can be formed in this fashion.

Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs.

You can select pairs in any order.

Example 1:

Input: [[1,2], [2,3], [3,4]]

Output: 2

Explanation: The longest chain is [1,2] -> [3,4]

Note:

The number of given pairs will be in the range [1, 1000].

Similar as Longest Increasing Subsequence

class Solution {

public int findLongestChain(int[][] pairs) {

if (pairs == null || pairs.length == 0){

return 0;

}

Arrays.sort(pairs, new Comparator<int[]>(){

public int compare(int[] p1, int[] p2){

return p1[0] - p2[0];

}

});

int[] f = new int[pairs.length];

for (int i = 0; i < f.length; i++){

f[i] = 1;

}

for (int i = 0; i < f.length; i++){

for (int j = 0; j < i; j++){

if (pairs[i][0] > pairs[j][1]){

f[i] = Math.max(f[i], f[j] + 1);

}

}

}

int max = 0;

for (int i = 0; i < f.length; i++){

max = Math.max(max, f[i]);

}

return max;

}

}

class Solution {

public int findLongestChain(int[][] pairs) {

Arrays.sort(pairs, new Comparator<int[]>(){

public int compare (int[] p1, int[] p2){

return p1[0] - p2[0];

}

});

int n = pairs.length;

int[] dp = new int[n];

int max = 1;

for (int i = 0; i < n; i++){

dp[i] = 1;

for (int j = 0; j < i; j++){

if (pairs[j][1] < pairs[i][0]){

dp[i] = Math.max(dp[i], dp[j] + 1);

}

}

max = Math.max(max, dp[i]);

}

return max;

}

}

## 647\_Palindromic.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string, your task is to count how many palindromic substrings in this string.

The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.

Example 1:

Input: "abc"

Output: 3

Explanation: Three palindromic strings: "a", "b", "c".

Example 2:

Input: "aaa"

Output: 6

Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

Similar to Longest palindromic substring

Time complexity: O(n^2)

class Solution {

public int countSubstrings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int count = 0;

for (int i = 0; i < s.length(); i++){

count += expand(s, i, i);

count += expand(s, i, i+1);

}

return count;

}

private int expand(String s, int i, int j){

int ans = 0;

while (i >= 0 && j < s.length() && s.charAt(i) == s.charAt(j)){

i--;

j++;

ans++;

}

return ans;

}

}

Method 2: DP Best solution

https://github.com/optimisea/Leetcode/blob/master/Java/5\_LongestPalindromicSubstring.java

O(n^2)

class Solution {

public int countSubstrings(String s) {

int count = 0;

int n = s.length();

boolean[][] dp = new boolean[n][n];

for (int i = n-1; i >= 0; i--){

for (int j = i; j < n; j++){

dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i <= 2 || dp[i+1][j-1]);

if (dp[i][j]){

count++;

}

}

}

return count;

}

}

Note that

i must go backward from i = n - 1, can't start from 0, because dp[i][j] replies on dp[i+1]

j must go forward from i can't start from n - 1, because dp[i][j] replies on dp[j-1]

## 647\_SubstringAnagrams.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s and a non-empty string p, find all the start indices of p's anagrams in s.

Strings consists of lowercase English letters only and the length of both strings s and p will not be larger than 40,000.

The order of output does not matter.

Have you met this question in a real interview? Yes

Example

Given s = "cbaebabacd" p = "abc"

return [0, 6]

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

思路:

• Anagrams 的充要条件?

• 元素出现的次数一样就好了

• 一个基本的想法:假设p串的长度为l , 那么就找出s中所有长度为l 的子 串,并统计它们中元素出现的个数

• 朴素算法 O(nl) n为s串长度

• 可以更快吗?

• 想想相邻的两个子串的差别?

• 相当于一个长度为l 的sliding window 从左往右扫一遍

• 每次只增加一个右边元素 && 减少一个左边的元素

• 用什么统计元素个数? – 数组

考点:

• Sliding window + hash

Method 1:

Time complexity: O(n)

public class Solution {

/\*

\* @param s: a string

\* @param p: a string

\* @return: a list of index

\*/

public List<Integer> findAnagrams(String s, String p) {

List<Integer> result = new ArrayList<>();

if (s.length() < p.length()){

return result;

}

int[] count = new int[256];

char[] sc = s.toCharArray();

char[] pc = p.toCharArray();

for (int i = 0; i < p.length(); i++){

count[sc[i]]++;

count[pc[i]]--;

}

int absSum = 0;

for (int item : count){

absSum += Math.abs(item);

}

if (absSum == 0){

result.add(0);

}

for (int i = p.length(); i < s.length(); i++){

absSum = absSum - Math.abs(count[sc[i]]) -

Math.abs(count[sc[i-p.length()]]); //remove the absolute sum

count[sc[i]]++;

count[sc[i-p.length()]]--;

absSum = absSum + Math.abs(count[sc[i]]) +

Math.abs(count[sc[i-p.length()]]); //add the updated absolute sum

if (absSum == 0){

result.add(i-p.length()+1);

}

}

return result;

}

}

Method 2:

Time complexity: O(n \* 256)

public class Solution {

/\*

\* @param s: a string

\* @param p: a string

\* @return: a list of index

\*/

public List<Integer> findAnagrams(String s, String p) {

List<Integer> result = new ArrayList<>();

if (s.length() < p.length()){

return result;

}

char[] sc = s.toCharArray();

char[] pc = p.toCharArray();

int[] countSc = new int[256];

int[] countPc = new int[256];

for (int i = 0; i < p.length(); i++){

countSc[sc[i]]++;

countPc[pc[i]]++;

}

if (Arrays.equals(countSc, countPc)){ //compare two arrays take 256

result.add(0);

}

for (int i = p.length(); i < s.length(); i++){

countSc[sc[i]]++;

countSc[sc[i-p.length()]]--;

if (Arrays.equals(countSc, countPc)){

result.add(i-p.length()+1);

}

}

return result;

}

}

## 648\_Replace.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In English, we have a concept called root, which can be followed by some other words to form another longer word - let's call this word successor. For example, the root an, followed by other, which can form another word another.

Now, given a dictionary consisting of many roots and a sentence. You need to replace all the successor in the sentence with the root forming it. If a successor has many roots can form it, replace it with the root with the shortest length.

You need to output the sentence after the replacement.

Example 1:

Input: dict = ["cat", "bat", "rat"]

sentence = "the cattle was rattled by the battery"

Output: "the cat was rat by the bat"

Note:

The input will only have lower-case letters.

1 <= dict words number <= 1000

1 <= sentence words number <= 1000

1 <= root length <= 100

1 <= sentence words length <= 1000

Implement Trie

Time Complexity: O(N) where N is the length of the sentence. Every query of a word is in linear time.

Space Complexity: O(N), the size of our trie.

class Solution {

class TrieNode{

TrieNode[] children;

String word;

public TrieNode() {

children = new TrieNode[26];

}

}

public String replaceWords(List<String> dict, String sentence) {

TrieNode trie = new TrieNode();

for (int i = 0; i < dict.size(); i++){

TrieNode cur = trie;

for (char c : dict.get(i).toCharArray()){

if (cur.children[c - 'a'] == null){

cur.children[c -'a'] = new TrieNode();

}

cur = cur.children[c - 'a'];

}

cur.word = dict.get(i);

}

String[] strArray = sentence.split("\\s+");

StringBuilder sb = new StringBuilder();

for (String str : strArray){

TrieNode cur = trie;

for (char c : str.toCharArray()){

if (cur.children[c - 'a'] == null || cur.word != null){

break;

}

cur= cur.children[c - 'a'];

}

sb.append(cur.word != null ? cur.word : str);

sb.append(" ");

}

// sb.deleteCharAt(sb.length() - 1); //return sb

// sb.setLength(sb.length() - 1); //return void

return sb.deleteCharAt(sb.length() - 1).toString();

}

}

class Solution {

class TrieNode{

TrieNode[] children;

boolean isEnd;

public TrieNode(){

children = new TrieNode[26];

isEnd = false;

}

}

public String replaceWords(List<String> dict, String sentence) {

TrieNode root = new TrieNode();

for (String str : dict){

TrieNode node = root;

for (int i = 0; i < str.length(); i++){

char c = str.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

String[] strs = sentence.split("\\s+");

StringBuilder res = new StringBuilder();

for (String str : strs){

TrieNode node = root;

StringBuilder sb = new StringBuilder();

boolean found = false;

for (int i = 0; i < str.length(); i++){

char c = str.charAt(i);

sb.append(c);

if (node.children[c - 'a'] == null){

break;

}

node = node.children[c - 'a'];

if (node.isEnd){

found = true;

break;

}

}

if (found){

res.append(sb.toString() + " ");

}else{

res.append(str + " ");

}

}

return res.toString().trim();

}

}

## 648\_WordAbbreviationSet.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

An abbreviation of a word follows the form . Below are some examples of word abbreviations:

a) it --> it (no abbreviation)

1

b) d|o|g --> d1g

1 1 1

1---5----0----5--8

c) i|nternationalizatio|n --> i18n

1

1---5----0

d) l|ocalizatio|n --> l10n

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no other word from the dictionary has the same abbreviation.

Have you met this question in a real interview? Yes

Example

Given dictionary = [ "deer", "door", "cake", "card" ]

isUnique("dear") // return false

isUnique("cart") // return true

isUnique("cane") // return false

isUnique("make") // return true

public class ValidWordAbbr {

private Map<String, Integer> dict = new HashMap<>();

private Map<String, Integer> abbr = new HashMap<>();

/\*

\* @param dictionary: a list of words

\*/public ValidWordAbbr(String[] dictionary) {

for (int i = 0; i < dictionary.length; i++){

String s = dictionary[i];

if (!dict.containsKey(s)){

dict.put(s, 1);

}else{

dict.put(s, dict.get(s) + 1);

}

String a = getAbbr(s);

if (!abbr.containsKey(a)){

abbr.put(a, 1);

}else{

abbr.put(a, abbr.get(a) + 1);

}

}

}

/\*

\* @param word: a string

\* @return: true if its abbreviation is unique or false

\*/

public boolean isUnique(String word) {

String ab = getAbbr(word);

return dict.get(word) == abbr.get(ab);

}

private String getAbbr(String word){

if (word.length() <= 2){

return word;

}

//use Integer.toString

return (word.charAt(0) + Integer.toString(word.length() - 2) +

word.charAt(word.length() - 1));

}

}

/\*\*

\* Your ValidWordAbbr object will be instantiated and called as such:

\* ValidWordAbbr obj = new ValidWordAbbr(dictionary);

\* boolean param = obj.isUnique(word);

\*/

## 649\_BinaryTreeFlipping.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or

empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

Have you met this question in a real interview? Yes

Example

Given a binary tree {1,2,3,4,5}

1

/ \

2 3

/ \

4 5

return the root of the binary tree {4,5,2,#,#,3,1}.

4

/ \

5 2

/ \

3 1

Method 1: Recursion

/\*\*

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

public class Solution {

/\*

\* @param root: the root of binary tree

\* @return: new root

\*/

TreeNode newRoot;

public TreeNode upsideDownBinaryTree(TreeNode root) {

if (root == null){

return null;

}

dfs(root);

return newRoot;

}

private void dfs(TreeNode cur){

if (cur.left == null){

newRoot = cur;

return;

}

dfs(cur.left);

cur.left.right = cur;

cur.left.left = cur.right;

cur.left = null;

cur.right = null;

}

}

Method 2: iteration (best)

public class Solution {

/\*

\* @param root: the root of binary tree

\* @return: new root

\*/

public TreeNode upsideDownBinaryTree(TreeNode root) {

if (root == null || root.left == null){

return root;

}

TreeNode cur = root;

TreeNode next = null;

TreeNode temp = null;

TreeNode prev = null;

while (cur != null){

next = cur.left;

cur.left = temp;

temp = cur.right;

cur.right = prev;

prev = cur;

cur = next;

}

return prev;

}

}

## 65\_Valid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Validate if a given string can be interpreted as a decimal number.

Some examples:

"0" => true

" 0.1 " => true

"abc" => false

"1 a" => false

"2e10" => true

" -90e3 " => true

" 1e" => false

"e3" => false

" 6e-1" => true

" 99e2.5 " => false

"53.5e93" => true

" --6 " => false

"-+3" => false

"95a54e53" => false

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

However, here is a list of characters that can be in a valid decimal number:

Numbers 0-9

Exponent - "e"

Positive/negative sign - "+"/"-"

Decimal point - "."

Of course, the context of these characters also matters in the input.

class Solution {

public boolean isNumber(String s) {

s = s.trim();

if (s.length() == 0){

return false;

}

boolean hasPoint = false;

boolean hasExp = false;

boolean hasSign = false;

boolean hasNum = false;

for (int i = 0; i < s.length(); i++){

char c = s.charAt(i);

if (Character.isLetter(c)){

if (c != 'e' || hasExp || i == 0 || !hasNum){

return false;

}

hasExp = true;

if ((!Character.isDigit(s.charAt(i-1)) && s.charAt(i-1) != '.') || i == s.length() - 1 || s.charAt(i+1) == '.' || s.charAt(i+1) == ' '){

return false;

}

}else if (c == '+' || c == '-'){

if ((hasSign && !hasExp) || i == s.length() - 1 || (i != 0 && s.charAt(i-1) != 'e')){

return false;

}

hasSign = true;

}else if (c == '.'){

if (hasPoint || hasExp){

return false;

}

hasPoint = true;

}else if (c == ' '){

return false;

}else {

hasNum = true;

}

}

return hasNum == true;

}

}

## 650\_BinaryTreeLeavesOrderTraversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, collect a tree's nodes as if you were doing this: Collect and remove all leaves, repeat until the tree is empty.

Have you met this question in a real interview? Yes

Example

Given binary tree:

1

/ \

2 3

/ \

4 5

Returns [[4, 5, 3], [2], [1]].

/\*\*

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

• 看看每个子树的高度?

– 其实第k层包含的就是所有高度为k的节点

同一层中,要求的顺序是从左往右 (use postorder traversal to keep the order)

所以怎么求高度?怎么保存答案? – DFS计算节点高度,hash 保存答案

Method 1: label node from bottom to top, e.g., bottom: 0...

public class Solution {

/\*

\* @param root: the root of binary tree

\* @return: collect and remove all leaves

\*/

Map<Integer, List<Integer>> map = new HashMap<>();

public List<List<Integer>> findLeaves(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

int depth = dfs(root);

for (int i = 0; i <= depth; i++){

result.add(map.get(i));

}

return result;

}

private int dfs(TreeNode cur){

if (cur == null){

return -1;

}

int leftDepth = dfs(cur.left);

int rightDepth = dfs(cur.right);

int depth = Math.max(leftDepth, rightDepth) + 1;

if (!map.containsKey(depth)){

map.put(depth, new ArrayList<Integer>());

}

map.get(depth).add(cur.val);

return depth;

}

}

Method 2: label node (label node from top to bottom, e.g. top: 0,1,2........)

Similar to https://github.com/optimisea/Leetcode/blob/master/Java/865\_Smallest.java

## 650\_Keys.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Initially on a notepad only one character 'A' is present. You can perform two operations on this notepad for each step:

Copy All: You can copy all the characters present on the notepad (partial copy is not allowed).

Paste: You can paste the characters which are copied last time.

Given a number n. You have to get exactly n 'A' on the notepad by performing the minimum number of steps permitted. Output the minimum number of steps to get n 'A'.

Example 1:

Input: 3

Output: 3

Explanation:

Intitally, we have one character 'A'.

In step 1, we use Copy All operation.

In step 2, we use Paste operation to get 'AA'.

In step 3, we use Paste operation to get 'AAA'.

Method 1:

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public int minSteps(int n) {

int[] dp = new int[n+1];

for (int i = 2; i <= n; i++){

dp[i] = i;

for (int j = 2; j < i; j++){

if (i % j == 0){

dp[i] = Math.min(dp[i], dp[j] + i/j);

}

}

}

return dp[n];

}

}

Method 2: Best solution

https://leetcode.com/problems/2-keys-keyboard/discuss/105897/Loop-best-case-log(n)-no-DP-no-extra-space-no-recursion-with-explanation

Time complexity: O(n) approximately

Space complexity: O(1)

class Solution {

public int minSteps(int n) {

int ans = 0;

int d = 2;

while (n > 1){

while (n % d == 0){

ans += d;

n /= d;

}

d++;

}

return ans;

}

}

class Solution {

public int minSteps(int n) {

int ans = 0;

for (int i = 2; i <= n; i++){

while (n % i == 0){

ans += i;

n /= i;

}

}

return ans;

}

}

## 651\_BinaryTreeVerticalOrderTraversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the vertical order traversal of its nodes' values. (ie, from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from left to right.

Have you met this question in a real interview? Yes

Example

Given binary tree {3,9,20,#,#,15,7}

3

/\

/ \

9 20

/\

/ \

15 7

Return its vertical order traversal as:

[[9],[3,15],[20],[7]]

Given binary tree {3,9,8,4,0,1,7}

3

/\

/ \

9 8

/\ /\

/ \/ \

4 01 7

Return its vertical order traversal as:

[[4],[9],[3,0,1],[8],[7]]

/\*\*

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

思路:

• 这一题有三个顺序:

– 第一,列数从小到大

– 第二,列数相同时行数从小到大

– 第三,列数行数都相同时,从左到右

怎样计算列数?

– Root为0 向左-1 向右+1

• 怎样保证第一个顺序,列数从小到大 – Hash

怎样保证第二个顺序,行数从小到大 – BFS 一行一行的访问

Check another one:

Refer to https://github.com/optimisea/Leetcode/blob/master/Java/987\_Vertical.java

for DFS solution with more challenging condition

public class Solution {

/\*

\* @param root: the root of tree

\* @return: the vertical order traversal

\*/

public List<List<Integer>> verticalOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null){

return result;

}

Map<Integer, List<Integer>> map = new HashMap<>();

Queue<TreeNode> qNode = new LinkedList<>();

Queue<Integer> qCol = new LinkedList<>();

qNode.offer(root);

qCol.offer(0);

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;

while (!qNode.isEmpty()){

TreeNode node = qNode.poll();

int col = qCol.poll();

if (!map.containsKey(col)){

map.put(col, new ArrayList<Integer>());

}

map.get(col).add(node.val);

min = Math.min(min, col);

max = Math.max(max, col);

if (node.left != null){

qNode.offer(node.left);

qCol.offer(col - 1);

}

if (node.right != null){

qNode.offer(node.right);

qCol.offer(col + 1);

}

}

for (int i = min; i <= max; i++){

result.add(map.get(i));

}

return result;

}

}

Method 2: label node

similar as https://github.com/optimisea/Leetcode/blob/master/Java/865\_Smallest.java

## 651\_Keys.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Imagine you have a special keyboard with the following keys:

Key 1: (A): Print one 'A' on screen.

Key 2: (Ctrl-A): Select the whole screen.

Key 3: (Ctrl-C): Copy selection to buffer.

Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.

Now, you can only press the keyboard for N times (with the above four keys), find out the maximum numbers of 'A' you can print on screen.

Example 1:

Input: N = 3

Output: 3

Explanation:

We can at most get 3 A's on screen by pressing following key sequence:

A, A, A

Example 2:

Input: N = 7

Output: 9

Explanation:

We can at most get 9 A's on screen by pressing following key sequence:

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

Note:

1 <= N <= 50

Answers will be in the range of 32-bit signed integer.

Method:

Time complexity: O(n^2)

Space complexity: O(1)

We use i steps to reach maxA(i) then use the remaining n - i steps to reach n - i - 1 copies of maxA(i)

For example:

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

Here we have n = 7 and we used i = 3 steps to reach AAA

Then we use the remaining n - i = 4 steps: Ctrl A, Ctrl C, Ctrl V, Ctrl V, to reach n - i - 1 = 3 copies of AAA

We either don't make copies at all, in which case the answer is just n, or if we want to make copies, we need to have 3 steps

reserved for Ctrl A, Ctrl C, Ctrl V so i can be at most n - 3

http://www.cnblogs.com/grandyang/p/7448390.html

class Solution {

public int maxA(int N) {

int max = N;

for (int i = 1; i <= N - 3; i++){

max = Math.max(max, maxA(i) \* (N-i-1));

}

return max;

}

}

class Solution {

public int maxA(int N) {

int[] dp = new int[N+1];

for (int i = 1; i <= N; i++){

dp[i] = i;

for (int j = 1; j < i - 2; j++){

dp[i] = Math.max(dp[i], dp[j] \* (i-2-j+1));

}

}

return dp[N];

}

}

## 652\_Factorization.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A non-negative numbers can be regarded as product of its factors.

Write a function that takes an integer n and return all possible combinations of its factors.

Notice

Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).

The solution set must not contain duplicate combination.

Have you met this question in a real interview? Yes

Example

Given n = 8

return [[2,2,2],[2,4]]

// 8 = 2 x 2 x 2 = 2 x 4.

Given n = 1

return []

Given n = 12

return [[2,6],[2,2,3],[3,4]]

Method 1:

public class Solution {

/\*

\* @param n: An integer

\* @return: a list of combination

\*/

public List<List<Integer>> getFactors(int n) {

List<List<Integer>> result = new ArrayList<>();

dfs(result, new ArrayList<>(), 2, n);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int lastFactor, int remain){

if (remain == 1){

if (item.size() != 1)

result.add(new ArrayList<Integer>(item));

return;

}

for (int i = lastFactor; i <= remain/i; i++){

if (remain % i == 0){

item.add(i);

dfs(result, item, i, remain/i);

item.remove(item.size() - 1);

}

}

item.add(remain);

dfs(result,item, remain, 1);

item.remove(item.size() - 1);

}

}

Method 2:

public class Solution {

/\*

\* @param n: An integer

\* @return: a list of combination

\*/

public List<List<Integer>> getFactors(int n) {

List<List<Integer>> result = new ArrayList<>();

dfs(result, new ArrayList<>(), 2, n);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int lastFactor, int remain){

if (!item.isEmpty()){

item.add(remain);

result.add(new ArrayList<Integer>(item));

item.remove(item.size() - 1);

}

for (int i = lastFactor; i <= remain/i; i++){

if (remain % i == 0){

item.add(i);

dfs(result, item, i, remain/i);

item.remove(item.size() - 1);

}

}

}

}

## 652\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return all duplicate subtrees. For each kind of duplicate subtrees, you only need to return the root node of

any one of them.

Two trees are duplicate if they have the same structure with same node values.

Example 1:

1

/ \

2 3

/ / \

4 2 4

/

4

The following are two duplicate subtrees:

2

/

4

and

4

Therefore, you need to return above trees' root in the form of a list.

Method 1: Serialization

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

Time Complexity: O(N2), where N is the number of nodes in the tree. We visit each node once, but each creation of serial may take O(N)) work.

Space Complexity: O(N2), the size of map.

serialization 1,2,#,#,3,4,#,#,5,#,#, which is a unique representation of the tree.

Perform a depth-first search, where the recursive function returns the serialization of the tree. At each node, record the result in a map, and analyze the map after to determine duplicate subtrees.

class Solution {

public List<TreeNode> findDuplicateSubtrees(TreeNode root) {

List<TreeNode> res = new ArrayList<>();

Map<String, Integer> map = new HashMap<>();

dfs(root, map, res);

return res;

}

private String dfs(TreeNode root, Map<String, Integer> map, List<TreeNode> res){

if (root == null){

return "n";

}

String str = root.val + "," + dfs(root.left, map, res) + "," + dfs(root.right, map, res);

map.put(str, map.getOrDefault(str, 0) + 1);

if (map.get(str) == 2){

res.add(root);

}

return str;

}

}

Method 2:

Time complexity: O(n)

Space complexity: O(n^2)

class Solution {

public List<TreeNode> findDuplicateSubtrees(TreeNode root) {

List<TreeNode> res = new ArrayList<>();

Map<String, Integer> map = new HashMap<>();

dfs(root, map, res);

return res;

}

private String dfs(TreeNode root, Map<String, Integer> map, List<TreeNode> res){

if (root == null){

return "n";

}

StringBuilder sb = new StringBuilder();

sb.append(root.val);

sb.append(",");

sb.append(dfs(root.left, map, res));

sb.append(",");

sb.append(dfs(root.right, map, res));

String str = sb.toString();

map.put(str, map.getOrDefault(str, 0) + 1);

if (map.get(str) == 2){

res.add(root);

}

return str;

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<TreeNode> findDuplicateSubtrees(TreeNode root) {

List<TreeNode> res = new ArrayList<>();

Map<String, Integer> map = new HashMap<>();

preOrder(res, root, map);

return res;

}

private String preOrder(List<TreeNode> res, TreeNode root, Map<String, Integer> map){

if (root == null){

return "null,";

}

String str = root.val + ",";

str += preOrder(res, root.left, map);

str += preOrder(res, root.right, map);

map.put(str, map.getOrDefault(str, 0) + 1);

if (map.get(str) == 2){

res.add(root);

}

return str;

}

}

## 653\_AddOperators.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string that contains only digits 0-9 and a target value, return all possibilities to add binary operators (not unary) +, -, or \* between the digits so they evaluate to the target value.

Have you met this question in a real interview? Yes

Example

"123", 6 -> ["1+2+3", "1\*2\*3"]

"232", 8 -> ["2\*3+2", "2+3\*2"]

"105", 5 -> ["1\*0+5","10-5"]

"00", 0 -> ["0+0", "0-0", "0\*0"]

"3456237490", 9191 -> []

思路:

• 简化版,只有加减没有乘号怎么做? – 枚举数字后再枚举符号

– sum作为中间状态记录下前面的和

• 只有加减的时候比较好处理,那么怎么处理乘号呢?

– 再记录一个状态,最后的连乘的那个数lastFactor

• 34-56\*23\*74 90 枚举完34562374时,lastFactor = 56 \* 23 \* 74 • 34-562-374 90 枚举完34562374时,lastFactor = -374

– 乘号时:

• sum更新方法:sum = sum – lastFactor + lastFactor \* 当前枚举的数

• lastFactor更新方法:lastFactor = lastFactor \* 当前枚举的数

– 加号/减号时:

• sum更新方法:sum = sum +/- 当前枚举的数

• lastFactor更新方法:lastFactor = 当前枚举的数(加号时为正,减号时为负)

• 有哪些情况需要特殊判断?

– 第一个数字前不能有符号 (+3456 -23 -74 +90 错误) – 数字不能有前导0 (2543+034 错误)

public class Solution {

/\*

\* @param num: a string contains only digits 0-9

\* @param target: An integer

\* @return: return all possibilities

\*/

public List<String> addOperators(String num, int target) {

List<String> result = new ArrayList<>();

dfs(result, "", num, target, 0, 0, 0);

return result;

}

private void dfs(List<String> result, String path, String num, int target, int start, long sum, long lastFactor){

if (start == num.length()){

if (sum == target){

result.add(path);

}

return;

}

for (int i = start; i < num.length(); i++){

String str = num.substring(start, i + 1);

long cur = Long.parseLong(str);

if (start == 0){

dfs(result, str, num, target, i + 1, cur, cur);

}else{

dfs(result, path + "\*" + str, num, target, i + 1, sum - lastFactor + lastFactor \* cur, lastFactor \* cur);

dfs(result, path + "+" + str, num, target, i+1, sum + cur, cur);

dfs(result, path + "-" + str, num, target, i+1, sum - cur, -cur);

}

if (num.charAt(start) == '0'){

break;

}

}

}

}

# 653\_Two.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a Binary Search Tree and a target number, return true if there exist two elements in the BST such that their sum is equal to the given target.

Example 1:

Input:

5

/ \

3 6

/ \ \

2 4 7

Target = 9

Output: True

Example 2:

Input:

5

/ \

3 6

/ \ \

2 4 7

Target = 28

Output: False

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

Method 1: HashMap

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public boolean findTarget(TreeNode root, int k) {

Map<Integer, Integer> map = new HashMap<>();

dfs(root, map);

for (int key : map.keySet()){

if (map.containsKey(k - key)){

if (key != k - key || map.get(key) >= 2){

return true;

}

}

}

return false;

}

private void dfs(TreeNode root, Map<Integer, Integer> map){

if (root == null){

return;

}

map.put(root.val, map.getOrDefault(map.get(root.val), 0) + 1);

dfs(root.left, map);

dfs(root.right, map);

}

}

Method 2: HashSet

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public boolean findTarget(TreeNode root, int k) {

Set<Integer> set = new HashSet<>();

return dfs(root, set, k);

}

private boolean dfs(TreeNode root, Set<Integer> set, int k){

if (root == null){

return false;

}

if (set.contains(k - root.val)){

return true;

}

set.add(root.val);

return dfs(root.left, set, k) || dfs(root.right, set, k);

}

}

Method 3: Two points due to BST

Time complexity: O(n)

Space complexity: O(n)

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean findTarget(TreeNode root, int k) {

List<Integer> list = new ArrayList<>();

inOrder(root, list);

int left = 0;

int right = list.size() - 1;

while (left < right){

if (list.get(left) + list.get(right) == k){

return true;

}else if (list.get(left) + list.get(right) < k){

left++;

}else{

right--;

}

}

return false;

}

private void inOrder(TreeNode root, List<Integer> list){

if (root == null){

return;

}

inOrder(root.left, list);

list.add(root.val);

inOrder(root.right, list);

}

}

## 654\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array with no duplicates. A maximum tree building on this array is defined as follow:

The root is the maximum number in the array.

The left subtree is the maximum tree constructed from left part subarray divided by the maximum number.

The right subtree is the maximum tree constructed from right part subarray divided by the maximum number.

Construct the maximum tree by the given array and output the root node of this tree.

Example 1:

Input: [3,2,1,6,0,5]

Output: return the tree root node representing the following tree:

6

/ \

3 5

\ /

2 0

\

1

Note:

The size of the given array will be in the range [1,1000].

Time complexity: O(n)

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode constructMaximumBinaryTree(int[] nums) {

return dfs(nums, 0, nums.length - 1);

}

private TreeNode dfs(int[] nums, int start, int end){

if (start > end){

return null;

}

int max = nums[start];

int index = start;

for (int i = start; i <= end; i++){

if (max < nums[i]){

max = nums[i];

index = i;

}

}

TreeNode root = new TreeNode(max);

root.left = dfs(nums, start, index-1);

root.right = dfs(nums, index+1, end);

return root;

}

}

class Solution {

public TreeNode constructMaximumBinaryTree(int[] nums) {

return dfs(nums, 0, nums.length - 1);

}

private TreeNode dfs(int[] nums, int start, int end){

if (start > end){

return null;

}

int index = start;

for (int i = start; i <= end; i++){

if (nums[index] < nums[i]){

index = i;

}

}

TreeNode root = new TreeNode(nums[index]);

root.left = dfs(nums, start, index-1);

root.right = dfs(nums, index+1, end);

return root;

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode constructMaximumBinaryTree(int[] nums) {

if (nums == null || nums.length == 0){

return null;

}

return build(nums, 0, nums.length - 1);

}

private TreeNode build(int[] nums, int start, int end){

if (start > end){

return null;

}

if (start == end){

return new TreeNode(nums[start]);

}

int max = Integer.MIN\_VALUE;

int index = -1;

for (int i = start; i <= end; i++){

if (max < nums[i]){

max = nums[i];

index = i;

}

}

TreeNode root = new TreeNode(max);

root.left = build(nums, start, index-1);

root.right = build(nums, index+1, end);

return root;

}

}

## 654\_SparseMatrixMultiplication.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two Sparse Matrix A and B, return the result of AB.

You may assume that A's column number is equal to B's row number.

Have you met this question in a real interview?

Example

A = [

[ 1, 0, 0],

[-1, 0, 3]

]

B = [

[ 7, 0, 0 ],

[ 0, 0, 0 ],

[ 0, 0, 1 ]

]

| 1 0 0 | | 7 0 0 | | 7 0 0 |

AB = | -1 0 3 | x | 0 0 0 | = | -7 0 3 |

| 0 0 1 |

Method 1: O(nmt)

public class Solution {

/\*

\* @param A: a sparse matrix

\* @param B: a sparse matrix

\* @return: the result of A \* B

\*/

public int[][] multiply(int[][] A, int[][] B) {

int n = A.length;

int t = A[0].length;

int m = B[0].length;

int[][] result = new int[n][m];

for (int i = 0; i < n; i++){

for (int j = 0; j < m; j++){

for (int k = 0; k < t; k++){

result[i][j] += A[i][k] \* B[k][j];

}

}

}

return result;

}

}

Method 2: O(small percent of mnt)

public class Solution {

/\*

\* @param A: a sparse matrix

\* @param B: a sparse matrix

\* @return: the result of A \* B

\*/

public int[][] multiply(int[][] A, int[][] B) {

int n = A.length;

int t = A[0].length;

int m = B[0].length;

int[][] result = new int[n][m];

for (int i = 0; i < n; i++){

for (int k = 0; k < t; k++){

if (A[i][k] == 0){

continue;

}

for (int j = 0; j < m; j++){

if (B[k][j] == 0){

continue;

}

result[i][j] += A[i][k] \* B[k][j];

}

}

}

return result;

}

}

Method 3: transfer sparse matrix to 链表 ArrayList

public class Solution {

/\*

\* @param A: a sparse matrix

\* @param B: a sparse matrix

\* @return: the result of A \* B

\*/

public int[][] multiply(int[][] A, int[][] B) {

int n = A.length;

int t = A[0].length;

int m = B[0].length;

int[][] result = new int[n][m];

//pre-process to get the sparse items

List<List<Integer>> col = new ArrayList<>();

List<List<Integer>> val = new ArrayList<>();

for (int k = 0 ; k < t; k++){

col.add(new ArrayList<Integer>());

val.add(new ArrayList<Integer>());

for (int j = 0; j < m; j++){

if (B[k][j] != 0){

col.get(k).add(j);

val.get(k).add(B[k][j]);

}

}

}

for (int i = 0; i < n; i++){

for (int k = 0; k < t; k++){

if (A[i][k] != 0){

for (int p = 0; p < val.get(k).size(); p++){

int j = col.get(k).get(p);

int v = val.get(k).get(p);

result[i][j] += A[i][k] \* v;

}

}

}

}

return result;

}

}

## 655\_BigIntegerAddition.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two non-negative integers num1 and num2 represented as string, return the sum of num1 and num2.

Notice

The length of both num1 and num2 is < 5100.

Both num1 and num2 contains only digits 0-9.

Both num1 and num2 does not contain any leading zero.

You must not use any built-in BigInteger library or convert the inputs to integer directly.

Have you met this question in a real interview? Yes

Example

Given num1 = "123", num2 = "45"

return "168"

高精度加法的实现?

– 方法1:每次模拟进位 (见代码)

– 方法2:对应数位相加后一次性进位

Method 1:

public class Solution {

/\*

\* @param num1: a non-negative integers

\* @param num2: a non-negative integers

\* @return: return sum of num1 and num2

\*/

public String addStrings(String num1, String num2) {

if (num1 == null || num2 == null){

return null;

}

int n = num1.length();

int m = num2.length();

String ans = "";

int carry = 0;

int sum;

for (int i = n-1, j = m-1; i >= 0 || j >=0 ; i--,j --){

sum = carry;

sum += (i >= 0) ? num1.charAt(i) - '0' : 0;

sum += (j >= 0) ? num2.charAt(j) - '0' : 0;

ans = sum % 10 + ans;

carry = sum / 10;

}

if (carry != 0){

ans = carry + ans;

}

return ans;

}

}

## 655\_Print.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Print a binary tree in an m\*n 2D string array following these rules:

The row number m should be equal to the height of the given binary tree.

The column number n should always be an odd number.

The root node's value (in string format) should be put in the exactly middle of the first row it can be put. The column and the row where the root node belongs will separate the rest space into two parts (left-bottom part and right-bottom part). You should print the left subtree in the left-bottom part and print the right subtree in the right-bottom part. The left-bottom part and the right-bottom part should have the same size. Even if one subtree is none while the other is not, you don't need to print anything for the none subtree but still need to leave the space as large as that for the other subtree. However, if two subtrees are none, then you don't need to leave space for both of them.

Each unused space should contain an empty string "".

Print the subtrees following the same rules.

Example 1:

Input:

1

/

2

Output:

[["", "1", ""],

["2", "", ""]]

Example 2:

Input:

1

/ \

2 3

\

4

Output:

[["", "", "", "1", "", "", ""],

["", "2", "", "", "", "3", ""],

["", "", "4", "", "", "", ""]]

Example 3:

Input:

1

/ \

2 5

/

3

/

4

Output:

[["", "", "", "", "", "", "", "1", "", "", "", "", "", "", ""]

["", "", "", "2", "", "", "", "", "", "", "", "5", "", "", ""]

["", "3", "", "", "", "", "", "", "", "", "", "", "", "", ""]

["4", "", "", "", "", "", "", "", "", "", "", "", "", "", ""]]

Note: The height of binary tree is in the range of [1, 10]

Method 1: Iteration

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<String>> printTree(TreeNode root) {

List<List<String>> res = new ArrayList<>();

int row = depth(root);

int col = (int)Math.pow(2, row) - 1;

Queue<TreeNode> queue = new LinkedList<>();

Queue<Integer> index = new LinkedList<>();

queue.offer(root);

index.offer(col/2);

int len = col/2;

for (int i = 0; i < row; i++){

List<String> item = new ArrayList<>();

int size = queue.size();

int k = 0;

for (int j = 0; j < size; j++){

TreeNode node = queue.poll();

int idx = index.poll();

if (node.left != null){

queue.offer(node.left);

index.offer(idx-1-len/2);

}

if (node.right != null){

queue.offer(node.right);

index.offer(idx+1+len/2);

}

while (k < idx){

item.add("");

k++;

}

item.add(String.valueOf(node.val));

k++;

}

while (k < col){

item.add("");

k++;

}

len /= 2;

res.add(item);

}

return res;

}

private int depth(TreeNode root){

if (root == null){

return 0;

}

return Math.max(depth(root.left), depth(root.right)) + 1;

}

}

Note 1: col = (int) Math.pow(2, row) - 1;

Note 2: we always print a node at the center of its subtree index range.

root is at the center of left and right, say mid

root.left (if not null) is at the center of left and mid - 1

root.right (if not null) is at the center of mid + 1 and right

https://leetcode.com/problems/print-binary-tree/discuss/106269/Java-Iterative-Level-Order-Traversal-with-Queue

class Solution {

public List<List<String>> printTree(TreeNode root) {

List<List<String>> res = new ArrayList<>();

int row = depth(root);

int col = (int)Math.pow(2, row) - 1;

for (int i = 0; i < row; i++){

List<String> item = new ArrayList<>();

for (int j = 0; j < col; j++){

item.add("");

}

res.add(item);

}

Queue<TreeNode> queue = new LinkedList<>();

Queue<int[]> index = new LinkedList<>();

queue.offer(root);

index.offer(new int[]{0, col-1});

int r = -1;

for (int i = 0; i < row; i++){

int size = queue.size();

r++;

for (int j = 0; j < size; j++){

TreeNode node = queue.poll();

int[] idx = index.poll();

int start = idx[0];

int end = idx[1];

int mid = start + (end - start) / 2;

if (node.left != null){

queue.offer(node.left);

index.offer(new int[]{start, mid-1});

}

if (node.right != null){

queue.offer(node.right);

index.offer(new int[]{mid+1, end});

}

res.get(r).set(mid, String.valueOf(node.val));

}

}

return res;

}

private int depth(TreeNode root){

if (root == null){

return 0;

}

return Math.max(depth(root.left), depth(root.right)) + 1;

}

}

Method 2: Recursion

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<String>> printTree(TreeNode root) {

List<List<String>> res = new ArrayList<>();

int row = depth(root);

int col = (int) Math.pow(2, row) - 1;

for (int i = 0; i < row; i++){

List<String> item = new ArrayList<>();

for (int j = 0; j < col; j++){

item.add("");

}

res.add(item);

}

populate(root, res, 0, row, 0, col-1);

return res;

}

private void populate(TreeNode root, List<List<String>> res, int rowStart, int rowEnd, int colStart, int colEnd){

if (root == null || rowStart == rowEnd){

return;

}

int mid = colStart + (colEnd - colStart) / 2;

res.get(rowStart).set(mid, String.valueOf(root.val));

populate(root.left, res, rowStart+1, rowEnd, colStart, mid - 1);

populate(root.right, res, rowStart+1, rowEnd, mid + 1, colEnd);

}

private int depth(TreeNode root){

if (root == null){

return 0;

}

return Math.max(depth(root.left), depth(root.right)) + 1;

}

}

## 656\_BigIntegerMultiplication.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2

Have you met this question in a real interview? Yes

Example

The length of both num1 and num2 is < 110.

Both num1 and num2 contains only digits 0-9.

Both num1 and num2 does not contain any leading zero.

You must not use any built-in BigInteger library or convert the inputs to integer directly.

高精度乘法的实现

对应数位相加后一次性进位

The same as String multipy

https://github.com/optimisea/Leetcode/blob/master/Java/43\_MultiplyStrings.java

public class Solution {

/\*

\* @param num1: a non-negative integers

\* @param num2: a non-negative integers

\* @return: return product of num1 and num2

\*/

public String multiply(String num1, String num2) {

int l1 = num1.length();

int l2 = num2.length();

int[] ans = new int[l1 + l2 + 1];

for (int i = 0; i < l1; i++){

for (int j = 0; j < l2; j++){

ans[i + j] += (num1.charAt(l1 - 1 - i) - '0') \* (num2.charAt(l2 - 1 - j) - '0');

}

}

for (int i = 0; i < l1 + l2; i++){

ans[i + 1] += ans[i] / 10;

ans[i] = ans[i] % 10;

}

int i = l1 + l2;

while (ans[i] == 0 && i >= 1){

i--;

}

String str = "";

while (i >= 0){

str = str + ans[i--];

}

return str;

}

}

## 656\_Coin.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A (index starts at 1) consisting of N integers: A1, A2, ..., AN and an integer B. The integer B denotes that

from any place (suppose the index is i) in the array A, you can jump to any one of the place in the array A indexed

i+1, i+2, …, i+B if this place can be jumped to. Also, if you step on the index i, you have to pay Ai coins. If Ai is -1,

it means you can’t jump to the place indexed i in the array.

Now, you start from the place indexed 1 in the array A, and your aim is to reach the place indexed N using the minimum coins.

You need to return the path of indexes (starting from 1 to N) in the array you should take to get to the place indexed N using

minimum coins.

If there are multiple paths with the same cost, return the lexicographically smallest such path.

If it's not possible to reach the place indexed N then you need to return an empty array.

Example 1:

Input: [1,2,4,-1,2], 2

Output: [1,3,5]

Example 2:

Input: [1,2,4,-1,2], 1

Output: []

Note:

Path Pa1, Pa2, ..., Pan is lexicographically smaller than Pb1, Pb2, ..., Pbm, if and only if at the first i where

Pai and Pbi differ, Pai < Pbi; when no such i exists, then n < m.

A1 >= 0. A2, ..., AN (if exist) will in the range of [-1, 100].

Length of A is in the range of [1, 1000].

B is in the range of [1, 100].

LIS DP + Record Path

class Solution {

class Pair {

int num;

List<Integer> list;

public Pair (int num, List<Integer> list){

this.num = num;

this.list = list;

}

}

public List<Integer> cheapestJump(int[] A, int B) {

List<Integer> res = new ArrayList<>();

int n = A.length;

Pair[] dp = new Pair[n+1];

for (int i = 0; i < dp.length; i++){

if (i == 0 || i == 1){

dp[i] = new Pair(0, new ArrayList<>());

}else{

dp[i] = new Pair(Integer.MAX\_VALUE, new ArrayList<>());

}

}

for (int i = 1; i <= n; i++){

if (A[i-1] == -1){

continue;

}

for (int j = 1; j <= B; j++){

if (i > j && A[i-j-1] != -1){

if (dp[i-j].num + A[i-j-1] < dp[i].num){

int newNum = A[i-j-1] + dp[i-j].num;

List<Integer> newList = new ArrayList<>();

newList.addAll(dp[i-j].list);

newList.add(i-j-1);

dp[i] = new Pair(newNum, newList);

}else if (dp[i-j].num + A[i-j-1] == dp[i].num){

List<Integer> newList = new ArrayList<>();

newList.addAll(dp[i-j].list);

newList.add(i-j-1);

String s1 = "";

for (int k : newList){

s1 += k;

}

String s2 = "";

for (int k : dp[i].list){

s2 += k;

}

if (s1.compareTo(s2) < 0){

dp[i] = new Pair(dp[i].num, newList);

}

}

}

}

}

if (dp[n].num == Integer.MAX\_VALUE){

return new ArrayList<Integer>();

}

dp[n].list.add(n-1);

for (int i = 0; i < dp[n].list.size(); i++){

dp[n].list.set(i, dp[n].list.get(i) + 1);

}

return dp[n].list;

}

}

## 657\_Judge.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Initially, there is a Robot at position (0, 0). Given a sequence of its moves, judge if this robot makes a circle, which means it moves back to the original place.

The move sequence is represented by a string. And each move is represent by a character. The valid robot moves are R (Right), L (Left), U (Up) and D (down). The output should be true or false representing whether the robot makes a circle.

Example 1:

Input: "UD"

Output: true

Example 2:

Input: "LL"

Output: false

class Solution {

public boolean judgeCircle(String moves) {

int x = 0;

int y = 0;

for (int i = 0; i < moves.length(); i++){

char c = moves.charAt(i);

if (c == 'U'){

y++;

}else if (c == 'D'){

y--;

}else if (c == 'L'){

x--;

}else if (c == 'R'){

x++;

}

}

return x == 0 && y == 0;

}

}

class Solution {

public boolean judgeCircle(String moves) {

int x = 0;

int y = 0;

for (char c : moves.toCharArray()){

if (c == 'U'){

x--;

}else if (c == 'D'){

x++;

}else if (c == 'R'){

y++;

}else{

y--;

}

}

return x == 0 && y == 0;

}

}

## 658\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted array, two integers k and x, find the k closest elements to x in the array. The result should also be sorted in

ascending order. If there is a tie, the smaller elements are always preferred.

Example 1:

Input: [1,2,3,4,5], k=4, x=3

Output: [1,2,3,4]

Example 2:

Input: [1,2,3,4,5], k=4, x=-1

Output: [1,2,3,4]

Note:

The value k is positive and will always be smaller than the length of the sorted array.

Length of the given array is positive and will not exceed 104

Absolute value of elements in the array and x will not exceed 104

binary search here is used to find the best left bound index rather than finding the closest element to x

class Solution {

public List<Integer> findClosestElements(int[] arr, int k, int x) {

List<Integer> res = new ArrayList<>();

int start = 0;

int end = arr.length - k;

while (start < end){

int mid = start + (end - start) / 2;

if (x - arr[mid] > arr[mid+k] - x){

start = mid + 1;

}else{

end = mid;

}

}

for (int i = 0; i < k; i++){

res.add(arr[start+i]);

}

return res;

}

}

Better solution:

class Solution {

public List<Integer> findClosestElements(int[] arr, int k, int x) {

List<Integer> res = new ArrayList<>();

if (arr.length == 1){

res.add(arr[0]);

return res;

}

int start = 0;

int end = arr.length - k;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (Math.abs(x - arr[mid]) > Math.abs(arr[mid+k] - x)){

start = mid;

}else{

end = mid;

}

}

int index = start;

if (Math.abs(x-arr[start]) > Math.abs(arr[start+k] - x)){

index = end;

}

for (int i = 0; i < k; i++){

res.add(arr[index+i]);

}

return res;

}

}

## 658\_FindKClosestElements.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted array, two integers k and x, find the k closest elements to x in the array.

The result should also be sorted in ascending order. If there is a tie, the smaller elements are always preferred.

Example 1:

Input: [1,2,3,4,5], k=4, x=3

Output: [1,2,3,4]

Example 2:

Input: [1,2,3,4,5], k=4, x=-1

Output: [1,2,3,4]

Note:

The value k is positive and will always be smaller than the length of the sorted array.

Length of the given array is positive and will not exceed 104

Absolute value of elements in the array and x will not exceed 104

Method 1:

Time complexity: O(nlogn)

class Solution {

public List<Integer> findClosestElements(int[] arr, int k, int x) {

List<Integer> result = new ArrayList<>();

int[] resultArr = new int[k];

Queue<Integer> maxHeap = new PriorityQueue<Integer>(new Comparator<Integer>(){

public int compare(Integer a, Integer b){

if (Math.abs(b-x) != Math.abs(a-x) ){

return Math.abs(b-x) - Math.abs(a-x);

}

return b-a;

}

});

for (int i = 0; i < arr.length; i++){

maxHeap.offer(arr[i]);

if (maxHeap.size() > k){

maxHeap.poll();

}

}

int index = 0;

while (!maxHeap.isEmpty()){

resultArr[index] = maxHeap.poll();

index++;

}

Arrays.sort(resultArr);

for (int i = 0; i < k; i++){

result.add(resultArr[i]);

}

return result;

}

}

## 659\_Split.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Intuition

Call a chain a sequence of 3 or more consecutive numbers.

Considering numbers x from left to right, if x can be added to a current chain, it's at least as good to add x to that chain first,

rather than to start a new chain.

Why? If we started with numbers x and greater from the beginning, the shorter chains starting from x could be concatenated with the

chains ending before x, possibly helping us if there was a "chain" from x that was only length 1 or 2.

Algorithm

Say we have a count of each number, and let tails[x] be the number of chains ending right before x.

Now let's process each number. If there's a chain ending before x, then add it to that chain. Otherwise, if we can start a new chain,

do so.

It's worth noting that our solution can be amended to take only O(1)O(1)O(1) additional space, since we could do our counts similar

to Approach #1, and we only need to know the last 3 counts at a time.

1. We iterate through the array once to get the frequency of all the elements in the array

2. We iterate through the array once more and for each element we either see if it can be appended to a previously constructed

consecutive sequence or if it can be the start of a new consecutive sequence. If neither are true, then we return false.

Key Note:

appending to existing array has higher priority than creating one array

Greedy

public boolean isPossible(int[] nums) {

Map<Integer, Integer> freq = new HashMap<>(), appendfreq = new HashMap<>();

for (int i : nums) freq.put(i, freq.getOrDefault(i,0) + 1);

for (int i : nums) {

if (freq.get(i) == 0) continue;

else if (appendfreq.getOrDefault(i,0) > 0) {

appendfreq.put(i, appendfreq.get(i) - 1);

appendfreq.put(i+1, appendfreq.getOrDefault(i+1,0) + 1);

}

else if (freq.getOrDefault(i+1,0) > 0 && freq.getOrDefault(i+2,0) > 0) {

freq.put(i+1, freq.get(i+1) - 1);

freq.put(i+2, freq.get(i+2) - 1);

appendfreq.put(i+3, appendfreq.getOrDefault(i+3,0) + 1);

}

else return false;

freq.put(i, freq.get(i) - 1);

}

return true;

}

class Solution {

public boolean isPossible(int[] nums) {

Map<Integer, Integer> freq = new HashMap<>();

Map<Integer, Integer> append = new HashMap<>();

for (int num : nums){

freq.put(num, freq.getOrDefault(num, 0) + 1);

}

for (int i = 0; i < nums.length; i++){

if (freq.get(nums[i]) == 0){

continue;

}

if (append.getOrDefault(nums[i], 0) > 0){

append.put(nums[i], append.get(nums[i]) - 1);

append.put(nums[i] + 1, append.getOrDefault(nums[i] + 1, 0) + 1);

}else if (freq.getOrDefault(nums[i] + 1, 0) > 0 && freq.getOrDefault(nums[i] + 2, 0) > 0){

freq.put(nums[i] + 1, freq.get(nums[i] + 1) - 1);

freq.put(nums[i] + 2, freq.get(nums[i] + 2) - 1);

append.put(nums[i] + 3, append.getOrDefault(nums[i] + 3, 0) + 1);

}else{

return false;

}

freq.put(nums[i], freq.get(nums[i]) - 1);

}

return true;

}

}

思路：

用freq map先过一遍存频率，再建一个map存我们能用到的tail number。再过第二遍的时候，若freq==0 continue；若能接上前面的顺子，就接；不能则新开一个顺子（记住新开时候直接要把连着的两个数字剔除，因为要保证长度为三）；都不行则为false。记住最后别忘了更新当前频率

对于每一个element，我们有两种选择

1. 把它加入之前构造好的顺子中

2. 用它新开一个顺子

此处用贪心策略，如果1能满足总是先满足1，因为新开顺子可能失败，即使新开顺子成功，当1能满足的时候，将新开顺子加入之前的顺子也能成功，所以能够选择策略1的时候没必要冒风险选择策略2

目标是用策略1或者2消耗掉所有的元素

如果两个策略都无法选择，直接返回false

用另一个map记录已经构造好的顺子中现在需要哪些尾巴，来实现将当前元素加入构造好的顺子中

## 659\_StringsSerialization.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded

back to the original list of strings.

Please implement encode and decode

Have you met this question in a real interview? Yes

Example

Given strs = ["lint","code","love","you"]

string encoded\_string = encode(strs)

return ["lint","code","love","you"] when you call decode(encoded\_string)

Method 1: length code

public class Solution {

/\*

\* @param strs: a list of strings

\* @return: encodes a list of strings to a single string.

\*/

public String encode(List<String> strs) {

String ans = "";

for (String str : strs){

ans += String.valueOf(str.length()) + ":" + str;

}

return ans;

}

/\*

\* @param str: A string

\* @return: dcodes a single string to a list of strings

\*/

public List<String> decode(String str) {

List<String> result = new ArrayList<String>();

int i = 0;

int numPtr = 0;

while (i < str.length()){

if (str.charAt(i) == ':'){

String num = str.substring(numPtr, i);

int len = Integer.parseInt(num);

String temp = str.substring(i+1, i + 1 + len);

result.add(temp);

i += len + 1;

numPtr = i;

}else{

i++;

}

}

return result;

}

}

Method 2: length code (best solution)

public class Solution {

/\*\*

\* @param strs a list of strings

\* @return encodes a list of strings to a single string.

\*/

public String encode(List<String> strs) {

// Write your code here

StringBuilder sb = new StringBuilder();

StringBuilder number = new StringBuilder();

for (String s: strs){

number.append("/").append(s.length());

sb.append(s);

}

return sb.toString() + "+" + number.toString();

}

/\*\*

\* @param str a string

\* @return dcodes a single string to a list of strings

\*/

public List<String> decode(String str) {

// Write your code here

// have string like this

// lintcodelikeyou + /4/4/4/3

int index = str.lastIndexOf("+");

String prefix = str.substring(0, index);

String postfix = str.substring(index+1);

List<String> res = new ArrayList<>();

int pointer = 0;

String[] number = postfix.split("/");

/\* for (String num: number){

if (num.equals("")) continue; //note the first num is empty string

int temp = Integer.parseInt(num);

res.add(prefix.substring(pointer, pointer + temp));

pointer += temp;

}\*/

for (int i = 1; i < number.length; i++){

int temp = Integer.parseInt(number[i]);

res.add(prefix.substring(pointer, pointer + temp));

pointer += temp;

}

return res;

}

}

Method 3: translation code

public class Solution {

/\*

\* @param strs: a list of strings

\* @return: encodes a list of strings to a single string.

\*/

public String encode(List<String> strs) {

String ans = "";

for (String str : strs){

for (char c : str.toCharArray()){

if (c == ':'){

ans += "::";

}else{

ans += c;

}

}

ans += ":;";

}

return ans;

}

/\*

\* @param str: A string

\* @return: dcodes a single string to a list of strings

\*/

public List<String> decode(String str) {

List<String> result = new ArrayList<>();

String item = "";

char[] strChar = str.toCharArray();

int i = 0;

while (i < str.length()){

if (strChar[i] == ':'){

if (strChar[i+1] == ';'){

result.add(item);

item = "";

i += 2;

}else{

item += strChar[i+1];

i += 2;

}

}else{

item += strChar[i];

i++;

}

}

return result;

}

}

## 66\_PlusOne.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-negative integer represented as a non-empty array of digits, plus one to the integer.

You may assume the integer do not contain any leading zero, except the number 0 itself.

The digits are stored such that the most significant digit is at the head of the list.

class Solution {

public int[] plusOne(int[] digits) {

for(int i = digits.length - 1; i >= 0; i--){

if (digits[i] < 9){

digits[i]++;

return digits;

}

digits[i] = 0;

}

int[] result = new int[digits.length+1];

result[0] = 1;

return result;

}

}

class Solution {

public int[] plusOne(int[] digits) {

int carry = 1;

for (int i = digits.length - 1; i >= 0; i--){

int num = digits[i] + carry;

digits[i] = num % 10;

carry = num / 10;

}

if (carry == 1){

int[] res = new int[digits.length+1];

res[0] = 1;

for (int i = 1; i < res.length; i++){

res[i] = digits[i-1];

}

return res;

}

return digits;

}

}

## 660\_ReadCharactersFromFile.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The API: int read4(char \*buf) reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the read4 API, implement the function int read(char \*buf, int n) that reads n characters from the file.

Notice

The read function may be called multiple times.

/\* The read4 API is defined in the parent class Reader4.

int read4(char[] buf); \*/

思路:

• 做一个buffer,类似内存从磁盘读数据

• buffer是一个队列:

– 队列先进先出可以保持顺序不变

– 队列为空时就进队(read4)

– 队列不为空时就出队,并把出队的元素放到答案中

public class Solution extends Reader4 {

/\*\*

\* @param buf destination buffer

\* @param n maximum number of characters to read

\* @return the number of characters read

\*/

char[] buffer = new char[4];

int bufferPrt = 0;

int bufferCnt = 0;

public int read(char[] buf, int n) {

int ptr = 0;

while (ptr < n){

if (bufferPrt == bufferCnt){

bufferCnt = read4(buffer);

bufferPrt = 0;

}

if (bufferCnt == 0){

break;

}

while (ptr < n && bufferPrt < bufferCnt){

buf[ptr++] = buffer[bufferPrt++];

}

}

return ptr;

}

}

## 660\_Remove.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Start from integer 1, remove any integer that contains 9 such as 9, 19, 29...

So now, you will have a new integer sequence: 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, ...

Given a positive integer n, you need to return the n-th integer after removing. Note that 1 will be the first integer.

Example 1:

Input: 9

Output: 10

Hint: n will not exceed 9 x 10^8.

class Solution {

public int newInteger(int n) {

int res = 0;

int base = 1;

while (n > 0){

res += n % 9 \* base;

n /= 9;

base \*= 10;

}

return res;

}

}

class Solution {

public int newInteger(int n) {

return Integer.parseInt(Integer.toString(n, 9));

}

}

the above change base is just a way to count... the actual reason why it works is ... every 10 number we have 9 numbers without '9' digit; every 100 number we have 9x9 numbers without '9' digit; 1000 number we have 9x9x9 numbers without '9' digit; .... did you see the pattern?

this pattern is essentially means we want a number n represent in base 9, what would the value of the same digit then if represent in base 10... so the above 1 lines change base method helps calculate this...

## 661\_ConverBSTtoGreaterTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.

Have you met this question in a real interview? Yes

Example

Given a binary search Tree `{5,2,13}｀:

5

/ \

2 13

Return the root of new tree

18

/ \

20 13

/\*\*

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\*/

算法流程:

1. 定义累计求和变量sum=0

2. 按照右中左这样的“反中根遍历”,依次访问每个节点 1 更新累计求和变量 sum = sum + 当前节点的值

2 更新当前节点的值 = sum

按照DFS的定义宏观理解:

我们这里DFS函数的定义是什么?任务是什么?

1. “反中根遍历”的顺序访问cur这颗树每个节点

2. sum要在当前的基础上要加上所有访问元素的值

3. 所有访问元素要变成大于等于它的元素之和

public class Solution {

/\*

\* @param root: the root of binary tree

\* @return: the new root

\*/

int sum = 0;

public TreeNode convertBST(TreeNode root) {

dfs(root);

return root;

}

private void dfs(TreeNode cur){

if (cur == null){

return;

}

dfs(cur.right);

sum += cur.val;

cur.val = sum;

dfs(cur.left);

}

}

## 661\_ImageSmoother.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2D integer matrix M representing the gray scale of an image, you need to design a

smoother to make the gray scale of each cell becomes the average gray scale (rounding down)

of all the 8 surrounding cells and itself. If a cell has less than 8 surrounding cells, then use as many as you can.

Example 1:

Input:

[[1,1,1],

[1,0,1],

[1,1,1]]

Output:

[[0, 0, 0],

[0, 0, 0],

[0, 0, 0]]

Explanation:

For the point (0,0), (0,2), (2,0), (2,2): floor(3/4) = floor(0.75) = 0

For the point (0,1), (1,0), (1,2), (2,1): floor(5/6) = floor(0.83333333) = 0

For the point (1,1): floor(8/9) = floor(0.88888889) = 0

Note:

The value in the given matrix is in the range of [0, 255].

The length and width of the given matrix are in the range of [1, 150].

class Solution {

public int[][] imageSmoother(int[][] M) {

if (M == null || M.length == 0 || M[0].length == 0){

return M;

}

int m = M.length;

int n = M[0].length;

int[][] result = new int[m][n];

int[] dx = {0, 1, 1, 0, -1, -1, -1, 0, 1};

int[] dy = {0, 0, 1, 1, 1, 0, -1, -1, -1};

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

int count = 0;

for (int k = 0; k < dx.length; k++){

int cx = i + dx[k];

int cy = j + dy[k];

if (0 <= cx && cx < m && 0 <= cy && cy < n){

count++;

result[i][j] += M[cx][cy];

}

}

result[i][j] = (int) (result[i][j] / count);

}

}

return result;

}

}

## 662\_MaximumWidthofBinaryTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, write a function to get the maximum width of the given tree. The width of a tree is the maximum width among all levels. The binary tree has the same structure as a full binary tree, but some nodes are null.

The width of one level is defined as the length between the end-nodes (the leftmost and right most non-null nodes in the level, where the null nodes between the end-nodes are also counted into the length calculation.

Example 1:

Input:

1

/ \

3 2

/ \ \

5 3 9

Output: 4

Explanation: The maximum width existing in the third level with the length 4 (5,3,null,9).

Example 2:

Input:

1

/

3

/ \

5 3

Output: 2

Explanation: The maximum width existing in the third level with the length 2 (5,3).

Example 3:

Input:

1

/ \

3 2

/

5

Output: 2

Explanation: The maximum width existing in the second level with the length 2 (3,2).

Example 4:

Input:

1

/ \

3 2

/ \

5 9

/ \

6 7

Output: 8

Explanation:The maximum width existing in the fourth level with the length 8 (6,null,null,null,null,null,null,7).

Note: Answer will in the range of 32-bit signed integer.

Use heap index:

Regardless whether these nodes exist:

Always make the id of left child as parent\_id \* 2;

Always make the id of right child as parent\_id \* 2 + 1;

change the val of node to be the index to save space. The value is useless. All we need is just the index.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int widthOfBinaryTree(TreeNode root) {

if (root == null){

return 0;

}

int max = 0;

Deque<TreeNode> deque = new LinkedList<>(); //deque

root.val = 0;

deque.offer(root);

while (!deque.isEmpty()){

int size = deque.size();

max = Math.max(max, deque.peekLast().val - deque.peekFirst().val + 1);

for (int i = 0; i < size; i++){

TreeNode node = deque.poll();

if (node.left != null){

node.left.val = node.val \* 2;

deque.offer(node.left);

}

if (node.right != null){

node.right.val = node.val \* 2 + 1;

deque.offer(node.right);

}

}

}

return max;

}

}

Method 2:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int widthOfBinaryTree(TreeNode root) {

if (root == null){

return 0;

}

Queue<TreeNode> queue = new LinkedList<>();

Queue<Integer> count = new LinkedList<>();

int max = 1;

queue.offer(root);

count.offer(1);

while (!queue.isEmpty()){

int size = queue.size();

int left = 0;

int right = 0;

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

Integer index = count.poll();

if (i == 0){

left = index;

}

if (i == size - 1){

right = index;

}

if (node.left != null){

queue.offer(node.left);

count.offer(index\*2);

}

if (node.right != null){

queue.offer(node.right);

count.offer(index\*2+1);

}

}

max = Math.max(max, right - left + 1);

}

return max;

}

}

## 663\_EqualTreePartition.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree with n nodes, your task is to check if it's possible to partition the tree to two trees which have the equal

sum of values after removing exactly one edge on the original tree.

Example 1:

Input:

5

/ \

10 10

/ \

2 3

Output: True

Explanation:

5

/

10

Sum: 15

10

/ \

2 3

Sum: 15

Method 1: DFS: HashMap

Best solution

class Solution {

public boolean checkEqualTree(TreeNode root) {

Map<Integer, Integer> map = new HashMap<>();

int sum = dfs(root, map);

if (sum == 0){

return map.get(sum) > 1;

}

if (sum % 2 != 0){

return false;

}

return map.containsKey(sum / 2);

}

private int dfs(TreeNode root, Map<Integer, Integer> map){

if (root == null){

return 0;

}

int curSum = dfs(root.left, map) + dfs(root.right, map) + root.val;

map.put(curSum, map.getOrDefault(curSum, 0) + 1);

return curSum;

}

}

Method 2: Iteration Stack

class Solution {

Stack<Integer> seen;

public boolean checkEqualTree(TreeNode root) {

seen = new Stack();

int total = sum(root);

seen.pop();

if (total % 2 == 0)

for (int s: seen)

if (s == total / 2)

return true;

return false;

}

public int sum(TreeNode node) {

if (node == null) return 0;

seen.push(sum(node.left) + sum(node.right) + node.val);

return seen.peek();

}

}

Method 3: DFS( can't pass some cases as below)

0

/ \

-1 1

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

boolean isHalf = false;

public boolean checkEqualTree(TreeNode root) {

if (root == null){

return false;

}

int total = getTotal(root);

checkSum(root, total);

return isHalf;

}

private int getTotal(TreeNode root){

if (root == null){

return 0;

}

int left = getTotal(root.left);

int right = getTotal(root.right);

return left + right + root.val;

}

private int checkSum(TreeNode root,int total){

if (root == null || isHalf){

return 0;

}

int left = checkSum(root.left, total);

int right = checkSum(root.right, total);

int sum = left + right + root.val;

if (sum \* 2 == total){

isHalf = true;

return 0;

}

return sum;

}

}

## 663\_NearestExit.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a m x n 2D grid initialized with these three possible values.

-1 - A wall or an obstacle.

0 - A gate.

INF - Infinity means an empty room. We use the value 2^31 - 1 = 2147483647 to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF.

Have you met this question in a real interview? Yes

Example

Given the 2D grid:

INF -1 0 INF

INF INF INF -1

INF -1 INF -1

0 -1 INF INF

return the result:

3 -1 0 1

2 2 1 -1

1 -1 2 -1

0 -1 3 4

思路:

￼• 多源点单终点 == 单源点多终点,最短路常用转化套路

• 多个门,多源点多终点怎么办?

– 多个源点同时注水,看谁流得快

• 多源点同时注水,其实相当于增加了一个超级源点(dummy),超级源点 连接每个普通源点一条边

• 增加了超级源后,其实相当于从超级源的单源最短路

多源多终点 ==> 单源多终点 (增加超级源,最短路常用转化套路)

 小技巧总结:

• 多源点单终点 单源点多终点,最短路常用转化套路

• 多源多终点单源多终点 (增加超级源,最短路常用转化套路) • BFS可以求边长=1的图的最短路(如此题的棋盘图)

public class Solution {

/\*

\* @param rooms: m x n 2D grid

\* @return: nothing

\*/

public void wallsAndGates(int[][] rooms) {

int n = rooms.length;

if (n == 0){

return;

}

int m = rooms[0].length;

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

for (int i = 0; i < n; i++){

for (int j = 0; j < m; j++){

if (rooms[i][j] == 0){

qx.offer(i);

qy.offer(j);

}

}

}

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

for (int i = 0; i < 4; i++){

int nx = cx + dx[i];

int ny = cy + dy[i];

if (nx >= 0 && nx < n && ny >= 0 && ny < m

&& rooms[nx][ny] == Integer.MAX\_VALUE){

rooms[nx][ny] = rooms[cx][cy] + 1;

qx.offer(nx);

qy.offer(ny);

}

}

}

}

}

## 665\_Nondecreasing.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array with n integers, your task is to check if it could become non-decreasing by modifying at most 1 element.

We define an array is non-decreasing if array[i] <= array[i + 1] holds for every i (1 <= i < n).

Example 1:

Input: [4,2,3]

Output: True

Explanation: You could modify the first 4 to 1 to get a non-decreasing array.

Example 2:

Input: [4,2,1]

Output: False

Explanation: You can't get a non-decreasing array by modify at most one element.

Best solution:

Try 3, 4, 2, 3

need left to catch: 2, 4, 2, 3

class Solution {

public boolean checkPossibility(int[] nums) {

int left = 0;

int right = nums.length-1;

int asc = 0;

int des = 0;

for (int i = 0; i < nums.length; i++){

if (nums[left] <= nums[i]){

left = i;

}else{

asc++;

}

}

for (int i = nums.length - 1; i>= 0; i--){

if (nums[right] >= nums[i]){

right = i;

}else{

des++;

}

}

if (asc > 1 && des > 1){

return false;

}

return true;

}

}

提供一种略有不同的方法。

顺序检查凹变段和逆序检查凸变段。

如果满足，则asc和desc中的较小值必然不大于1。

时间开销O(n)，空间开销O(1)，缺点是双向检查，优点是便于理解o(╯□╰)o。

class Solution {

public boolean checkPossibility(int[] nums) {

for(int i = 0, m = 0, n = nums.length - 1, asc = 0, desc = 0;i < nums.length;i++) {

if(nums[m] <= nums[i]) m = i;

else asc++;

if(nums[n] >= nums[nums.length - 1 - i]) n = nums.length - 1 - i;

else desc++;

if(asc > 1 && desc > 1) return false;

}

return true;

}

}

class Solution {

public boolean checkPossibility(int[] nums) {

int count = 0;

for (int i = 1; i < nums.length; i++){

if (nums[i-1] > nums[i]){

count++;

if (count >= 2){

return false;

}

if (i < 2 || nums[i-2] <= nums[i]){

nums[i-1] = nums[i];

}else{

nums[i] = nums[i-1];

}

}

}

return true;

}

}

https://leetcode.com/problems/non-decreasing-array/discuss/106826/JavaC++-Simple-greedy-like-solution-with-explanation

The problem requires that every number has to be equal or greater than previous number.

If we encounter a failing condition where the number is not greater or equal to previous (smaller than previous) we need to make a correction. Correction can be made in either of two ways:

Make the previous number smaller or equal to current number

Make the current number equal to previous number

We can do (1) as long as the number at position i-2 is equal or lower than the current element. (if i-2 is valid)

In case 1 below we can do this at (3) (i = 2) as the element 1 (i = 0) fulfills 1 <= 3. We can replace 7 with 3.

However, this cannot be done in case 2 as 4 <= 3 does not satisfy.

Correction with technique (1) takes priority as there is no risk in lowering the value but there is a risk associated if the value is increased. (Consider scenario in case 1 if we replace 3 with 7, it will fail to satisfy the condition for the last element)

We have to make correction with (2) if we cannot achieve it by (1). In which case we increase the value of current element by matching previous element. In case 2, we replace 3 with 7.

Also we only compare condition with the previous element only because as we move forward we know the previous numbers are already validated .

Case 1:

7

/\ 4

/ \ /

/ \/

/ 3

1

Case 2:

9

/

7 /

/\ /

/ \/

/ 3

4

## 666\_Path.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If the depth of a tree is smaller than 5, then this tree can be represented by a list of three-digits integers.

For each integer in this list:

The hundreds digit represents the depth D of this node, 1 <= D <= 4.

The tens digit represents the position P of this node in the level it belongs to, 1 <= P <= 8. The position is the same

as that in a full binary tree.

The units digit represents the value V of this node, 0 <= V <= 9.

Given a list of ascending three-digits integers representing a binary with the depth smaller than 5.

You need to return the sum of all paths from the root towards the leaves.

Example 1:

Input: [113, 215, 221]

Output: 12

Explanation:

The tree that the list represents is:

3

/ \

5 1

The path sum is (3 + 5) + (3 + 1) = 12.

Example 2:

Input: [113, 221]

Output: 4

Explanation:

The tree that the list represents is:

3

\

1

The path sum is (3 + 1) = 4.

Tree traversal

How do we solve problem like this if we were given a normal tree? Yes, traverse it, keep a root to leaf running sum.

If we see a leaf node (node.left == null && node.right == null), we add the running sum to the final result.

Now each tree node is represented by a number. 1st digits is the level, 2nd is the position in that level

(note that it starts from 1 instead of 0). 3rd digit is the value. We need to find a way to traverse this tree and get the sum.

The idea is, we can form a tree using a HashMap. The key is first two digits which marks the position of a node in the tree.

The value is value of that node. Thus, we can easily find a node's left and right children using math.

Formula: For node xy? its left child is (x+1)(y\*2-1)? and right child is (x+1)(y\*2)?

class Solution {

int sum = 0;

public int pathSum(int[] nums) {

Map<Integer, Integer> tree = new HashMap<>();

for (int num : nums){

int key = num / 10;

int val = num % 10;

tree.put(key, val);

}

traverse(nums[0] / 10, 0, tree);

return sum;

}

private void traverse(int root, int preSum, Map<Integer, Integer> tree){

int level = root / 10;

int pos = root % 10;

int left = (level + 1) \* 10 + (pos \* 2 - 1);

int right = (level + 1) \* 10 + pos \* 2;

int curSum = preSum + tree.get(root);

if (!tree.containsKey(left) && !tree.containsKey(right)){

sum += curSum;

return;

}

if (tree.containsKey(left)){

traverse(left, curSum, tree);

}

if (tree.containsKey(right)){

traverse(right, curSum, tree);

}

}

}

## 668\_Kth.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Nearly every one have used the Multiplication Table. But could you find out the k-th smallest number quickly from the

multiplication table?

Given the height m and the length n of a m \* n Multiplication Table, and a positive integer k, you need to return the

k-th smallest number in this table.

Example 1:

Input: m = 3, n = 3, k = 5

Output:

Explanation:

The Multiplication Table:

1 2 3

2 4 6

3 6 9

The 5-th smallest number is 3 (1, 2, 2, 3, 3).

Example 2:

Input: m = 2, n = 3, k = 6

Output:

Explanation:

The Multiplication Table:

1 2 3

2 4 6

The 6-th smallest number is 6 (1, 2, 2, 3, 4, 6).

Note:

The m and n will be in the range [1, 30000].

The k will be in the range [1, m \* n]

Method 1: Binary Search

Time complexity: O(m\*log(m\*n)

Space complexity: O(1)

class Solution {

public int findKthNumber(int m, int n, int k) {

int low = 1;

int high = m \* n;

while (low <= high){

int mid = low + (high - low)/ 2;

int count = getLessEqual(m, n, mid);

if (count < k){

low = mid + 1;

}else{

high = mid - 1;

}

}

return low;

}

private int getLessEqual(int m, int n, int val){

int res = 0;

for (int i = 1; i <= m; i++){

res += Math.min(val / i, n);

}

return res;

}

}

class Solution {

public int findKthNumber(int m, int n, int k) {

int low = 1;

int high = m \* n;

while (low <= high){

int mid = low + (high - low)/ 2;

int count = getLessEqual(m, n, mid);

if (count < k){

low = mid + 1;

}else{

high = mid - 1;

}

}

return low;

}

private int getLessEqual(int m, int n, int val){

int res = 0;

for (int i = 1; i <= m; i++){

int low = 1;

int high = n;

while (low <= high){

int mid = low + (high - low) / 2;

if (mid \* i <= val){

low = mid + 1;

}else{

high = mid - 1;

}

}

res += low - 1;

}

return res;

}

}

Method 2: Heap TLE

Time complexity: O(k\*m\*logm)

Space complexity: O(m)

class Solution {

class Pair {

int x;

int y;

public Pair (int x, int y){

this.x = x;

this. y = y;

}

}

public int findKthNumber(int m, int n, int k) {

Queue<Pair> minPQ = new PriorityQueue<>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p1.x \* p1.y - p2.x \* p2.y;

}

});

for (int i = 1; i <= m; i++){

minPQ.offer(new Pair(i, 1));

}

for (int i = 0; i < k - 1; i++){

Pair p = minPQ.poll();

if (p.y < n){

minPQ.offer(new Pair(p.x, p.y + 1));

}

}

Pair res = minPQ.poll();

return res.x \* res.y;

}

}

## 669\_Trim.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary search tree and the lowest and highest boundaries as L and R, trim the tree so that all its elements lies in [L, R]

(R >= L). You might need to change the root of the tree, so the result should return the new root of the trimmed binary search tree.

Example 1:

Input:

1

/ \

0 2

L = 1

R = 2

Output:

1

\

2

Example 2:

Input:

3

/ \

0 4

\

2

/

1

L = 1

R = 3

Output:

3

/

2

/

1

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode trimBST(TreeNode root, int L, int R) {

if (root == null){

return null;

}

if (root.val == L){

root.right = trimBST(root.right, L, R);

root.left = null;

return root;

}

if (root.val == R){

root.left = trimBST(root.left, L, R);

root.right = null;

return root;

}

if (L < root.val && root.val < R){

root.left = trimBST(root.left, L, root.val);

root.right = trimBST(root.right, root.val, R);

return root;

}

if (L > root.val){

return trimBST(root.right, L, R);

}

if (R < root.val){

return trimBST(root.left, L, R);

}

return null;

}

}

class Solution {

public TreeNode trimBST(TreeNode root, int L, int R) {

if (root == null){

return null;

}

if (L > root.val){

return trimBST(root.right, L, R);

}

if (R < root.val){

return trimBST(root.left, L, R);

}

root.left = trimBST(root.left, L, root.val);

root.right = trimBST(root.right, root.val, R);

return root;

}

}

Best solutoin:

class Solution {

public TreeNode trimBST(TreeNode root, int L, int R) {

if (root == null) return root;

if (root.val > R) return trimBST(root.left, L, R);

if (root.val < L) return trimBST(root.right, L, R);

root.left = trimBST(root.left, L, R);

root.right = trimBST(root.right, L, R);

return root;

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode trimBST(TreeNode root, int L, int R) {

if (root == null){

return root;

}

if (root.val > R){

return trimBST(root.left, L, R);

}else if (root.val < L){

return trimBST(root.right, L, R);

}else if (root.val == R){

root.right = null;

root.left = trimBST(root.left, L, R);

return root;

}else if (root.val == L){

root.left = null;

root.right = trimBST(root.right, L, R);

return root;

}

root.left = trimBST(root.left, L, R);

root.right = trimBST(root.right, L, R);

return root;

}

}

Complexity Analysis

Time Complexity: O(N), where N is the total number of nodes in the given tree. We visit each node at most once.

Space Complexity: O(N). Even though we don't explicitly use any additional memory, the call stack of our recursion could be as large as the number of nodes in the worst case.

# 670\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-negative integer, you could swap two digits at most once to get the maximum valued number. Return the maximum valued number you could get.

Example 1:

Input: 2736

Output: 7236

Explanation: Swap the number 2 and the number 7.

Example 2:

Input: 9973

Output: 9973

Explanation: No swap.

Note:

The given number is in the range [0, 108]

Method 1:

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public int maximumSwap(int num) {

String str = String.valueOf(num);

int n = str.length();

char[] charArray = str.toCharArray();

for (int i = 0; i < n; i++){

int start = (int) (charArray[i] - '0');

int maxIndex = i;

int max = start;

for (int j = i+1; j < n; j++){

int digit = (int) (charArray[j] - '0');

if (digit >= max){

max = digit;

maxIndex = j;

}

}

if (max > start){

char temp = charArray[i];

charArray[i] = charArray[maxIndex];

charArray[maxIndex] = temp;

break;

}

}

return Integer.parseInt(String.valueOf(charArray));

}

}

class Solution {

public int maximumSwap(int num) {

String str = String.valueOf(num);

int n = str.length();

char[] charArray = str.toCharArray();

int maxIndex = n - 1;

int minIndex = 0;

boolean foundMax = false;

boolean foundMin = false;

for (int i = 0; i < n; i++){

int max = 0;

for (int j = n - 1; j >= i; j--){

int digitMax = (int) (charArray[j] - '0');

System.out.println("digitMax:" + i + digitMax);

System.out.println("max:" + i + max);

if (digitMax > max){

max = digitMax;

maxIndex = j;

foundMax = true;

}

}

for (int j = i; j < maxIndex; j++){

int digitMin = (int) (charArray[j] - '0');

if (digitMin < max){

minIndex = j;

foundMin = true;

break;

}

}

if (foundMax && foundMin){

char temp = charArray[maxIndex];

charArray[maxIndex] = charArray[minIndex];

charArray[minIndex] = temp;

break;

}

}

return Integer.parseInt(String.valueOf(charArray));

}

}

Method 2:

Time complexity: O(n)

Space complexity: O(n)

Intuition

At each digit of the input number in order, if there is a larger digit that occurs later, we know that the best swap must occur with

the digit we are currently considering.

Algorithm

We will compute last[d] = i\text{last[d] = i}last[d] = i, the index i\text{i}i of the last occurrence of digit d\text{d}d

(if it exists).

Afterwards, when scanning the number from left to right, if there is a larger digit in the future, we will swap it with the largest

such digit; if there are multiple such digits, we will swap it with the one that occurs the latest.

class Solution {

public int maximumSwap(int num) {

char[] charArray = Integer.toString(num).toCharArray();

int[] last = new int[10];

for (int i = 0; i < charArray.length; i++){

last[charArray[i] - '0'] = i;

}

for (int i = 0; i < charArray.length; i++){

for (int d = 9; d > (charArray[i] - '0'); d--){

if (last[d] > i){

char temp = charArray[i];

charArray[i] = charArray[last[d]];

charArray[last[d]] = temp;

return Integer.valueOf(new String(charArray));

}

}

}

return num;

}

}

Method 3: Best solution

Time complexity: O(n)

https://leetcode.com/problems/maximum-swap/discuss/107102/Simple-AC-O(n)-java-solution-with-ex

class Solution {

public int maximumSwap(int num) {

//step 1: find the cutoff index which does not meet the reverse array and cut array into two parts

char[] charArr = Integer.toString(num).toCharArray();

int n = charArr.length;

int index = -1;

for (int i = 0; i < n-1; i++){

if (charArr[i] < charArr[i+1]){

index = i+1;

break;

}

}

if (index == -1){//already the largest

return num;

}

//step 2: find the LAST (i.e. 1993)largest value in the 2nd part

int max = -1;

int index2 = -1;

for (int i = index; i < n; i++){

int val = (int)(charArr[i] - '0');

if (val >= max){

max = val;

index2 = i;

}

}

//step 3: find the largest value in the 1st part which is slightly less than the max, and swap them

for (int i = 0; i < index; i++){

int val = (int)(charArr[i] - '0');

if (val < max){

char temp = charArr[index2];

charArr[index2] = charArr[i];

charArr[i] = temp;

break;

}

}

return Integer.parseInt(new String(charArr));

}

}

# 675\_CutOffTreesforGolfEvent.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are asked to cut off trees in a forest for a golf event. The forest is represented as a non-negative 2D map, in this map:

0 represents the obstacle can't be reached.

1 represents the ground can be walked through.

The place with number bigger than 1 represents a tree can be walked through, and this positive number represents

the tree's height.

You are asked to cut off all the trees in this forest in the order of tree's height - always cut off the tree with

lowest height first. And after cutting, the original place has the tree will become a grass (value 1).

You will start from the point (0, 0) and you should output the minimum steps you need to walk to cut off all the trees.

If you can't cut off all the trees, output -1 in that situation.

You are guaranteed that no two trees have the same height and there is at least one tree needs to be cut off.

Example 1:

Input:

[

[1,2,3],

[0,0,4],

[7,6,5]

]

Output: 6

Example 2:

Input:

[

[1,2,3],

[0,0,0],

[7,6,5]

]

Output: -1

Example 3:

Input:

[

[2,3,4],

[0,0,5],

[8,7,6]

]

Output: 6

Explanation: You started from the point (0,0) and you can cut off the tree in (0,0) directly without walking.

Hint: size of the given matrix will not exceed 50x50.

class Solution {

class Pair{

int x;

int y;

int height;

public Pair(int x, int y, int height){

this.x = x;

this.y = y;

this.height = height;

}

}

public int cutOffTree(List<List<Integer>> forest) {

if (forest == null || forest.size() == 0){

return -1;

}

int ans = 0;

Queue<Pair> pq = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p1.height - p2.height;

}

});

int m = forest.size();

int n = forest.get(0).size();

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (forest.get(i).get(j) > 1){

pq.offer(new Pair(i, j, forest.get(i).get(j)));

}

}

}

int txPrev = 0;

int tyPrev = 0;

int step = 0;

while (!pq.isEmpty()){

Pair pair = pq.poll();

int tx = pair.x;

int ty = pair.y;

if (tx == txPrev && ty == tyPrev){

step = 0;

}else{

step = bfs(tx, ty, m, n, txPrev, tyPrev, forest);

}

if (step == -1){

return -1;

}

ans += step;

txPrev = tx;

tyPrev = ty;

}

return ans;

}

private int bfs(int tx, int ty, int m, int n, int txPrev, int tyPrev, List<List<Integer>> forest){

int step = 0;

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

boolean[][] visited = new boolean[m][n];

visited[txPrev][tyPrev] = true;

qx.offer(txPrev);

qy.offer(tyPrev);

while (!qx.isEmpty()){

int size = qx.size();

step++;

for (int k = 0; k < size; k++){

int cx = qx.poll();

int cy = qy.poll();

for (int i = 0; i < dx.length; i++){

int nx = cx + dx[i];

int ny = cy + dy[i];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && forest.get(nx).get(ny) != 0){

if (nx == tx && ny == ty){

return step;

}

if (!visited[nx][ny]){

qx.offer(nx);

qy.offer(ny);

visited[nx][ny] = true;

}

}

}

}

}

return -1;

}

}

# 676\_Implement.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement a magic directory with buildDict, and search methods.

For the method buildDict, you'll be given a list of non-repetitive words to build a dictionary.

For the method search, you'll be given a word, and judge whether if you modify exactly one character into another character in this word, the modified word is in the dictionary you just built.

Example 1:

Input: buildDict(["hello", "leetcode"]), Output: Null

Input: search("hello"), Output: False

Input: search("hhllo"), Output: True

Input: search("hell"), Output: False

Input: search("leetcoded"), Output: False

Note:

You may assume that all the inputs are consist of lowercase letters a-z.

For contest purpose, the test data is rather small by now. You could think about highly efficient algorithm after the contest.

Please remember to RESET your class variables declared in class MagicDictionary, as static/class variables are persisted across multiple test cases. Please see here for more details.

Method 1: HashMap (faster)

class MagicDictionary {

Map<Integer, Set<String>> map;

/\*\* Initialize your data structure here. \*/

public MagicDictionary() {

map = new HashMap<>();

}

/\*\* Build a dictionary through a list of words \*/

public void buildDict(String[] dict) {

for (String word : dict){

int len = word.length();

if (!map.containsKey(len)){

map.put(len, new HashSet<>());

}

map.get(len).add(word);

}

}

/\*\* Returns if there is any word in the trie that equals to the given word after modifying exactly one character \*/

public boolean search(String word) {

int len = word.length();

if (!map.containsKey(len)){

return false;

}

Set<String> set = map.get(len);

for (String s : set){

int count = 0;

for (int i = 0; i < s.length(); i++){

if (s.charAt(i) != word.charAt(i)){

count++;

}

}

if (count == 1){

return true;

}

}

return false;

}

}

/\*\*

\* Your MagicDictionary object will be instantiated and called as such:

\* MagicDictionary obj = new MagicDictionary();

\* obj.buildDict(dict);

\* boolean param\_2 = obj.search(word);

\*/

Method 2: Trie (like BFS, search the entire trie)

class MagicDictionary {

class TrieNode {

TrieNode[] children;

boolean isWord;

public TrieNode (){

children = new TrieNode[26];

isWord = false;

}

}

TrieNode root;

/\*\* Initialize your data structure here. \*/

public MagicDictionary() {

root = new TrieNode();

}

/\*\* Build a dictionary through a list of words \*/

public void buildDict(String[] dict) {

for (String word : dict){

TrieNode node = root;

for (int i = 0 ;i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isWord = true;

}

}

/\*\* Returns if there is any word in the trie that equals to the given word after modifying exactly one character \*/

public boolean search(String word) {

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

for (int j = 0; j < 26; j++){

if ((char)(j + 'a') == c || node.children[j] == null){

continue;

}

if (isSame(node.children[j], word, i+1)){

return true;

}

}

node = node.children[c - 'a'];

if (node == null){

return false;

}

}

return false;

}

private boolean isSame(TrieNode node, String word, int start){

for (int i = start ; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return node.isWord;

}

}

/\*\*

\* Your MagicDictionary object will be instantiated and called as such:

\* MagicDictionary obj = new MagicDictionary();

\* obj.buildDict(dict);

\* boolean param\_2 = obj.search(word);

\*/

# 677\_Map.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement a MapSum class with insert, and sum methods.

For the method insert, you'll be given a pair of (string, integer). The string represents the key and the integer represents

the value. If the key already existed, then the original key-value pair will be overridden to the new one.

For the method sum, you'll be given a string representing the prefix, and you need to return the sum of all the pairs' value whose

key starts with the prefix.

Example 1:

Input: insert("apple", 3), Output: Null

Input: sum("ap"), Output: 3

Input: insert("app", 2), Output: Null

Input: sum("ap"), Output: 5

class MapSum {

class TrieNode{

TrieNode[] children;

int[] nums;

public TrieNode (){

children = new TrieNode[26];

nums = new int[26];

}

}

/\*\* Initialize your data structure here. \*/

TrieNode trie;

Map<String, Integer> map;

public MapSum() {

trie = new TrieNode();

map = new HashMap<String, Integer>();

}

public void insert(String key, int val) {

TrieNode cur = trie;

int delta = val - map.getOrDefault(key, 0);

map.put(key, val);

for (int i = 0; i < key.length(); i++){

if (cur.children[key.charAt(i) - 'a'] == null){

cur.children[key.charAt(i) - 'a'] = new TrieNode();

}

cur.nums[key.charAt(i) - 'a'] += delta;

cur = cur.children[key.charAt(i) - 'a'];

}

}

public int sum(String prefix) {

TrieNode cur = trie;

for (int i = 0; i < prefix.length() - 1; i++){

if (cur.children[prefix.charAt(i) - 'a'] != null){

cur = cur.children[prefix.charAt(i) - 'a'];

}else{

return 0;

}

}

return cur.nums[prefix.charAt(prefix.length() - 1) - 'a'];

}

}

Modified version: Better

class MapSum {

class TrieNode{

TrieNode[] children;

int score;

public TrieNode (){

children = new TrieNode[26];

}

}

/\*\* Initialize your data structure here. \*/

TrieNode trie;

Map<String, Integer> map;

public MapSum() {

trie = new TrieNode();

map = new HashMap<String, Integer>();

}

public void insert(String key, int val) {

TrieNode cur = trie;

int delta = val - map.getOrDefault(key, 0);

map.put(key, val);

for (int i = 0; i < key.length(); i++){

if (cur.children[key.charAt(i) - 'a'] == null){

cur.children[key.charAt(i) - 'a'] = new TrieNode();

}

cur = cur.children[key.charAt(i) - 'a'];

cur.score += delta;

}

}

public int sum(String prefix) {

TrieNode cur = trie;

for (int i = 0; i < prefix.length() ; i++){

if (cur.children[prefix.charAt(i) - 'a'] != null){

cur = cur.children[prefix.charAt(i) - 'a'];

}else{

return 0;

}

}

return cur.score;

}

}

class MapSum {

class TrieNode{

TrieNode[] children;

int num;

public TrieNode(){

children = new TrieNode[26];

num = 0;

}

}

/\*\* Initialize your data structure here. \*/

TrieNode root;

Map<String, Integer> map;

public MapSum() {

root = new TrieNode();

map = new HashMap<>();

}

public void insert(String key, int val) {

TrieNode node = root;

if (map.containsKey(key)){

val -= map.get(key);

}

map.put(key, val);

for (int i = 0; i < key.length(); i++){

char c = key.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

node.num += val;

}

}

public int sum(String prefix) {

TrieNode node = root;

for (int i = 0; i < prefix.length(); i++){

char c = prefix.charAt(i);

if (node.children[c - 'a'] != null){

node = node.children[c - 'a'];

}else{

return 0;

}

}

return node.num;

}

}

/\*\*

\* Your MapSum object will be instantiated and called as such:

\* MapSum obj = new MapSum();

\* obj.insert(key,val);

\* int param\_2 = obj.sum(prefix);

\*/

/\*\*

\* Your MapSum object will be instantiated and called as such:

\* MapSum obj = new MapSum();

\* obj.insert(key,val);

\* int param\_2 = obj.sum(prefix);

\*/

class MapSum {

HashMap<String, Integer> map;

TrieNode root;

public MapSum() {

map = new HashMap();

root = new TrieNode();

}

public void insert(String key, int val) {

int delta = val - map.getOrDefault(key, 0);

map.put(key, val);

TrieNode cur = root;

cur.score += delta;

for (char c: key.toCharArray()) {

cur.children.putIfAbsent(c, new TrieNode());

cur = cur.children.get(c);

cur.score += delta;

}

}

public int sum(String prefix) {

TrieNode cur = root;

for (char c: prefix.toCharArray()) {

cur = cur.children.get(c);

if (cur == null) return 0;

}

return cur.score;

}

}

class TrieNode {

Map<Character, TrieNode> children = new HashMap();

int score;

}

# 678\_ValidParenthesisString.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

https://leetcode.com/problems/valid-parenthesis-string/discuss/

Given a string containing only three types of characters: '(', ')' and '\*', write a function to check whether this string is valid. We define the validity of a string by these rules:

Any left parenthesis '(' must have a corresponding right parenthesis ')'.

Any right parenthesis ')' must have a corresponding left parenthesis '('.

Left parenthesis '(' must go before the corresponding right parenthesis ')'.

'\*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string.

An empty string is also valid.

Example 1:

Input: "()"

Output: True

Example 2:

Input: "(\*)"

Output: True

Example 3:

Input: "(\*))"

Output: True

Note:

The string size will be in the range [1, 100].

Method 1: Best solution

https://leetcode.com/problems/valid-parenthesis-string/discuss/107577/Short-Java-O(n)-time-O(1)-space-one-pass

low keeps the minimum number of UNBALANCED open braces

high keeps the maximum number of UNBALANCED open braces

class Solution {

public boolean checkValidString(String s) {

int low = 0;

int high = 0;

for (int i = 0; i < s.length(); i++){

if (s.charAt(i) == '('){

low++;

high++;

}else if (s.charAt(i) == ')'){

if (low > 0){

low--;

}

high--;

}else{

if (low > 0){

low--;

}

high++;

}

if (high < 0){

return false;

}

}

return low == 0;

}

}

Method 2: best solution too and easiest logic

https://leetcode.com/problems/valid-parenthesis-string/discuss/139759/Java-Very-easy-solution.-No-recursion-dp.

class Solution {

public boolean checkValidString(String s) {

int count = 0;

for (int i = 0; i < s.length(); i++){

if (s.charAt(i) == '(' || s.charAt(i) == '\*'){

count++;

}else{

count--;

}

if (count < 0){

return false;

}

}

//not necessary to have these three lines

// if (count == 0){

// return true;

// }

count = 0;

for (int i = s.length() - 1; i >= 0; i--){

if (s.charAt(i) == ')' || s.charAt(i) == '\*'){

count++;

}else{

count--;

}

if (count < 0){

return false;

}

}

return true;

}

}

Method 3: Recursion, slowest solution

class Solution {

public boolean checkValidString(String s) {

return check(s, 0, 0);

}

private boolean check(String s, int start, int count){

if (count < 0){

return false;

}

while (start < s.length()){

if (s.charAt(start) == '('){

count++;

start++;

}else if (s.charAt(start) == ')'){

count--;

start++;

}else{

break;

}

if (count < 0){

return false;

}

}

if (start == s.length()){

return count == 0;

}

if (check(s, start + 1, count - 1) || check(s, start + 1, count) || check(s, start + 1, count + 1)){

return true;

}

return false;

}

}

class Solution {

public boolean checkValidString(String s) {

return check(s, 0, 0);

}

private boolean check(String s, int start, int count){

if (count < 0){

return false;

}

for (int i = start; i < s.length(); i++){

char c = s.charAt(i);

if (c == '('){

count++;

}else if (c == ')'){

count--;

}else{

if (check(s, i + 1, count - 1) || check(s, i + 1, count) || check(s, i + 1, count + 1)){

return true;

}else{

return false;

}

}

if (count < 0){

return false;

}

}

return count == 0;

}

}

# 679\_Game.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have 4 cards each containing a number from 1 to 9. You need to judge whether they could operated through \*, /, +, -, (, ) to get the value of 24.

Example 1:

Input: [4, 1, 8, 7]

Output: True

Explanation: (8-4) \* (7-1) = 24

Example 2:

Input: [1, 2, 1, 2]

Output: False

Note:

The division operator / represents real division, not integer division. For example, 4 / (1 - 2/3) = 12.

Every operation done is between two numbers. In particular, we cannot use - as a unary operator. For example, with [1, 1, 1, 1] as input, the expression -1 - 1 - 1 - 1 is not allowed.

You cannot concatenate numbers together. For example, if the input is [1, 2, 1, 2], we cannot write this as 12 + 12.

Backtracking within backtracking

https://leetcode.com/problems/24-game/discuss/107673/JAVA-Easy-to-understand.-Backtracking.

class Solution {

private final double eps = 0.000001;

public boolean judgePoint24(int[] nums) {

List<Double> list = new ArrayList<>();

for (int num : nums){

list.add((double)num);

}

return judgePoint24(list);

}

private boolean judgePoint24(List<Double> list){

if (list.size() == 1){

return Math.abs(list.get(0) - 24.0) < eps;

}

for (int i = 0; i < list.size() - 1; i++){

for (int j = i + 1; j < list.size(); j++){

double op1 = list.get(i);

double op2 = list.get(j);

List<Double> calList = new ArrayList<>();

calList.addAll(Arrays.asList(op1+op2, op1-op2, op2-op1, op1\*op2));

if (Math.abs(op1) > eps){

calList.add(op2/op1);//it is double division/real division

}

if (Math.abs(op2) > eps){

calList.add(op1/op2);

}

list.remove(j);//can't remove i first, must remove j first

list.remove(i);

for (double d : calList){

list.add(d);

if (judgePoint24(list)){

return true;

}

list.remove(list.size() - 1);

}

list.add(i, op1);//can't add j first, must add i first

list.add(j, op2);

}

}

return false;

}

}

# 68\_Text.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly maxWidth characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

Note:

A word is defined as a character sequence consisting of non-space characters only.

Each word's length is guaranteed to be greater than 0 and not exceed maxWidth.

The input array words contains at least one word.

Example 1:

Input:

words = ["This", "is", "an", "example", "of", "text", "justification."]

maxWidth = 16

Output:

[

"This is an",

"example of text",

"justification. "

]

Example 2:

Input:

words = ["What","must","be","acknowledgment","shall","be"]

maxWidth = 16

Output:

[

"What must be",

"acknowledgment ",

"shall be "

]

Explanation: Note that the last line is "shall be " instead of "shall be",

because the last line must be left-justified instead of fully-justified.

Note that the second line is also left-justified becase it contains only one word.

Example 3:

Input:

words = ["Science","is","what","we","understand","well","enough","to","explain",

"to","a","computer.","Art","is","everything","else","we","do"]

maxWidth = 20

Output:

[

"Science is what we",

"understand well",

"enough to explain to",

"a computer. Art is",

"everything else we",

"do "

]

class Solution {

public List<String> fullJustify(String[] words, int maxWidth) {

List<String> res = new ArrayList<>();

int index = 0;

while (index < words.length){

//1.calculate how many words to fit in one line

int last = index + 1;

int count = words[index].length();

while (last < words.length){

if (count + 1 + words[last].length() > maxWidth){

break;

}

count += 1 + words[last].length();

last++;

}

//2. calculate how many spaces

StringBuilder sb = new StringBuilder();

sb.append(words[index]);

//check if it is the last line

int diff = last - 1 - index;

if (last == words.length || diff == 0){//left justify, when diff == 0, it means one word for one line

for (int i = index + 1; i < last; i++){

sb.append(" ");

sb.append(words[i]);

}

for (int i = sb.length(); i < maxWidth; i++){

sb.append(" ");

}

}else{//center justify

int spaces = (maxWidth - count) / diff;

int remainder = (maxWidth - count) % diff;

for (int i = index + 1; i < last; i++){

for (int j = 0; j < spaces; j++){

sb.append(" ");//even distributed space

}

if (remainder > 0){//remainder for left adjust

sb.append(" ");

remainder--;

}

sb.append(" ");// default one " " space

sb.append(words[i]);

}

}

res.add(sb.toString());

index = last;

}

return res;

}

}

https://leetcode.com/problems/text-justification/discuss/24876/Simple-Java-Solution

//首先要做的就是确定每一行能放下的单词数，这个不难，就是比较n个单词的长度和加上n - 1个空格的长度跟给定的长度L来比较即可

//找到了一行能放下的单词个数，然后计算出这一行存在的空格的个数，是用给定的长度L减去这一行所有单词的长度和。

//得到了空格的个数之后，就要在每个单词后面插入这些空格，这里有两种情况，比如某一行有两个单词"to" 和 "a"，给定长度L为6

//如果这行不是最后一行，那么应该输出"to a"，如果是最后一行，则应该输出 "to a "，所以这里需要分情况讨论，最后一行的处理方法和其他行之间略有不同。

//最后一个难点就是，如果一行有三个单词，这时候中间有两个空，如果空格数不是2的倍数，那么左边的空间里要比右边的空间里多加入一个空格，那么我们只需要用总的空格数除以空间个数

//能除尽最好，说明能平均分配，除不尽的话就多加个空格放在左边的空间里"

public class Solution {

public List<String> fullJustify(String[] words, int maxWidth) {

List<String> lines = new ArrayList<String>();

int index = 0;

while (index < words.length) {

//count：该行所有单词累计总长度

int count = words[index].length();

//last:该行最后一个词的index

int last = index + 1;

while (last < words.length) {

//out of bound

if (words[last].length() + count + 1 > maxWidth) break;

//plus one for the space, if its a perfect fit it will fit

count += 1 + words[last].length();

last++;

}

StringBuilder builder = new StringBuilder();

//append该行第一个单词

builder.append(words[index]);

//这一行除去第一个已经append的单词，共剩下几个词语：diff 个：从index到last-1

int diff = last - index - 1;

// if last line or number of words in the line is 1, left-justified

//最后一行：每个单词中间一个空格， 剩余补上空白

if (last == words.length || diff == 0) {

for (int i = index+1; i < last; i++) {

builder.append(" ");

builder.append(words[i]);

}

for (int i = builder.length(); i < maxWidth; i++) {

builder.append(" ");

}

} else {

//不是最后一行：middle justified

//这一行总space的个数：（长度-累计单词总长度）

//每个单词后面space的个数：（长度-累计单词总长度）/单词个数

// r为需要平均分配到中间的空格总数

int spaces = (maxWidth - count) / diff;

int r = (maxWidth - count) % diff;

for (int i = index+1; i < last; i++) {

for(int k=spaces; k > 0; k--) {

builder.append(" ");

}

if(r > 0) {

builder.append(" ");

r--;

}

builder.append(" ");

builder.append(words[i]);

}

}

lines.add(builder.toString());

index = last;

}

return lines;

}

}

# 680\_Valid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty string s, you may delete at most one character. Judge whether you can make it a palindrome.

Example 1:

Input: "aba"

Output: True

Example 2:

Input: "abca"

Output: True

Explanation: You could delete the character 'c'.

Note:

The string will only contain lowercase characters a-z. The maximum length of the string is 50000.

class Solution {

public boolean validPalindrome(String s) {

int left = 0;

int right = s.length() - 1;

while (left < right){

if (s.charAt(left) != s.charAt(right)){

return isPalindrome(s, left+1, right) || isPalindrome(s, left, right-1);

}

left++;

right--;

}

return true;

}

private boolean isPalindrome(String s, int l, int r){

while (l < r){

if (s.charAt(l) != s.charAt(r)){

return false;

}

l++;

r--;

}

return true;

}

}

# 681\_Next.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a time represented in the format "HH:MM", form the next closest time by reusing the current digits. There is no limit on

how many times a digit can be reused.

You may assume the given input string is always valid. For example, "01:34", "12:09" are all valid. "1:34", "12:9" are all invalid.

Example 1:

Input: "19:34"

Output: "19:39"

Explanation: The next closest time choosing from digits 1, 9, 3, 4, is 19:39, which occurs 5 minutes later. It is not 19:33,

because this occurs 23 hours and 59 minutes later.

Example 2:

Input: "23:59"

Output: "22:22"

Explanation: The next closest time choosing from digits 2, 3, 5, 9, is 22:22. It may be assumed that the returned time is n

ext day's time since it is smaller than the input time numerically.

Similar to Leetcode 949 Largest Time for given digits

Method 1: backtrack

class Solution {

String result = "";

int diff = Integer.MAX\_VALUE;

public String nextClosestTime(String time) {

Set<Integer> set = new HashSet<>();

set.add(Integer.parseInt(time.substring(0, 1)));

set.add(Integer.parseInt(time.substring(1, 2)));

set.add(Integer.parseInt(time.substring(3, 4)));

set.add(Integer.parseInt(time.substring(4)));

if (set.size() == 1){

return time;

}

List<Integer> list = new ArrayList<Integer>(set);

int target = Integer.parseInt(time.substring(0, 2)) \* 60 + Integer.parseInt(time.substring(3));

dfs(list, "", 0, target);

return result;

}

private void dfs(List<Integer> list, String cur, int pos, int target){

if (pos == 4){

int minute = Integer.parseInt(cur.substring(0, 2)) \* 60 + Integer.parseInt(cur.substring(2));

if (minute == target){

return;

}

int d = minute - target > 0 ? minute - target : minute + 24 \* 60 - target;

if (d < diff){

diff = d;

result = cur.substring(0, 2) + ":" + cur.substring(2);

}

return;

}

for (int i = 0; i < list.size(); i++){

if (pos == 0 && list.get(i) > 2){

continue;

}

if (pos == 1 && Integer.parseInt(cur) \* 10 + list.get(i) > 23){

continue;

}

if (pos == 2 && list.get(i) > 5){

continue;

}

if (pos == 3 && Integer.parseInt(cur.substring(2)) \* 10 + list.get(i) > 59){

continue;

}

dfs(list, cur + list.get(i), pos + 1, target);

}

}

}

## 682\_BaseballGame.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You're now a baseball game point recorder.

Given a list of strings, each string can be one of the 4 following types:

Integer (one round's score): Directly represents the number of points you get in this round.

"+" (one round's score): Represents that the points you get in this round are the sum of the last two valid round's points.

"D" (one round's score): Represents that the points you get in this round are the doubled data of the last valid round's points.

"C" (an operation, which isn't a round's score): Represents the last valid round's points you get were invalid and should be removed.

Each round's operation is permanent and could have an impact on the round before and the round after.

You need to return the sum of the points you could get in all the rounds.

Example 1:

Input: ["5","2","C","D","+"]

Output: 30

Explanation:

Round 1: You could get 5 points. The sum is: 5.

Round 2: You could get 2 points. The sum is: 7.

Operation 1: The round 2's data was invalid. The sum is: 5.

Round 3: You could get 10 points (the round 2's data has been removed). The sum is: 15.

Round 4: You could get 5 + 10 = 15 points. The sum is: 30.

Example 2:

Input: ["5","-2","4","C","D","9","+","+"]

Output: 27

Explanation:

Round 1: You could get 5 points. The sum is: 5.

Round 2: You could get -2 points. The sum is: 3.

Round 3: You could get 4 points. The sum is: 7.

Operation 1: The round 3's data is invalid. The sum is: 3.

Round 4: You could get -4 points (the round 3's data has been removed). The sum is: -1.

Round 5: You could get 9 points. The sum is: 8.

Round 6: You could get -4 + 9 = 5 points. The sum is 13.

Round 7: You could get 9 + 5 = 14 points. The sum is 27.

Note:

The size of the input list will be between 1 and 1000.

Every integer represented in the list will be between -30000 and 30000.

class Solution {

public int calPoints(String[] ops) {

int sum = 0;

int[] nums = new int[ops.length];

int vp = -1;

for (int i = 0; i < ops.length; i++){

switch (ops[i]){

case "C":

sum -= nums[vp];

vp--;

break;

case "D":

int d = nums[vp] \* 2;

vp++;

nums[vp] = d;

sum += d;

break;

case "+":

int p = nums[vp] + nums[vp - 1];

vp++;

nums[vp] = p;

sum += p;

break;

default:

int n = Integer.parseInt(ops[i]);

vp++;

nums[vp] = n;

sum += n;

break;

}

}

return sum;

}

}

Better version:

class Solution {

public int calPoints(String[] ops) {

Stack<Integer> stack = new Stack<>();

for (String op : ops){

if (op.equals("C")){

stack.pop();

}else if (op.equals("D")){

stack.push(2 \* stack.peek());

}else if (op.equals("+")){

int temp = stack.pop();

int val = stack.peek() + temp;

stack.push(temp);

stack.push(val);

}else{

stack.push(Integer.parseInt(op));

}

}

int sum = 0;

for (int i : stack){

sum += i;

}

return sum;

}

}

## 683\_Empty.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a garden with N slots. In each slot, there is a flower. The N flowers will bloom one by one in N days. In each day,

there will be exactly one flower blooming and it will be in the status of blooming since then.

Given an array flowers consists of number from 1 to N. Each number in the array represents the place where the flower will open

in that day.

For example, flowers[i] = x means that the unique flower that blooms at day i will be at position x, where i and x will be in the

range from 1 to N.

Also given an integer k, you need to output in which day there exists two flowers in the status of blooming, and also the number

of flowers between them is k and these flowers are not blooming.

If there isn't such day, output -1.

Example 1:

Input:

flowers: [1,3,2]

k: 1

Output: 2

Explanation: In the second day, the first and the third flower have become blooming.

Example 2:

Input:

flowers: [1,2,3]

k: 1

Output: -1

Note:

The given array will be in the range [1, 20000].

It seems that this question has some mistakes. I think there are two places that might lead to misunderstandings: (please feel free to

tell me if I'm incorrect)

flowers[i] = x should mean that the unique flower that blooms at day i+1 (not i) will be at position x.

If you can get multiple possible results, then you need to return the minimum one.

The idea is to use an array days[] to record each position's flower's blooming day. That means days[i] is the blooming day of the

flower in position i+1. We just need to find a subarray days[left, left+1,..., left+k-1, right] which satisfies: for any

i = left+1,..., left+k-1, we can have days[left] < days[i] && days[right] < days[i]. Then, the result is max(days[left], days[right]).

There are two cases that we need to update left and right:

1. i arrives the position right (it means that we get a vliad subarray);

2. we find days[i] < days[left] || days[i] < days[right] (it means that the pre-assumed subarray is not correct).

Key : Maintain a size of k sliding window with the max day of left (outside of window) and right (outside of window) is greater than

the min of the days within the window. Find the minimum of the max day.

Sliding window

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int kEmptySlots(int[] flowers, int k) {

int N = flowers.length;

int[] days = new int[N];

for (int i = 0; i < N; i++){

days[flowers[i]-1] = i+1;

}

int left = 0;

int right = k+1;

int res = Integer.MAX\_VALUE;

for (int i = 0; right < N; i++){

if (days[i] < days[left] || days[i] <= days[right]){

if (i == right){

res = Math.min(res, Math.max(days[left], days[right]));

}

left = i;

right = k+1+i;

}

}

return res == Integer.MAX\_VALUE ? -1 : res;

}

}

## 684\_Redundant.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In this problem, a tree is an undirected graph that is connected and has no cycles.

The given input is a graph that started as a tree with N nodes (with distinct values 1, 2, ..., N), with one additional edge added. The added edge has two different vertices chosen from 1 to N, and was not an edge that already existed.

The resulting graph is given as a 2D-array of edges. Each element of edges is a pair [u, v] with u < v, that represents an undirected edge connecting nodes u and v.

Return an edge that can be removed so that the resulting graph is a tree of N nodes. If there are multiple answers, return the answer that occurs last in the given 2D-array. The answer edge [u, v] should be in the same format, with u < v.

Example 1:

Input: [[1,2], [1,3], [2,3]]

Output: [2,3]

Explanation: The given undirected graph will be like this:

1

/ \

2 - 3

Example 2:

Input: [[1,2], [2,3], [3,4], [1,4], [1,5]]

Output: [1,4]

Explanation: The given undirected graph will be like this:

5 - 1 - 2

| |

4 - 3

Note:

The size of the input 2D-array will be between 3 and 1000.

Every integer represented in the 2D-array will be between 1 and N, where N is the size of the input array.

Method 1: UF

Time complexity: O(N)

Space complexity: O(N)

class Solution {

class UF {

int[] parent;

public UF (int N){

parent = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

}

}

public int find(int x){

if (parent[x] == x){

return x;

}

return parent[x] = find(parent[x]); // path compression

}

public void union(int u, int v){

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV){

parent[rootU] = rootV;

}

}

}

public int[] findRedundantConnection(int[][] edges) {

int[] res = new int[2];

UF uf = new UF(1001);

for (int[] edge : edges){

int u = edge[0];

int v = edge[1];

if (uf.find(u) != uf.find(v)){

uf.union(u, v);

}else{

res[0] = u;

res[1] = v;

return res;

}

}

return res;

}

}

Method 2: DFS

Time complexity: O(N^2)

Space complexity: O(N)

For each edge (u, v), traverse the graph with a depth-first search to see if we can connect u to v.

If we can, then it must be the duplicate edge.

class Solution {

public int[] findRedundantConnection(int[][] edges) {

int[] res = new int[2];

Map<Integer, Set<Integer>> graph = new HashMap<>();

for (int[] edge : edges){

int u = edge[0];

int v = edge[1];

if (!graph.containsKey(u)){

graph.put(u, new HashSet<>());

}

if (!graph.containsKey(v)){

graph.put(v, new HashSet<>());

}

Set<Integer> seen = new HashSet<>();

if (!graph.get(u).isEmpty() && !graph.get(v).isEmpty() && dfs(graph, u, v, seen)){

//note that here we check isEmpty() to ensure it is added in the graph

// if not yet added in the graph, no need to check dfs; if added, check if u and v are connected

res[0] = u;

res[1] = v;

return res;

}

graph.get(u).add(v);

graph.get(v).add(u);

}

return res;

}

private boolean dfs(Map<Integer, Set<Integer>> graph, int u, int v, Set<Integer> seen){

if (!seen.contains(u)){

seen.add(u);

if (u == v){

return true;

}

for (int i : graph.get(u)){

if (dfs(graph, i, v, seen)){

return true;

}

}

}

return false;

}

}

class Solution {

public int[] findRedundantConnection(int[][] edges) {

Map<Integer, Set<Integer>> graph = new HashMap<>();

int[] res = new int[2];

for (int[] edge : edges){

int u = edge[0];

int v = edge[1];

if (!graph.containsKey(u)){

graph.put(u, new HashSet<>());

}

if (!graph.containsKey(v)){

graph.put(v, new HashSet<>());

}

Set<Integer> set = new HashSet<>();

if (!graph.get(u).isEmpty() && !graph.get(v).isEmpty()){

if (hasCircle(graph, u, v, set)){

res[0] = u;

res[1] = v;

return res;

}

}

graph.get(u).add(v);

graph.get(v).add(u);

}

return res;

}

private boolean hasCircle(Map<Integer, Set<Integer>> graph, int u, int v, Set<Integer> set){

if (u == v){

return true;

}

set.add(u);

for (int nei: graph.get(u)){

if (set.contains(nei)){

continue;

}

if (hasCircle(graph, nei, v, set)){

return true;

}

}

set.remove(u);

return false;

}

}

## 685\_Redundant.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In this problem, a rooted tree is a directed graph such that, there is exactly one node (the root) for which all other nodes are descendants of this node, plus every node has exactly one parent, except for the root node which has no parents.

The given input is a directed graph that started as a rooted tree with N nodes (with distinct values 1, 2, ..., N), with one additional directed edge added. The added edge has two different vertices chosen from 1 to N, and was not an edge that already existed.

The resulting graph is given as a 2D-array of edges. Each element of edges is a pair [u, v] that represents a directed edge connecting nodes u and v, where u is a parent of child v.

Return an edge that can be removed so that the resulting graph is a rooted tree of N nodes. If there are multiple answers, return the answer that occurs last in the given 2D-array.

Example 1:

Input: [[1,2], [1,3], [2,3]]

Output: [2,3]

Explanation: The given directed graph will be like this:

1

/ \

v v

2-->3

Example 2:

Input: [[1,2], [2,3], [3,4], [4,1], [1,5]]

Output: [4,1]

Explanation: The given directed graph will be like this:

5 <- 1 -> 2

^ |

| v

4 <- 3

Note:

The size of the input 2D-array will be between 3 and 1000.

Every integer represented in the 2D-array will be between 1 and N, where N is the size of the input array.

class Solution {

class UF {

int[] parent;

public UF (int N){

parent = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

}

}

public int find (int x){

if (parent[x] == x){

return x;

}

return find(parent[x]);

}

public void union (int u, int v){//set the root of u to be the root of v

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV){

parent[rootV] = rootU;

}

}

}

public int[] findRedundantDirectedConnection(int[][] edges) {

int[] cand1 = {-1, -1};

int[] cand2 = {-1, -1};

Map<Integer, Integer> map = new HashMap<>();

for (int[] edge : edges){

if (!map.containsKey(edge[1])){

map.put(edge[1], edge[0]);

}else{

cand1[0] = map.get(edge[1]);

cand1[1] = edge[1];

cand2[0] = edge[0];

cand2[1] = edge[1];

}

}

//union find to check which cand is the answer.

//There are 3 cases

if (cand1[0] == -1 && cand1[1] == -1){//Case 1: no node has two parents, but the graph has a circle

return findCircleEdge(edges, new int[]{0, 0});

}

int[] res = findCircleEdge(edges, cand2);//return the answer that occurs last

if (res[0] == 0 && res[1] == 0){//Case 2: remove cand2, tree is valid, so cand2 is the answer

return cand2;

}

return cand1;//Case 3: cand1 is the answer

}

private int[] findCircleEdge(int[][] edges, int[] edgeToRemove){

int[] res = new int[2];

UF uf = new UF(1001);

for (int[] edge: edges){

int u = edge[0];

int v = edge[1];

if (u == edgeToRemove[0] && v == edgeToRemove[1]){

continue;

}

if (uf.find(u) != uf.find(v)){

uf.union(u, v);

}else{

res[0] = u;

res[1] = v;

return res;

}

}

return res;

}

}

## 686\_RepeatedStringMatch.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings A and B, find the minimum number of times A has to be repeated such that B is a substring of it.

If no such solution, return -1.

For example, with A = "abcd" and B = "cdabcdab".

Return 3, because by repeating A three times (“abcdabcdabcd”), B is a substring of it; and B is not a substring of

A repeated two times ("abcdabcd").

Note:

The length of A and B will be between 1 and 10000.

If the string obtained as concatenating copies of string A has a length greater than the sum of

the lengths of A and B and still it does not have B as a substring, then any further A-concatenation is useless.

class Solution {

public int repeatedStringMatch(String A, String B) {

StringBuilder sb = new StringBuilder();

sb.append(A);

int count = 1;

while (sb.toString().indexOf(B) < 0){

if (sb.length() >= A.length() + B.length()){

return -1;

}

sb.append(A);

count++;

}

return count;

}

}

class Solution {

public int repeatedStringMatch(String A, String B) {

int len1 = A.length();

int len2 = B.length();

int n = len2 / len1;

if (len2 % len1 != 0){

n++;

}

StringBuilder sb = new StringBuilder();

for (int i = 0; i < n; i++){

sb.append(A);

}

String str = sb.toString();

int index = str.indexOf(B);

if (index >= 0){

return n;

}

str += A;

n++;

if (str.indexOf(B) >= 0){

return n;

}

return -1;

}

}

## 687\_Longest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, find the length of the longest path where each node in the path has the same value. This path may or may not pass through the root.

Note: The length of path between two nodes is represented by the number of edges between them.

Example 1:

Input:

5

/ \

4 5

/ \ \

1 1 5

Output:

2

Example 2:

Input:

1

/ \

4 5

/ \ \

4 4 5

Output:

2

Note: The given binary tree has not more than 10000 nodes. The height of the tree is not more than 1000.

Method:

https://leetcode.com/problems/longest-univalue-path/solution/

Let arrow\_length(node) be the length of the longest arrow that extends from the node.

That will be 1 + arrow\_length(node.left) if node.left exists and has the same value as node. Similarly for the node.right case.

While we are computing arrow lengths, each candidate answer will be the sum of the arrows in both directions from that node.

We record these candidate answers and return the best one.

Note that

For each node, there are 5 cases for the longest path as candidates

1. not pass node but on the left subtree

2. not pass node but on the right subtree

3. pass node and on the left subtree

4. pass node and on the right subtree

5. pass node and cross both left subtree and right subtree

We can use variable ans to track the longest path and cover cases 1, 2, 5; use recursion function return to cover cases 3, 4

Note that we are looking for path and graph, which means there are start node and end node. So in the function return, we have to

use Math.max(leftMax, rightMax)

Similar to 124. Binary Tree Maximum Path Sum

class Solution {

int max = Integer.MIN\_VALUE;

public int longestUnivaluePath(TreeNode root) {

if (root == null){

return 0;

}

longestIncludeRoot(root); //return the longest path that pass root, note that this is not looking for graph

return max;

}

private int longestIncludeRoot(TreeNode root){

if (root == null){

return 0;

}

int left = longestIncludeRoot(root.left);

int right = longestIncludeRoot(root.right);

int leftIncludeRoot = 0; //number of longest path that pass through root

int rightIncludeRoot = 0;

if (root.left != null && root.val == root.left.val){

leftIncludeRoot = left + 1;

}

if (root.right != null && root.val == root.right.val){

rightIncludeRoot = right + 1;

}

max = Math.max(max, leftIncludeRoot + rightIncludeRoot);

return Math.max(leftIncludeRoot, rightIncludeRoot);

}

}

Time Complexity: O(N), where NN is the number of nodes in the tree. We process every node once.

Space Complexity: O(logN or H), where H is the height of the tree. Our recursive call stack could be up to H layers deep.

Remove global variable

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int longestUnivaluePath(TreeNode root) {

int[] max = new int[1];

longestIncludeRoot(root, max);

return max[0];

}

private int longestIncludeRoot(TreeNode root, int[] max){

if (root == null){

return 0;

}

int left = longestIncludeRoot(root.left, max);

int right = longestIncludeRoot(root.right, max);

int leftIncludeRoot = 0;

int rightIncludeRoot = 0;

if (root.left != null && root.val == root.left.val){

leftIncludeRoot = left + 1;

}

if (root.right != null && root.val == root.right.val){

rightIncludeRoot = right + 1;

}

max[0] = Math.max(max[0], leftIncludeRoot + rightIncludeRoot);

return Math.max(leftIncludeRoot, rightIncludeRoot);

}

}

## 688\_KnightProbabilityinChessboard.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On an NxN chessboard, a knight starts at the r-th row and c-th column and attempts to make exactly K moves. T

he rows and columns are 0 indexed, so the top-left square is (0, 0), and the bottom-right square is (N-1, N-1).

A chess knight has 8 possible moves it can make, as illustrated below. Each move is two squares in a cardinal direction,

then one square in an orthogonal direction.

Each time the knight is to move, it chooses one of eight possible moves uniformly at random (even if the piece

would go off the chessboard) and moves there.

The knight continues moving until it has made exactly K moves or has moved off the chessboard.

Return the probability that the knight remains on the board after it has stopped moving.

Example:

Input: 3, 2, 0, 0

Output: 0.0625

Explanation: There are two moves (to (1,2), (2,1)) that will keep the knight on the board.

From each of those positions, there are also two moves that will keep the knight on the board.

The total probability the knight stays on the board is 0.0625.

Note:

N will be between 1 and 25.

K will be between 0 and 100.

The knight always initially starts on the board.

Method: BFS will lead to TLE, so use DP

https://www.youtube.com/watch?v=MyJvMydR2G4

dp[i][j] is the number of ways to move to [i][j]

class Solution {

public double knightProbability(int N, int K, int r, int c) {

double[][] dpPrev = new double[N][N];

dpPrev[r][c] = 1.0;

int[] dx = {2, 1, -1, -2, -2, -1, 1, 2};

int[] dy = {1, 2, 2, 1, -1, -2, -2, -1};

for (int k = 0; k < K; k++){

double[][] dpCur = new double[N][N];

for (int i = 0; i < N; i++){

for (int j = 0; j < N; j++){

for (int l = 0; l < dx.length; l++){

int nx = i + dx[l];

int ny = j + dy[l];

if (inBound(N, nx, ny)){

dpCur[i][j] += dpPrev[nx][ny];

}

}

}

}

dpPrev = dpCur;

}

double sum = 0.0;

for (int i = 0; i < N; i++){

for (int j = 0; j < N; j++){

sum += dpPrev[i][j];

}

}

return sum / Math.pow(8, K);

}

private boolean inBound(int N, int nx, int ny){

return nx >= 0 && nx < N && ny >=0 && ny < N;

}

}

class Solution {

public double knightProbability(int N, int K, int r, int c) {

double[][] dpPrev = new double[N][N];

dpPrev[r][c] = 1.0;

int[] dx = {2, 1, -1, -2, -2, -1, 1, 2};

int[] dy = {1, 2, 2, 1, -1, -2, -2, -1};

for (int k = 0; k < K; k++){

double[][] dpCur = new double[N][N];

for (int i = 0; i < N; i++){

for (int j = 0; j < N; j++){

for (int l = 0; l < dx.length; l++){

int nx = i - dx[l];

int ny = j - dy[l];

if (nx >= 0 && nx < N && ny >= 0 && ny < N){

dpCur[i][j] += dpPrev[nx][ny];

}

}

}

}

dpPrev = dpCur;

}

double sum = 0.0;

for (int i = 0; i < N; i++){

for (int j = 0; j < N; j++){

sum += dpPrev[i][j];

}

}

return sum / Math.pow(8, K);

}

}

Similar as Leetcode 576

https://github.com/optimisea/Leetcode/blob/master/Java/576\_Out.java

## 689\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a given array nums of positive integers, find three non-overlapping subarrays with maximum sum.

Each subarray will be of size k, and we want to maximize the sum of all 3\*k entries.

Return the result as a list of indices representing the starting position of each interval (0-indexed). If there are multiple answers, return the lexicographically smallest one.

Example:

Input: [1,2,1,2,6,7,5,1], 2

Output: [0, 3, 5]

Explanation: Subarrays [1, 2], [2, 6], [7, 5] correspond to the starting indices [0, 3, 5].

We could have also taken [2, 1], but an answer of [1, 3, 5] would be lexicographically larger.

Note:

nums.length will be between 1 and 20000.

nums[i] will be between 1 and 65535.

k will be between 1 and floor(nums.length / 3).

https://leetcode.com/problems/maximum-sum-of-3-non-overlapping-subarrays/discuss/108230/Clean-Java-DP-O(n)-Solution.-Easy-extend-to-Sum-of-K-Non-Overlapping-SubArrays.

Method: best, DP

dp[i][j] deontes max total sum at the i-th part and ending at index j - 1

id[i][j] deontes starting index + 1 at the i-th part and ending at index j - 1

class Solution {

public int[] maxSumOfThreeSubarrays(int[] nums, int k) {

int m = 3;

int n = nums.length;

int[][] dp = new int[m+1][n+1];

int[][] idx = new int[m+1][n+1];

int[] preSum = new int[n+1];

for (int i = 1; i <= n; i++){

preSum[i] = preSum[i-1] + nums[i-1];

}

for (int i = 1; i <= m; i++){

for (int j = i \* k; j <= n - (m-i)\*k; j++){

//initialization for comparision

dp[i][j] = dp[i][j-1];

idx[i][j] = idx[i][j-1];

//find the largest sums and starting index

int cand = dp[i-1][j-k] + preSum[j] - preSum[j-k];

if (cand > dp[i][j]){

dp[i][j] = cand;

idx[i][j] = j - k + 1;

}

}

}

//generalized format

int[] res = new int[m];

res[m-1] = idx[m][n] - 1;

for (int i = m-2; i >= 0; i--){

res[i] = idx[i+1][res[i+1]] - 1;

}

return res;

// int last = idx[m][n];

// int mid = idx[m-1][last-1];

// int first = idx[m-2][mid-1];

// return new int[]{first-1, mid-1, last-1};

}

}

class Solution {

public int[] maxSumOfThreeSubarrays(int[] nums, int k) {

int m = 3;

int n = nums.length;

int[][] dp = new int[m+1][n+1];

int[][] idx = new int[m+1][n+1];

int[] preSum = new int[n+1];

for (int i = 1; i <= n; i++){

preSum[i] = preSum[i-1] + nums[i-1];

}

for (int i = 1; i <= m; i++){

for (int j = i \* k; j <= n - (m-i)\*k; j++){

//initialization for comparision

dp[i][j] = dp[i][j-1];

idx[i][j] = idx[i][j-1];

//find the largest sums and starting index

int cand = dp[i-1][j-k] + preSum[j] - preSum[j-k];

if (cand > dp[i][j]){

dp[i][j] = cand;

idx[i][j] = j - k + 1;

}

}

}

int[] res = new int[m];

res[m-1] = idx[m][n] - 1;

for (int i = m-2; i >= 0; i--){

res[i] = idx[i+1][res[i+1]] - 1;

}

return res;

// int last = idx[m][n];

// int mid = idx[m-1][last-1];

// int first = idx[m-2][mid-1];

// return new int[]{first-1, mid-1, last-1};

}

}

Method:

class Solution {

public int[] maxSumOfThreeSubarrays(int[] nums, int k) {

//prefix sum to construct W array with each elemenet representing the sum of k in nums

int[] W = new int[nums.length - k + 1];

int sum = 0;

for (int i = 0; i < nums.length; i++){

sum += nums[i];

if (i >= k){

sum -= nums[i - k];

}

if (i >= k - 1){

W[i - k + 1] = sum;

}

}

//build left array with each element representing the first occurance of largest element in W from left

int[] left = new int[W.length];

int best = 0;

for (int i = 0; i < W.length; i++){

if (W[i] > W[best]){//Note not equal

best = i;

}

left[i] = best;

}

//build right array with each element representing the last occurance of the lraget element in W from right

int[] right = new int[W.length];

best = W.length - 1;

for (int i = W.length - 1; i >= 0; i--){

if (W[i] >= W[best]){//Note must use equal

best = i;

}

right[i] = best;

}

//find the largest sum with the mid subarray j as the pointer

int[] res = new int[]{-1, -1, -1};

for (int j = k; j < W.length - k; j++){

int i = left[j-k];

int m = right[j+k];

if (res[0] == -1 || W[res[0]] + W[res[1]] + W[res[2]] < W[i] + W[j] + W[m]){

res[0] = i;

res[1] = j;

res[2] = m;

}

}

return res;

}

}

Intuition

It is natural to consider an array W of each interval's sum, where each interval is the given length K.

To create W, we can either use prefix sums, or manage the sum of the interval as a window slides along the array.

From there, we approach the reduced problem: Given some array W and an integer K, what is the lexicographically

smallest tuple of indices (i, j, k) with i + K <= j and j + K <= k that maximizes W[i] + W[j] + W[k]?

Algorithm

Suppose we fixed j. We would like to know on the intervals i∈[0,j−K]i \in [0, j-K]i∈[0,j−K] and k∈[j+K,len(W)−1]k \in [j+K,

\text{len}(W)-1]k∈[j+K,len(W)−1], where the largest value of W[i]W[i]W[i] (and respectively W[k]W[k]W[k]) occurs first.

(Here, first means the smaller index.)

We can solve these problems with dynamic programming. For example, if we know that iii is where the largest value of W[i]W[i]W[i]

occurs first on [0,5][0, 5][0,5], then on [0,6][0, 6][0,6] the first occurrence of the largest W[i]W[i]W[i] must be either iii or 666.

If say, 666 is better, then we set best = 6.

At the end, left[z] will be the first occurrence of the largest value of W[i] on the interval i∈[0,z]i \in [0, z]i∈[0,z],

and right[z] will be the same but on the interval i∈[z,len(W)−1]i \in [z, \text{len}(W) - 1]i∈[z,len(W)−1]. This means that for some choice j, the candidate answer must be (left[j-K], j, right[j+K]). We take the candidate that produces the maximum W[i] + W[j] + W[k].

## 690\_Employee.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given a data structure of employee information, which includes the employee's unique id, his importance value and his direct subordinates' id.

For example, employee 1 is the leader of employee 2, and employee 2 is the leader of employee 3. They have importance value 15, 10 and 5, respectively. Then employee 1 has a data structure like [1, 15, [2]], and employee 2 has [2, 10, [3]], and employee 3 has [3, 5, []]. Note that although employee 3 is also a subordinate of employee 1, the relationship is not direct.

Now given the employee information of a company, and an employee id, you need to return the total importance value of this employee and all his subordinates.

Example 1:

Input: [[1, 5, [2, 3]], [2, 3, []], [3, 3, []]], 1

Output: 11

Explanation:

Employee 1 has importance value 5, and he has two direct subordinates: employee 2 and employee 3. They both have importance value 3. So the total importance value of employee 1 is 5 + 3 + 3 = 11.

Note:

One employee has at most one direct leader and may have several subordinates.

The maximum number of employees won't exceed 2000.

Method 1: BFS

/\*

// Employee info

class Employee {

// It's the unique id of each node;

// unique id of this employee

public int id;

// the importance value of this employee

public int importance;

// the id of direct subordinates

public List<Integer> subordinates;

};

\*/

class Solution {

public int getImportance(List<Employee> employees, int id) {

Map<Integer, Employee> map = new HashMap<>();

for (Employee employee : employees){

map.put(employee.id, employee);

}

Queue<Integer> queue = new LinkedList<>();

queue.offer(id);

int sum = 0;

while (!queue.isEmpty()){

int empId = queue.poll();

Employee emp = map.get(empId);

sum += emp.importance;

for (int e : emp.subordinates){

queue.offer(e);

}

}

return sum;

}

}

Method 2: DFS

/\*

// Employee info

class Employee {

// It's the unique id of each node;

// unique id of this employee

public int id;

// the importance value of this employee

public int importance;

// the id of direct subordinates

public List<Integer> subordinates;

};

\*/

class Solution {

int sum = 0;

public int getImportance(List<Employee> employees, int id) {

Map<Integer, Employee> map = new HashMap<>();

for (Employee employee : employees){

map.put(employee.id, employee);

}

dfs(map, id);

return sum;

}

private void dfs(Map<Integer, Employee> map, int id){

Employee emp = map.get(id);

sum += emp.importance;

for (int e : emp.subordinates){

dfs(map, e);

}

}

}

class Solution {

public int getImportance(List<Employee> employees, int id) {

Map<Integer, Employee> map = new HashMap<>();

for (Employee employee : employees){

map.put(employee.id, employee);

}

return dfs(map, id);

}

private int dfs(Map<Integer, Employee> map, int id){

Employee emp = map.get(id);

int sum = emp.importance;

for (int e : emp.subordinates){

sum += dfs(map, e);

}

return sum;

}

}

## 691\_Stickers.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given N different types of stickers. Each sticker has a lowercase English word on it.

You would like to spell out the given target string by cutting individual letters from your collection of stickers and rearranging them.

You can use each sticker more than once if you want, and you have infinite quantities of each sticker.

What is the minimum number of stickers that you need to spell out the target? If the task is impossible, return -1.

Example 1:

Input:

["with", "example", "science"], "thehat"

Output:

3

Explanation:

We can use 2 "with" stickers, and 1 "example" sticker.

After cutting and rearrange the letters of those stickers, we can form the target "thehat".

Also, this is the minimum number of stickers necessary to form the target string.

Example 2:

Input:

["notice", "possible"], "basicbasic"

Output:

-1

Explanation:

We can't form the target "basicbasic" from cutting letters from the given stickers.

Note:

stickers has length in the range [1, 50].

stickers consists of lowercase English words (without apostrophes).

target has length in the range [1, 15], and consists of lowercase English letters.

In all test cases, all words were chosen randomly from the 1000 most common US English words, and the target was chosen as a concatenation of two random words.

The time limit may be more challenging than usual. It is expected that a 50 sticker test case can be solved within 35ms on average.

Method: DP + backtracking

https://leetcode.com/problems/stickers-to-spell-word/discuss/108318/C++JavaPython-DP-+-Memoization-with-optimization-29-ms-(C++)

class Solution {

public int minStickers(String[] stickers, String target) {

int m = stickers.length;

int[][] stickerMap = new int[m][26];

for (int i = 0; i < m; i++){

String s = stickers[i];

for (int j = 0; j < s.length(); j++){

stickerMap[i][s.charAt(j) - 'a']++;

}

}

Map<String, Integer> dp = new HashMap<>();

dp.put("", 0); // one exit condition

return minStickers(stickerMap, target, dp);

}

private int minStickers(int[][] stickerMap, String target, Map<String, Integer> dp){

if (dp.containsKey(target)){

return dp.get(target);

}

int[] targetMap = new int[26];

for (int i = 0; i < target.length(); i++){

targetMap[target.charAt(i) - 'a']++;

}

int ans = Integer.MAX\_VALUE;

for (int i = 0; i < stickerMap.length; i++){

if (stickerMap[i][target.charAt(0) - 'a'] == 0){ //to speed up

continue;

}

StringBuilder sb = new StringBuilder();

for (int j = 0; j < 26; j++){

if (targetMap[j] > 0){

for (int k = 0; k < Math.max(0, targetMap[j] - stickerMap[i][j]); k++){

sb.append((char)(j + 'a'));

}

}

}

String str = sb.toString();

int min = minStickers(stickerMap, str, dp);

if (min != -1){

ans = Math.min(ans, min + 1);

}

}

if (ans == Integer.MAX\_VALUE){

ans = -1;

}

dp.put(target, ans);

return ans;

}

}

class Solution {

public int minStickers(String[] stickers, String target) {

Map<String, Map<Character, Integer>> stickerMap = new HashMap<>();

for (String str : stickers){

Map<Character, Integer> map = new HashMap<>();

for (char c : str.toCharArray()){

map.put(c, map.getOrDefault(c, 0) + 1);

}

stickerMap.put(str, map);

}

Map<String, Integer> memo = new HashMap<>();

memo.put("", 0);

return dfs(stickers, target, memo, stickerMap);

}

private int dfs(String[] stickers, String target, Map<String, Integer> memo, Map<String, Map<Character, Integer>> stickerMap){

if (memo.containsKey(target)){

return memo.get(target);

}

Map<Character, Integer> targetMap = new HashMap<>();

for (char c : target.toCharArray()){

targetMap.put(c, targetMap.getOrDefault(c, 0) + 1);

}

int min = Integer.MAX\_VALUE;

for (int i = 0; i < stickers.length; i++){

String str = stickers[i];

Map<Character, Integer> map = stickerMap.get(str);

if (!map.containsKey(target.charAt(0))){

continue;

}

StringBuilder sb = new StringBuilder();

for (char c : targetMap.keySet()){

int t = targetMap.get(c);

int s = map.getOrDefault(c, 0);

for (int j = 0; j < Math.max(0, t - s); j++){

sb.append(c);

}

}

String next = sb.toString();

int nextMin = dfs(stickers, next, memo, stickerMap);

if (nextMin != -1){

min = Math.min(min, nextMin + 1);

}

}

if (min == Integer.MAX\_VALUE){

min = - 1;

}

memo.put(target, min);

return min;

}

}

## 692\_TopKFrequentWords.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty list of words, return the k most frequent elements.

Your answer should be sorted by frequency from highest to lowest. If two words have the same frequency,

then the word with the lower alphabetical order comes first.

Example 1:

Input: ["i", "love", "leetcode", "i", "love", "coding"], k = 2

Output: ["i", "love"]

Explanation: "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

Example 2:

Input: ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4

Output: ["the", "is", "sunny", "day"]

Explanation: "the", "is", "sunny" and "day" are the four most frequent words,

with the number of occurrence being 4, 3, 2 and 1 respectively.

Note:

You may assume k is always valid, 1 ≤ k ≤ number of unique elements.

Input words contain only lowercase letters.

Follow up:

Try to solve it in O(n log k) time and O(n) extra space.

class Solution {

class Freq{

String str;

int freq;

public Freq(String str, int freq){

this.str = str;

this.freq = freq;

}

}

public List<String> topKFrequent(String[] words, int k) {

List<String> result = new ArrayList<>();

Map<String, Integer> map = new HashMap<>();

for (int i = 0; i < words.length; i++){

map.put(words[i], map.getOrDefault(words[i], 0) + 1);

}

Queue<Freq> maxHeap = new PriorityQueue<Freq>(words.length, new Comparator<Freq>(){

public int compare(Freq a, Freq b){

if (a.freq != b.freq){

return b.freq - a.freq;

}

return a.str.compareTo(b.str);

}

});

for (Map.Entry<String, Integer> entry : map.entrySet()){

maxHeap.offer(new Freq(entry.getKey(), entry.getValue()));

}

for(int i = 0 ; i < k; i++){

result.add(maxHeap.poll().str);

}

return result;

}

}

## 693\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer, check whether it has alternating bits: namely, if two adjacent bits will always have different values.

Example 1:

Input: 5

Output: True

Explanation:

The binary representation of 5 is: 101

Example 2:

Input: 7

Output: False

Explanation:

The binary representation of 7 is: 111.

Example 3:

Input: 11

Output: False

Explanation:

The binary representation of 11 is: 1011.

Example 4:

Input: 10

Output: True

Explanation:

The binary representation of 10 is: 1010.

Method 1:

class Solution {

public boolean hasAlternatingBits(int n) {

int prev = 2;

while (n != 0){

int curr = n % 2;

n /= 2;

if (prev != 2 && (curr ^ prev) == 0){

return false;

}

prev = curr;

}

return true;

}

}

class Solution {

public boolean hasAlternatingBits(int n) {

int cur = n % 2;

n /= 2;

while (n > 0) {

if (cur == n % 2) return false;

cur = n % 2;

n /= 2;

}

return true;

}

}

Method 2:

class Solution {

public boolean hasAlternatingBits(int n) {

n = (n >> 1) ^ n;

return (n & (n+1)) == 0;

}

}

class Solution {

public boolean hasAlternatingBits(int n) {

return ((n ^= n/2) & n+1) == 0;

}

}

Method 3:

class Solution {

public boolean hasAlternatingBits(int n) {

String bits = Integer.toBinaryString(n);

for (int i = 0; i < bits.length() - 1; i++){

if (bits.charAt(i) == bits.charAt(i+1)){

return false;

}

}

return true;

}

}

## 694\_NumberIslands.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally

(horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Count the number of distinct islands. An island is considered to be the same as another

if and only if one island can be translated (and not rotated or reflected) to equal the other.

Example 1:

11000

11000

00011

00011

Given the above grid map, return 1.

Example 2:

11011

10000

00001

11011

Given the above grid map, return 3.

Notice that:

11

1

and

1

11

are considered different island shapes, because we do not consider reflection / rotation.

Note: The length of each dimension in the given grid does not exceed 50.

class Solution {

public int numDistinctIslands(int[][] grid) {

if (grid == null || grid.length == 0 || grid[0].length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

int count = 0;

Set<String> set = new HashSet<>();

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1){

String str = bfs(grid, i, j);

if (!set.contains(str)){

count++;

set.add(str);

}

}

}

}

return count;

}

private String bfs(int[][] grid, int i, int j){

int m = grid.length;

int n = grid[0].length;

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

qx.offer(i);

qy.offer(j);

grid[i][j] = 0;

StringBuilder sb = new StringBuilder();

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

for (int k = 0; k < dx.length; k++){

int nx = cx + dx[k];

int ny = cy + dy[k];

if (0 <= nx && nx < m && 0 <= ny && ny < n && grid[nx][ny] == 1){

qx.offer(nx);

qy.offer(ny);

grid[nx][ny] = 0;

sb.append(dx[k]);

sb.append(dy[k]);

}

}

sb.append("\_");

}

return sb.toString();

}

}

Method 2: DFS

class Solution {

public int numDistinctIslands(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

Set<String> set = new HashSet<>();

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1){

StringBuilder sb = new StringBuilder();

dfs(grid, i, j, sb, "0");

set.add(sb.toString());

}

}

}

return set.size();

}

private String dfs(int[][] grid, int x, int y, StringBuilder sb, String str){

int m = grid.length;

int n = grid[0].length;

if (x >= 0 && x < m && y >= 0 && y < n && grid[x][y] == 1){

grid[x][y] = 0;

sb.append(str);

dfs(grid, x+1, y, sb, "1");

dfs(grid, x, y+1, sb, "2");

dfs(grid, x-1, y, sb, "3");

dfs(grid, x, y-1, sb, "4");

sb.append("\_");

}

return sb.toString();

}

}

## 695\_Max.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Find the maximum area of an island in the given 2D array. (If there is no island, the maximum area is 0.)

Example 1:

[[0,0,1,0,0,0,0,1,0,0,0,0,0],

[0,0,0,0,0,0,0,1,1,1,0,0,0],

[0,1,1,0,1,0,0,0,0,0,0,0,0],

[0,1,0,0,1,1,0,0,1,0,1,0,0],

[0,1,0,0,1,1,0,0,1,1,1,0,0],

[0,0,0,0,0,0,0,0,0,0,1,0,0],

[0,0,0,0,0,0,0,1,1,1,0,0,0],

[0,0,0,0,0,0,0,1,1,0,0,0,0]]

Given the above grid, return 6. Note the answer is not 11, because the island must be connected 4-directionally.

Example 2:

[[0,0,0,0,0,0,0,0]]

Given the above grid, return 0.

Note: The length of each dimension in the given grid does not exceed 50.

Method 1: BFS Similar as Number of Island

Time complexity: O(mn \* mn)

Space complexity: O(mn)

class Solution {

public int maxAreaOfIsland(int[][] grid) {

int max = 0;

int m = grid.length;

int n = grid[0].length;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1){

int count = bfs(grid, i, j);

max = Math.max(max, count);

}

}

}

return max;

}

private int bfs(int[][] grid, int i, int j){

int count = 0;

int m = grid.length;

int n = grid[0].length;

int[] dx = {0, 1, 0, -1};

int[] dy = {1, 0, -1, 0};

Queue<int[]> queue = new LinkedList<>();

queue.offer(new int[]{i, j});

grid[i][j] = 0;

while (!queue.isEmpty()){

int[] p = queue.poll();

int cx = p[0];

int cy = p[1];

count++;

for (int k = 0; k < dx.length; k++){

int x = cx + dx[k];

int y = cy + dy[k];

if (x >= 0 && x < m && y >= 0 && y < n && grid[x][y] == 1){

queue.offer(new int[]{x, y});

grid[x][y] = 0;

}

}

}

return count;

}

}

Without modifying grid

Time complexity: O(M\*N)

class Solution {

public int maxAreaOfIsland(int[][] grid) {

if (grid == null || grid.length == 0){

return 0;

}

int m = grid.length;

int n = grid[0].length;

int max = 0;

boolean[][] visited = new boolean[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1 && !visited[i][j]){

max = Math.max(max, bfs(grid, i, j, visited));

}

}

}

return max;

}

private int bfs(int[][] grid, int x, int y, boolean[][] visited){

int m = grid.length;

int n = grid[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

queue.offer(new int[]{x, y});

visited[x][y] = true;

int count = 1;

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] == 1 && !visited[nx][ny]){

queue.offer(new int[]{nx, ny});

visited[nx][ny] = true;

count++;

}

}

}

return count;

}

}

Method 2: DFS recursion

Time complexity: O(mn \* mn)

Space complexity: O(mn)

class Solution {

int[][] grid;

boolean[][] seen;

public int area(int r, int c) {

if (r < 0 || r >= grid.length || c < 0 || c >= grid[0].length ||

seen[r][c] || grid[r][c] == 0)

return 0;

seen[r][c] = true;

return (1 + area(r+1, c) + area(r-1, c)

+ area(r, c-1) + area(r, c+1));

}

public int maxAreaOfIsland(int[][] grid) {

this.grid = grid;

seen = new boolean[grid.length][grid[0].length];

int ans = 0;

for (int r = 0; r < grid.length; r++) {

for (int c = 0; c < grid[0].length; c++) {

ans = Math.max(ans, area(r, c));

}

}

return ans;

}

}

## 696\_Count.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Give a string s, count the number of non-empty (contiguous) substrings that have the same number of 0's and 1's, and all the 0's and

all the 1's in these substrings are grouped consecutively.

Substrings that occur multiple times are counted the number of times they occur.

Example 1:

Input: "00110011"

Output: 6

Explanation: There are 6 substrings that have equal number of consecutive 1's and 0's: "0011", "01", "1100", "10", "0011", and "01".

Notice that some of these substrings repeat and are counted the number of times they occur.

Also, "00110011" is not a valid substring because all the 0's (and 1's) are not grouped together.

Example 2:

Input: "10101"

Output: 4

Explanation: There are 4 substrings: "10", "01", "10", "01" that have equal number of consecutive 1's and 0's.

Note:

s.length will be between 1 and 50,000.

s will only consist of "0" or "1" characters.

Method 1: Brute Force

Time complexity: worst O(n^2) because it has backward

Space complexity: O(1)

class Solution {

public int countBinarySubstrings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int start = 0;

int end = 0;

int i = 0;

int ans = 0;

while (i < s.length()){

int startCount = 0;

int endCount = 0;

while (i < s.length() && s.charAt(i) == s.charAt(start)){

i++;

startCount++;

}

end = i;

while (i < s.length() && s.charAt(i) == s.charAt(end)){

i++;

endCount++;

if (endCount <= startCount){

ans++;

}

}

start = end;

i = end;

}

return ans;

}

}

Method 2: Group By Character

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int countBinarySubstrings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int[] group = new int[s.length()];

int t = 0;

group[0] = 1;

for (int i = 1; i < s.length(); i++){

if (s.charAt(i) == s.charAt(i-1)){

group[t]++;

}else{

group[++t] = 1;

}

}

int ans = 0;

for (int i = 1; i < group.length; i++){

ans += Math.min(group[i-1], group[i]);

}

return ans;

}

}

Method 3: Best solution, based on method 2, just remember to add the last item Math.min(prev, curr) because it is out of loop, the

calculate is not triggered.

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public int countBinarySubstrings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int ans = 0;

int prev = 0;

int curr = 1;

for (int i = 1; i < s.length(); i++){

if (s.charAt(i) == s.charAt(i-1)){

curr++;

}else{

ans += Math.min(prev, curr);

prev = curr;

curr = 1;

}

}

return ans + Math.min(prev, curr);

}

}

Method 4: Best of Best

class Solution {

public int countBinarySubstrings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int ans = 0;

int prev = 0;

int curr = 1;

for (int i = 1; i < s.length(); i++){

if (s.charAt(i) == s.charAt(i-1)){

curr++;

}else{

prev = curr;

curr = 1;

}

if (prev >= curr){

ans++;

}

}

return ans;

}

}

## 697\_DegreeofArray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty array of non-negative integers nums, the degree of this array is defined as the maximum frequency of any one of its elements.

Your task is to find the smallest possible length of a (contiguous) subarray of nums, that has the same degree as nums.

Example 1:

Input: [1, 2, 2, 3, 1]

Output: 2

Explanation:

The input array has a degree of 2 because both elements 1 and 2 appear twice.

Of the subarrays that have the same degree:

[1, 2, 2, 3, 1], [1, 2, 2, 3], [2, 2, 3, 1], [1, 2, 2], [2, 2, 3], [2, 2]

The shortest length is 2. So return 2.

Example 2:

Input: [1,2,2,3,1,4,2]

Output: 6

class Solution {

public int findShortestSubArray(int[] nums) {

int ans = 0;

Map<Integer, int[]> map = new HashMap<>();

for (int i = 0; i < nums.length; i++){

if (!map.containsKey(nums[i])){

map.put(nums[i], new int[]{1, i, i});;

}else{

int[] temp = map.get(nums[i]);

temp[0]++;

temp[2] = i;

}

}

int max = Integer.MIN\_VALUE;

for (int[] val : map.values()){

int freq = val[0];

int start = val[1];

int end = val[2];

if (freq > max){

max = freq;

ans = end - start + 1;

}else if (freq == max){

ans = Math.min(ans, end - start + 1);

}

}

return ans;

}

}

Method 2: One pass, O(N), best solution

class Solution {

public int findShortestSubArray(int[] nums) {

int res = 0;

int degree = 0;

Map<Integer, Integer> map = new HashMap<>();//key: num, val: current degree

Map<Integer, Integer> first = new HashMap<>();//key: num val: first occurance

for (int i = 0; i < nums.length; i++){

map.put(nums[i], map.getOrDefault(nums[i], 0) + 1);

first.putIfAbsent(nums[i], i);

if (degree < map.get(nums[i])){

degree = map.get(nums[i]);

res = i - first.get(nums[i]) + 1;

}else if (degree == map.get(nums[i])){

res = Math.min(res, i - first.get(nums[i]) + 1);

}

}

return res;

}

}

# 698\_Partition.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers nums and a positive integer k, find whether it's possible to divide this array into k non-empty

subsets whose sums are all equal.

Example 1:

Input: nums = [4, 3, 2, 3, 5, 2, 1], k = 4

Output: True

Explanation: It's possible to divide it into 4 subsets (5), (1, 4), (2,3), (2,3) with equal sums.

Note:

1 <= k <= len(nums) <= 16.

0 < nums[i] < 10000.

Similar as matchstick to make squre, this is also a partition (backtracking) problem

not the same as Partition Equal Subset Sum which is DP problem

Method: backtracking, must add Arrays.sort(nums) otherwise it will be TLE

class Solution {

public boolean canPartitionKSubsets(int[] nums, int k) {

int sum = 0;

for (int i = 0; i < nums.length; i++){

sum += nums[i];

}

if (sum % k != 0){

return false;

}

int target = sum / k;

Arrays.sort(nums);

int[] sums = new int[k];

return canParition(nums, target, sums, k, nums.length - 1); //pos must go from largest to smallest otherwise it will TLE

}

private boolean canParition(int[] nums, int target, int[] sums, int k, int pos){

if (pos < 0){

for (int i = 0; i < k; i++){

if (sums[i] != target){

return false;

}

}

return true;

}

for (int i = 0; i < k; i++){

if (sums[i] + nums[pos] > target){

continue;

}

sums[i] += nums[pos];

if (canParition(nums, target, sums, k, pos-1)){

return true;

}

sums[i] -= nums[pos];

}

return false;

}

}

Intuition

As even when k = 2, the problem is a "Subset Sum" problem which is known to be NP-hard, (and because the given input limits are low,)

our solution will focus on exhaustive search.

A natural approach is to simulate the k groups (disjoint subsets of nums). For each number in nums, we'll check whether putting it

in the i-th group solves the problem. We can check those possibilities by recursively searching.

Algorithm

Firstly, we know that each of the k group-sums must be equal to target = sum(nums) / k. (If this quantity is not an integer,

the task is impossible.)

For each number in nums, we could add it into one of k group-sums, as long as the group's sum would not exceed the target.

For each of these choices, we recursively search with one less number to consider in nums. If we placed every

number successfully, then our search was successful.

One important speedup is that we can ensure all the 0 values of each group occur at the end of the array groups,

by enforcing if (groups[i] == 0) break;. This greatly reduces repeated work - for example, in the first run of search,

we will make only 1 recursive call, instead of k. Actually, we could do better by skipping any repeated values of groups[i],

but it isn't necessary.

Another speedup is we could sort the array nums, so that we try to place the largest elements first. When the answer is true and

involves subsets with a low size, this method of placing elements will consider these lower size subsets sooner. We can also handle

elements nums[i] >= target appropriately. These tricks are not necessary to solve the problem, but they are presented in the solutions

below.

class Solution {

public boolean search(int[] groups, int row, int[] nums, int target) {

if (row < 0) return true;

int v = nums[row--];

for (int i = 0; i < groups.length; i++) {

if (groups[i] + v <= target) {

groups[i] += v;

if (search(groups, row, nums, target)) return true;

groups[i] -= v;

}

if (groups[i] == 0) break;

}

return false;

}

public boolean canPartitionKSubsets(int[] nums, int k) {

int sum = Arrays.stream(nums).sum();

if (sum % k > 0) return false;

int target = sum / k;

Arrays.sort(nums);

int row = nums.length - 1;

if (nums[row] > target) return false;

while (row >= 0 && nums[row] == target) {

row--;

k--;

}

return search(new int[k], row, nums, target);

}

}

# 7\_ReverseInteger.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 32-bit signed integer, reverse digits of an integer.

Example 1:

Input: 123

Output: 321

Example 2:

Input: -123

Output: -321

Example 3:

Input: 120

Output: 21

Note:

Assume we are dealing with an environment which could only hold integers within the 32-bit signed integer range.

For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

Method:

Note the difference of negative modulo between Java and Python which is the same as Math definition

In Java: -5 % 3 = -2

In Python: -5 % 3 = 1

Math definiition: a % n = a - floor(a/n) \* n

https://stackoverflow.com/questions/5385024/mod-in-java-produces-negative-numbers

class Solution {

public int reverse(int x) {

long ans = 0;

while (x != 0){

ans = ans \* 10 + x % 10;

x /= 10;

if (ans > Integer.MAX\_VALUE || ans < Integer.MIN\_VALUE){

return 0;

}

}

return (int)ans;

}

}

class Solution {

public int reverse(int x) {

int ans = 0;

while (x != 0){

int newResult = 10 \* ans + x % 10;

if ((newResult - x % 10) / 10 != ans){

return 0;

}

ans = newResult;

x /= 10;

}

return ans;

}

}

Best solution:

class Solution {

public int reverse(int x) {

long res = 0;

while (x != 0){

res \*= 10;

res += x % 10;

x /= 10;

}

return (int) res == res ? (int)res : 0;

}

}

# 70\_ClimbingStairs.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Note: Given n will be a positive integer.

Example 1:

Input: 2

Output: 2

Explanation: There are two ways to climb to the top.

1. 1 step + 1 step

2. 2 steps

Example 2:

Input: 3

Output: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

class Solution {

public int climbStairs(int n) {

if (n <= 2){

return n;

}

int[] f = new int[n];

f[0] = 1;

f[1] = 2;

for (int i = 2 ; i < n; i++){

f[i] = f[i-1] + f[i-2];

}

return f[n-1];

}

}

# 700\_Search.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the root node of a binary search tree (BST) and a value. You need to find the node in the BST that the node's value equals the given value. Return the subtree rooted with that node. If such node doesn't exist, you should return NULL.

For example,

Given the tree:

4

/ \

2 7

/ \

1 3

And the value to search: 2

You should return this subtree:

2

/ \

1 3

In the example above, if we want to search the value 5, since there is no node with value 5, we should return NULL.

Note that an empty tree is represented by NULL, therefore you would see the expected output (serialized tree format) as [], not null.

Method 1: recursionclass Solution {

public TreeNode searchBST(TreeNode root, int val) {

if (root == null){

return null;

}

if (root.val == val){

return root;

}else if (root.left < val){

return searchBST(root.left, val);

}else{

return searchBST(root.right, val);;

}

return null;

}

}

class Solution {

public TreeNode searchBST(TreeNode root, int val) {

if (root == null){

return null;

}

if (root.val == val){

return root;

}else if (root.val < val){

return searchBST(root.right, val);

}else{

return searchBST(root.left, val);

}

}

}

Method 2: iteration

class Solution {

public TreeNode searchBST(TreeNode root, int val) {

while (root != null && root.val != val){

if (root.val < val){

root = root.right;

}else{

root = root.left;

}

}

return root;

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode searchBST(TreeNode root, int val) {

if (root == null || root.val == val){

return root;

}

if (root.val > val){

return searchBST(root.left, val);

}

return searchBST(root.right, val);

}

}

# 701\_Insert.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the root node of a binary search tree (BST) and a value to be inserted into the tree, insert the value into the BST. Return the root node of the BST after the insertion. It is guaranteed that the new value does not exist in the original BST.

Note that there may exist multiple valid ways for the insertion, as long as the tree remains a BST after insertion. You can return any of them.

For example,

Given the tree:

4

/ \

2 7

/ \

1 3

And the value to insert: 5

You can return this binary search tree:

4

/ \

2 7

/ \ /

1 3 5

This tree is also valid:

5

/ \

2 7

/ \

1 3

\

4

Method 1: recursion

class Solution {

public TreeNode insertIntoBST(TreeNode root, int val) {

if (root == null){

root = new TreeNode(val);

}else{

insert(root, val);

}

return root;

}

private void insert(TreeNode root, int val){

if (root.val < val){

if (root.right == null){

TreeNode node = new TreeNode(val);

root.right = node;

return;

}

insertIntoBST(root.right, val);

}else{

if (root.left == null){

TreeNode node = new TreeNode(val);

root.left = node;

return;

}

insertIntoBST(root.left, val);

}

}

}

class Solution {

public TreeNode insertIntoBST(TreeNode root, int val) {

if (root == null){

return new TreeNode(val);

}else if (root.val < val){

root.right = insertIntoBST(root.right, val);

}else{

root.left = insertIntoBST(root.left, val);

}

return root;

}

}

Method 2: iteration

class Solution {

public TreeNode insertIntoBST(TreeNode root, int val) {

TreeNode cur = root;

while (cur != null){

if (cur.val < val){

if (cur.right == null){

cur.right = new TreeNode(val);

break;

}

cur = cur.right;

}else{

if (cur.left == null){

cur.left = new TreeNode(val);

break;

}

cur = cur.left;

}

}

return root;

}

}

Use inorder iteration template

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode insertIntoBST(TreeNode root, int val) {

Stack<TreeNode> stack = new Stack<>();

TreeNode curr = root;

while (curr != null || !stack.isEmpty()){

while (curr != null){

stack.push(curr);

curr = curr.left;

}

TreeNode node = stack.pop();

if (val < node.val){

TreeNode temp = node.left;

node.left = new TreeNode(val);

node.left.left = temp;

break;

}

curr = node.right;

if (stack.isEmpty() && curr == null){

if (val > node.val){

node.right = new TreeNode(val);

}

}

}

return root;

}

}

# 702\_Search.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array sorted in ascending order, write a function to search target in nums. If target exists, then return its index,

otherwise return -1. However, the array size is unknown to you. You may only access the array using an ArrayReader interface, where

ArrayReader.get(k) returns the element of the array at index k (0-indexed).

You may assume all integers in the array are less than 10000, and if you access the array out of bounds, ArrayReader.get will

return 2147483647.

Example 1:

Input: array = [-1,0,3,5,9,12], target = 9

Output: 4

Explanation: 9 exists in nums and its index is 4

Example 2:

Input: array = [-1,0,3,5,9,12], target = 2

Output: -1

Explanation: 2 does not exist in nums so return -1

Note:

You may assume that all elements in the array are unique.

The value of each element in the array will be in the range [-9999, 9999].

Method 1:

class Solution {

public int search(ArrayReader reader, int target) {

int start = 0;

int end = 20000;

while (start <= end){

int mid = start + (end - start) / 2;

int val = reader.get(mid);

if (val == 2147483647){

end = mid -1 ;

}else if (val == target){

return mid;

}else if (val < target){

start = mid + 1;

}else{

end = mid - 1;

}

}

return -1;

}

}

class Solution {

public int search(ArrayReader reader, int target) {

int start = 0;

int end = 20000;

while (start <= end){

int mid = start + (end - start) / 2;

int val = reader.get(mid);

if (val == target){

return mid;

}else if (val < target){

start = mid + 1;

}else{

end = mid - 1;

}

}

return -1;

}

}

Method 2: Best solution

class Solution {

public int search(ArrayReader reader, int target) {

int end = 1;

while (reader.get(end) < target){

end <<= 1;

}

int start = end >> 1;

while (start <= end){

int mid = start + (end - start) / 2;

int val = reader.get(mid);

if (val == target){

return mid;

}else if (val < target){

start = mid + 1;

}else{

end = mid - 1;

}

}

return -1;

}

}

# 703\_Kth.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a class to find the kth largest element in a stream. Note that it is the kth largest element in the sorted order, not the

kth distinct element.

Your KthLargest class will have a constructor which accepts an integer k and an integer array nums, which contains initial elements

from the stream. For each call to the method KthLargest.add, return the element representing the kth largest element in the stream.

Example:

int k = 3;

int[] arr = [4,5,8,2];

KthLargest kthLargest = new KthLargest(3, arr);

kthLargest.add(3); // returns 4

kthLargest.add(5); // returns 5

kthLargest.add(10); // returns 5

kthLargest.add(9); // returns 8

kthLargest.add(4); // returns 8

Note:

You may assume that nums' length ≥ k-1 and k ≥ 1.

Method 1: Two PQ

class KthLargest {

Queue<Integer> minPQ;

Queue<Integer> maxPQ;

int K;

public KthLargest(int k, int[] nums) {

minPQ = new PriorityQueue<Integer>();

maxPQ = new PriorityQueue<Integer>(new Comparator<Integer>(){

public int compare (Integer i1, Integer i2){

return i2 - i1;

}

});

K = k;

int N = nums.length;

for (int num : nums){

minPQ.offer(num);

}

for (int i = 0; i < N - k + 1; i++){

maxPQ.offer(minPQ.poll());

}

}

public int add(int val) {

if (minPQ.isEmpty()){

minPQ.offer(val);

if (minPQ.size() > K - 1){

maxPQ.offer(minPQ.poll());

}

}else if (val <= minPQ.peek()){

maxPQ.offer(val);

}else{

maxPQ.offer(minPQ.poll());

minPQ.offer(val);

}

return maxPQ.peek();

}

}

/\*\*

\* Your KthLargest object will be instantiated and called as such:

\* KthLargest obj = new KthLargest(k, nums);

\* int param\_1 = obj.add(val);

\*/

Method 2: One PQ (Best solution)

class KthLargest {

Queue<Integer> minPQ;

int K;

public KthLargest(int k, int[] nums) {

minPQ = new PriorityQueue<Integer>();

K = k;

for (int num : nums){

if (minPQ.size() < k || num > minPQ.peek()){

minPQ.offer(num);

}

if (minPQ.size() > k){

minPQ.poll();

}

}

}

public int add(int val) {

if (minPQ.size() < K || val > minPQ.peek()){

minPQ.offer(val);

}

if (minPQ.size() > K){

minPQ.poll();

}

return minPQ.peek();

}

}

/\*\*

\* Your KthLargest object will be instantiated and called as such:

\* KthLargest obj = new KthLargest(k, nums);

\* int param\_1 = obj.add(val);

\*/

# 704\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted (in ascending order) integer array nums of n elements and a target value, write a function to search target in nums. If target exists, then return its index, otherwise return -1.

Example 1:

Input: nums = [-1,0,3,5,9,12], target = 9

Output: 4

Explanation: 9 exists in nums and its index is 4

Example 2:

Input: nums = [-1,0,3,5,9,12], target = 2

Output: -1

Explanation: 2 does not exist in nums so return -1

Note:

You may assume that all elements in nums are unique.

n will be in the range [1, 10000].

The value of each element in nums will be in the range [-9999, 9999].

Method:

class Solution {

public int search(int[] nums, int target) {

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return mid;

}else if (nums[mid] < target){

start = mid;

}else{

end = mid;

}

}

if (nums[start] == target){

return start;

}else if(nums[end] == target){

return end;

}

return -1;

}

}

class Solution {

public int search(int[] nums, int target) {

int start = 0;

int end = nums.length - 1;

while (start <= end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return mid;

}else if (nums[mid] < target){

start = mid + 1;

}else{

end = mid - 1;

}

}

return -1;

}

}

# 705\_DesignHashSet.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a HashSet without using any built-in hash table libraries.

To be specific, your design should include these functions:

add(value): Insert a value into the HashSet.

contains(value) : Return whether the value exists in the HashSet or not.

remove(value): Remove a value in the HashSet. If the value does not exist in the HashSet, do nothing.

Example:

MyHashSet hashSet = new MyHashSet();

hashSet.add(1);

hashSet.add(2);

hashSet.contains(1); // returns true

hashSet.contains(3); // returns false (not found)

hashSet.add(2);

hashSet.contains(2); // returns true

hashSet.remove(2);

hashSet.contains(2); // returns false (already removed)

Note:

All values will be in the range of [0, 1000000].

The number of operations will be in the range of [1, 10000].

Please do not use the built-in HashSet library.

class MyHashSet {

private int[] arr;

/\*\* Initialize your data structure here. \*/

public MyHashSet() {

arr = new int[1000001];

for (int i = 0; i < arr.length; i++){

arr[i] = -1;

}

}

public void add(int key) {

arr[key] = key;

}

public void remove(int key) {

arr[key] = -1;

}

/\*\* Returns true if this set contains the specified element \*/

public boolean contains(int key) {

return arr[key] != -1;

}

}

/\*\*

\* Your MyHashSet object will be instantiated and called as such:

\* MyHashSet obj = new MyHashSet();

\* obj.add(key);

\* obj.remove(key);

\* boolean param\_3 = obj.contains(key);

\*/

# 706\_DesignHashMap.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a HashMap without using any built-in hash table libraries.

To be specific, your design should include these functions:

put(key, value) : Insert a (key, value) pair into the HashMap. If the value already exists in the HashMap, update the value.

get(key): Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key.

remove(key) : Remove the mapping for the value key if this map contains the mapping for the key.

Example:

MyHashMap hashMap = new MyHashMap();

hashMap.put(1, 1);

hashMap.put(2, 2);

hashMap.get(1); // returns 1

hashMap.get(3); // returns -1 (not found)

hashMap.put(2, 1); // update the existing value

hashMap.get(2); // returns 1

hashMap.remove(2); // remove the mapping for 2

hashMap.get(2); // returns -1 (not found)

Note:

All keys and values will be in the range of [0, 1000000].

The number of operations will be in the range of [1, 10000].

Please do not use the built-in HashMap library.

Collisions are resolved using linked list

Best Solution:

class MyHashMap {

class ListNode {

int key;

int val;

ListNode next;

public ListNode(int key, int val){

this.key = key;

this.val = val;

this.next = null;

}

}

/\*\* Initialize your data structure here. \*/

private int N;

private ListNode[] nodes;

public MyHashMap() {

N = 10000;

nodes = new ListNode[N+1];

}

public int findBucket(int key){

return Integer.hashCode(key) % (N+1);

}

public ListNode findPrev(ListNode bucket, int key){

ListNode node = bucket;

ListNode prev = null;

while (node != null && node.key != key){

prev = node;

node = node.next;

}

return prev;

}

/\*\* value will always be non-negative. \*/

public void put(int key, int value) {

int index = findBucket(key);

if (nodes[index] == null){

nodes[index] = new ListNode(-1, -1);

}

ListNode prev = findPrev(nodes[index], key);

if (prev.next == null){

prev.next = new ListNode(key, value);

}else{

prev.next.val = value;

}

}

/\*\* Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key \*/

public int get(int key) {

int index = findBucket(key);

if (nodes[index] != null){

ListNode prev = findPrev(nodes[index], key);

if (prev.next != null){

return prev.next.val;

}

}

return -1;

}

/\*\* Removes the mapping of the specified value key if this map contains a mapping for the key \*/

public void remove(int key) {

int index = findBucket(key);

if (nodes[index] != null){

ListNode prev = findPrev(nodes[index], key);

if (prev.next != null){

prev.next = prev.next.next;

}

}

}

}

/\*\*

\* Your MyHashMap object will be instantiated and called as such:

\* MyHashMap obj = new MyHashMap();

\* obj.put(key,value);

\* int param\_2 = obj.get(key);

\* obj.remove(key);

\*/

class MyHashMap {

int[] arr;

/\*\* Initialize your data structure here. \*/

public MyHashMap() {

int N = 1000001;

arr = new int[N];

for (int i = 0; i < N; i++){

arr[i] = -1;

}

}

/\*\* value will always be non-negative. \*/

public void put(int key, int value) {

arr[key] = value;

}

/\*\* Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key \*/

public int get(int key) {

return arr[key];

}

/\*\* Removes the mapping of the specified value key if this map contains a mapping for the key \*/

public void remove(int key) {

arr[key] = -1;

}

}

/\*\*

\* Your MyHashMap object will be instantiated and called as such:

\* MyHashMap obj = new MyHashMap();

\* obj.put(key,value);

\* int param\_2 = obj.get(key);

\* obj.remove(key);

\*/

# 707\_DesignLinkedList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design your implementation of the linked list. You can choose to use the singly linked list or the doubly linked list. A node in a singly linked list should have two attributes: val and next. val is the value of the current node, and next is a pointer/reference to the next node. If you want to use the doubly linked list, you will need one more attribute prev to indicate the previous node in the linked list. Assume all nodes in the linked list are 0-indexed.

Implement these functions in your linked list class:

get(index) : Get the value of the index-th node in the linked list. If the index is invalid, return -1.

addAtHead(val) : Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.

addAtTail(val) : Append a node of value val to the last element of the linked list.

addAtIndex(index, val) : Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted.

deleteAtIndex(index) : Delete the index-th node in the linked list, if the index is valid.

Example:

MyLinkedList linkedList = new MyLinkedList();

linkedList.addAtHead(1);

linkedList.addAtTail(3);

linkedList.addAtIndex(1, 2); // linked list becomes 1->2->3

linkedList.get(1); // returns 2

linkedList.deleteAtIndex(1); // now the linked list is 1->3

linkedList.get(1); // returns 3

Note:

All values will be in the range of [1, 1000].

The number of operations will be in the range of [1, 1000].

Please do not use the built-in LinkedList library.

class MyLinkedList {

class SingleNode {

int val;

SingleNode next;

public SingleNode (int val){

this.val = val;

}

}

SingleNode head;

/\*\* Initialize your data structure here. \*/

public MyLinkedList() {

head = null;

}

/\*\* Get the value of the index-th node in the linked list. If the index is invalid, return -1. \*/

public int get(int index) {

int count = 0;

SingleNode node = head;

while (node != null && count < index){

node = node.next;

count++;

}

if (count == index && node != null){

return node.val;

}

return -1;

}

/\*\* Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list. \*/

public void addAtHead(int val) {

SingleNode node = new SingleNode(val);

node.next = head; //note how to add head note//

head = node;

}

/\*\* Append a node of value val to the last element of the linked list. \*/

public void addAtTail(int val) {

SingleNode node = head;

while (node != null && node.next != null){

node = node.next;

}

node.next = new SingleNode(val);

}

/\*\* Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted. \*/

public void addAtIndex(int index, int val) {

int count = 0;

SingleNode node = head;

while (node != null && count < index - 1){

node = node.next;

count++;

}

if (index == 0){//special case

SingleNode newNode = new SingleNode(val);

newNode.next = head;

head = newNode;

}else if (count == index - 1 && node != null){

SingleNode newNode = new SingleNode(val);

newNode.next = node.next;

node.next = newNode;

}

}

/\*\* Delete the index-th node in the linked list, if the index is valid. \*/

public void deleteAtIndex(int index) {

int count = 0;

SingleNode node = head;

while (node != null && count < index - 1){

node = node.next;

count++;

}

if (count == index -1 && node != null && node.next != null){

node.next = node.next.next;

}

}

}

/\*\*

\* Your MyLinkedList object will be instantiated and called as such:

\* MyLinkedList obj = new MyLinkedList();

\* int param\_1 = obj.get(index);

\* obj.addAtHead(val);

\* obj.addAtTail(val);

\* obj.addAtIndex(index,val);

\* obj.deleteAtIndex(index);

\*/

# 708\_insert.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a node from a cyclic linked list which is sorted in ascending order, write a function to insert a value into the list such that

it remains a cyclic sorted list. The given node can be a reference to any single node in the list, and may not be necessarily the

smallest value in the cyclic list.

If there are multiple suitable places for insertion, you may choose any place to insert the new value. After the insertion, the cyclic

list should remain sorted.

If the list is empty (i.e., given node is null), you should create a new single cyclic list and return the reference to that single

node. Otherwise, you should return the original given node.

The following example may help you understand the problem better:

https://www.cnblogs.com/grandyang/p/9981163.html

class Solution{

class Node{

int val;

Node next;

public Node(int val){

this.val = val;

next = null;

}

}

Node insert(Node head, int insertVal){

if (head == null){

Node newHead = new Node(insertVal);

newHead.next = newHead;

return newHead;

}

Node prev = head;

Node curr = head.next;

while (curr != head){

if (prev.val <= insertVal && insertVal <= curr.val){

break;

}

if (prev.val > curr.val && (prev.val <= insertVal || insertVal <= curr.val)){

break;

}

prev = curr;

curr = curr.next;

}

Node newNode = new Node(insertVal);

prev.next = newNode;

newNode.next = curr;

return head;

}

}

709\_To.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement function ToLowerCase() that has a string parameter str, and returns the same string in lowercase.

Example 1:

Input: "Hello"

Output: "hello"

Example 2:

Input: "here"

Output: "here"

Example 3:

Input: "LOVELY"

Output: "lovely"

class Solution {

public String toLowerCase(String str) {

StringBuilder sb = new StringBuilder();

for (int i = 0; i < str.length(); i++){

char c = str.charAt(i);

if (c <= 'Z' && c >= 'A'){

sb.append((char)(c - 'A' + 'a'));

}else{

sb.append(c);

}

}

return sb.toString();

}

}

# 71\_Simplify.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an absolute path for a file (Unix-style), simplify it.

For example,

path = "/home/", => "/home"

path = "/a/./b/../../c/", => "/c"

Corner Cases:

Did you consider the case where path = "/../"?

In this case, you should return "/".

Another corner case is the path might contain multiple slashes '/' together, such as "/home//foo/".

In this case, you should ignore redundant slashes and return "/home/foo".

class Solution {

public String simplifyPath(String path) {

Stack<String> stack = new Stack<>();

Set<String> skip = new HashSet<String>(Arrays.asList(".", "..", ""));

String[] dirs = path.split("/");

for (String dir : dirs){

if (dir.equals("..") && !stack.isEmpty()){

stack.pop();

}else if (!skip.contains(dir)){

stack.push(dir);

}

}

String res = "";

for (String dir : stack){

res = res + "/" + dir;

}

return res.length() == 0 ? "/" : res;

}

}

# 712\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings s1, s2, find the lowest ASCII sum of deleted characters to make two strings equal.

Example 1:

Input: s1 = "sea", s2 = "eat"

Output: 231

Explanation: Deleting "s" from "sea" adds the ASCII value of "s" (115) to the sum.

Deleting "t" from "eat" adds 116 to the sum.

At the end, both strings are equal, and 115 + 116 = 231 is the minimum sum possible to achieve this.

Example 2:

Input: s1 = "delete", s2 = "leet"

Output: 403

Explanation: Deleting "dee" from "delete" to turn the string into "let",

adds 100[d]+101[e]+101[e] to the sum. Deleting "e" from "leet" adds 101[e] to the sum.

At the end, both strings are equal to "let", and the answer is 100+101+101+101 = 403.

If instead we turned both strings into "lee" or "eet", we would get answers of 433 or 417, which are higher.

Note:

0 < s1.length, s2.length <= 1000.

All elements of each string will have an ASCII value in [97, 122].

Method: DP

the same as Edit Distance, Delete Operation for two strings

Convert character to ASCII:

char character = 'a';

int ascii = (int) character;

Let dp[i][j] be the answer to the problem for the strings s1[i:], s2[j:].

When one of the input strings is empty, the answer is the ASCII-sum of the other string. We can calculate this cumulatively using code like dp[i][s2.length()] = dp[i+1][s2.length()] + s1.codePointAt(i).

When s1[i] == s2[j], we have dp[i][j] = dp[i+1][j+1] as we can ignore these two characters.

When s1[i] != s2[j], we will have to delete at least one of them. We'll have dp[i][j] as the minimum of the answers after both deletion options.

The solutions presented will use bottom-up dynamic programming.

class Solution {

public int minimumDeleteSum(String s1, String s2) {

int m = s1.length();

int n = s2.length();

int[][] dp = new int[m+1][n+1];

dp[0][0] = 0;

for (int i = 1; i <= m; i++){

dp[i][0] = (int) s1.charAt(i-1) + dp[i-1][0];

}

for (int j = 1; j <= n; j++){

dp[0][j] = (int) s2.charAt(j-1) + dp[0][j-1];

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (s1.charAt(i-1) == s2.charAt(j-1)){

dp[i][j] = dp[i-1][j-1];

}else{

dp[i][j] = Math.min(dp[i-1][j] + (int) s1.charAt(i-1), dp[i][j-1] + (int) s2.charAt(j-1));

}

}

}

return dp[m][n];

}

}

class Solution {

public int minimumDeleteSum(String s1, String s2) {

int m = s1.length();

int n = s2.length();

int[][] dp = new int[m+1][n+1];

for (int i = 1; i <= m; i++){

dp[i][0] = dp[i-1][0] + s1.charAt(i-1);

}

for (int j = 1; j <= n; j++){

dp[0][j] = dp[0][j-1] + s2.charAt(j-1);

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (s1.charAt(i-1) == s2.charAt(j-1)){

dp[i][j] = dp[i-1][j-1];

}else{

dp[i][j] = Math.min(dp[i-1][j] + s1.charAt(i-1), dp[i][j-1] + s2.charAt(j-1));

}

}

}

return dp[m][n];

}

}

# 713\_SubarrayProductLessThanK.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Your are given an array of positive integers nums.

Count and print the number of (contiguous) subarrays where the product of all the elements in the subarray is less than k.

Example 1:

Input: nums = [10, 5, 2, 6], k = 100

Output: 8

Explanation: The 8 subarrays that have product less than 100 are: [10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6].

Note that [10, 5, 2] is not included as the product of 100 is not strictly less than k.

Note:

0 < nums.length <= 50000.

0 < nums[i] < 1000.

0 <= k < 10^6.

The idea is always keep an max-product-window less than K;

Every time shift window by adding a new number on the right(j), if the product is greater than k,

then try to reduce numbers on the left(i), until the subarray product fit less than k again, (subarray could be empty);

Each step introduces x new subarrays, where x is the size of the current window (j + 1 - i);

example:

for window (5, 2), when 6 is introduced, it add 3 new subarray: (5, (2, (6)))

(6)

(2, 6)

(5, 2, 6)

class Solution {

public int numSubarrayProductLessThanK(int[] nums, int k) {

if (nums == null || nums.length == 0 || k <= 0){

return 0;

}

int count = 0;

int product = 1;

for (int start = 0, end = 0; end < nums.length; end++){

product \*= nums[end];

while (product >= k && start <= end){

product /= nums[start];

start++;

}

count += 1 + end - start;

}

return count;

}

}

Better version:

class Solution {

public int numSubarrayProductLessThanK(int[] nums, int k) {

int res = 0;

int start = 0;

int end = 0;

int prod = 1;

while (end < nums.length){

prod \*= nums[end];

if (prod < k){

res += end - start + 1;

}else{

while (prod >= k && start < end){

prod /= nums[start];

start++;

}

if (prod < k){

res += end - start + 1;

}

}

end++;

}

return res;

}

}

# 714\_BestTimewithTransactionFee.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Your are given an array of integers prices, for which the i-th element is the price of a given stock on day i;

and a non-negative integer fee representing a transaction fee.

You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

You may not buy more than 1 share of a stock at a time (ie. you must sell the stock share before you buy again.)

Return the maximum profit you can make.

Example 1:

Input: prices = [1, 3, 2, 8, 4, 9], fee = 2

Output: 8

Explanation: The maximum profit can be achieved by:

Buying at prices[0] = 1

Selling at prices[3] = 8

Buying at prices[4] = 4

Selling at prices[5] = 9

The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

Note:

0 < prices.length <= 50000.

0 < prices[i] < 50000.

0 <= fee < 50000.

class Solution {

public int maxProfit(int[] prices, int fee) {

int maxProfit = 0;

int minCost = Integer.MAX\_VALUE;

for (int i = 0; i < prices.length; i++){

minCost = Math.min(minCost, prices[i] - maxProfit);

maxProfit = Math.max(maxProfit, prices[i] - minCost - fee);

}

return maxProfit;

}

}

# 715\_Range.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A Range Module is a module that tracks ranges of numbers. Your task is to design and implement the following interfaces in an efficient manner.

addRange(int left, int right) Adds the half-open interval [left, right), tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval [left, right) that are not already tracked.

queryRange(int left, int right) Returns true if and only if every real number in the interval [left, right) is currently being tracked.

removeRange(int left, int right) Stops tracking every real number currently being tracked in the interval [left, right).

Example 1:

addRange(10, 20): null

removeRange(14, 16): null

queryRange(10, 14): true (Every number in [10, 14) is being tracked)

queryRange(13, 15): false (Numbers like 14, 14.03, 14.17 in [13, 15) are not being tracked)

queryRange(16, 17): true (The number 16 in [16, 17) is still being tracked, despite the remove operation)

Note:

A half open interval [left, right) denotes all real numbers left <= x < right.

0 < left < right < 10^9 in all calls to addRange, queryRange, removeRange.

The total number of calls to addRange in a single test case is at most 1000.

The total number of calls to queryRange in a single test case is at most 5000.

The total number of calls to removeRange in a single test case is at most 1000.

Method 1: O(n) TLE

class RangeModule {

Set<Integer> trackSet;

public RangeModule() {

trackSet = new HashSet<>();

}

public void addRange(int left, int right) {

for (int i = left; i < right; i++){

trackSet.add(i);

}

}

public boolean queryRange(int left, int right) {

for (int i = left; i < right; i++){

if (!trackSet.contains(i)){

return false;

}

}

return true;

}

public void removeRange(int left, int right) {

for (int i = left; i < right; i++){

if (trackSet.contains(i)){

trackSet.remove(Integer.valueOf(i));

}

}

}

}

/\*\*

\* Your RangeModule object will be instantiated and called as such:

\* RangeModule obj = new RangeModule();

\* obj.addRange(left,right);

\* boolean param\_2 = obj.queryRange(left,right);

\* obj.removeRange(left,right);

\*/

# 716\_MaxStack.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design a max stack that supports push, pop, top, peekMax and popMax.

push(x) -- Push element x onto stack.

pop() -- Remove the element on top of the stack and return it.

top() -- Get the element on the top.

peekMax() -- Retrieve the maximum element in the stack.

popMax() -- Retrieve the maximum element in the stack, and remove it. If you find more than one maximum elements, only remove the top-most one.

Example 1:

MaxStack stack = new MaxStack();

stack.push(5);

stack.push(1);

stack.push(5);

stack.top(); -> 5

stack.popMax(); -> 5

stack.top(); -> 1

stack.peekMax(); -> 5

stack.pop(); -> 1

stack.top(); -> 5

Note:

-1e7 <= x <= 1e7

Number of operations won't exceed 10000.

The last four operations won't be called when stack is empty.

Similar to 155. Min Stack Method 2

Method 1: Two stacks O(N)

class MaxStack {

Stack<Integer> stack;

Stack<Integer> maxStack;

/\*\* initialize your data structure here. \*/

public MaxStack() {

stack = new Stack<Integer>();

maxStack = new Stack<>();

}

public void push(int x) {

stack.push(x);

if (maxStack.isEmpty()){

maxStack.push(x);

}else{

maxStack.push(Math.max(maxStack.peek(), x));

}

}

public int pop() {

maxStack.pop();

return stack.pop();

}

public int top() {

return stack.peek();

}

public int peekMax() {

return maxStack.peek();

}

public int popMax() {

int max = maxStack.peek();

Stack<Integer> buffer = new Stack<>();

while (top() != max){

buffer.push(pop()); // must be pop(), not stack.pop() or maxStack.pop()

}

stack.pop();

maxStack.pop();

while (!buffer.isEmpty()){

push(buffer.pop());

}

return max;

}

}

/\*\*

\* Your MaxStack object will be instantiated and called as such:

\* MaxStack obj = new MaxStack();

\* obj.push(x);

\* int param\_2 = obj.pop();

\* int param\_3 = obj.top();

\* int param\_4 = obj.peekMax();

\* int param\_5 = obj.popMax();

\*/

Method 2: Double Linked List + TreeMap

O(LogN)

https://leetcode.com/articles/max-stack/

# 717\_bit.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have two special characters. The first character can be represented by one bit 0. The second character can be represented by two bits (10 or 11).

Now given a string represented by several bits. Return whether the last character must be a one-bit character or not. The given string will always end with a zero.

Example 1:

Input:

bits = [1, 0, 0]

Output: True

Explanation:

The only way to decode it is two-bit character and one-bit character. So the last character is one-bit character.

Example 2:

Input:

bits = [1, 1, 1, 0]

Output: False

Explanation:

The only way to decode it is two-bit character and two-bit character. So the last character is NOT one-bit character.

Note:

1 <= len(bits) <= 1000.

bits[i] is always 0 or 1.

class Solution {

public boolean isOneBitCharacter(int[] bits) {

int ones = 0;

for (int i = bits.length - 2; i >= 0 && bits[i] != 0; i--){

ones++;

}

return (ones % 2 == 0);

}

}

# 718\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two integer arrays A and B, return the maximum length of an subarray that appears in both arrays.

Example 1:

Input:

A: [1,2,3,2,1]

B: [3,2,1,4,7]

Output: 3

Explanation:

The repeated subarray with maximum length is [3, 2, 1].

Note:

1 <= len(A), len(B) <= 1000

0 <= A[i], B[i] < 100

Method: Same idea of Longest Common Substring Best solution

dp[i][j] is the length of longest common subarray which MUST end with nums[i-1] and nums[j-1]

class Solution {

public int findLength(int[] A, int[] B) {

int m = A.length;

int n = B.length;

int[][] dp = new int[m+1][n+1];

int max = 0;

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (A[i-1] == B[j-1]){

dp[i][j] = dp[i-1][j-1] + 1;

max = Math.max(max, dp[i][j]);

}

}

}

return max;

}

}

Note that the following code is for longest common subsequence not longest common substring

dp[i][j] is the length of longest common subarray wich May or May NOT end with nums[i-1] and nums[j-1]

Example

For "ABCD" and "EDCA", the LCS is "A" (or "D", "C"), return 1.

For "ABCD" and "EACB", the LCS is "AC", return 2.

http://www.jiuzhang.com/solutions/longest-common-subsequence/

public class Solution {

/\*\*

\* @param A: A string

\* @param B: A string

\* @return: The length of longest common subsequence of A and B

\*/

public int longestCommonSubsequence(String A, String B) {

int m = A.length();

int n = B.length();

int[][] dp = new int[m+1][n+1];

for (int i = 0; i <= m; i++){

dp[i][0] = 0;

}

for (int j = 0; j <= n; j++){

dp[0][j] = 0;

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (A.charAt(i-1) == B.charAt(j-1)){

dp[i][j] = Math.max(dp[i-1][j-1] + 1, Math.max(dp[i-1][j], dp[i][j-1]));

}else{

dp[i][j] = Math.max(dp[i-1][j-1], Math.max(dp[i-1][j], dp[i][j-1]));

}

}

}

return dp[m][n];

}

}

# 719\_FindK-th.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array, return the k-th smallest distance among all the pairs. The distance of a pair (A, B) is defined

as the absolute difference between A and B.

Example 1:

Input:

nums = [1,3,1]

k = 1

Output: 0

Explanation:

Here are all the pairs:

(1,3) -> 2

(1,1) -> 0

(3,1) -> 2

Then the 1st smallest distance pair is (1,1), and its distance is 0.

Note:

2 <= len(nums) <= 10000.

0 <= nums[i] < 1000000.

1 <= k <= len(nums) \* (len(nums) - 1) / 2.

Method 1: Brute Force

Time complexity: O(n^2\*logk)

Space complexity: O(k)

class Solution {

public int smallestDistancePair(int[] nums, int k) {

int n = nums.length;

Queue<Integer> maxPQ = new PriorityQueue<>(new Comparator<Integer>(){

public int compare (Integer i1, Integer i2){

return i2 - i1;

}

});

for (int i = 0; i < n ; i++){

for (int j = i + 1; j < nums.length; j++){

int cand = Math.abs(nums[i] - nums[j]);

if (maxPQ.size() < k){

maxPQ.offer(cand);

}else if (maxPQ.size() == k && maxPQ.peek() > cand){

maxPQ.poll();

maxPQ.offer(cand);

}

}

}

return maxPQ.peek();

}

}

Method 2:

way to define the K-th smallest pair distance: given an integer num, let count(num) denote the number of pair

distances that are no greater than num, then the K-th smallest pair distance will be the smallest integer such that count(num) >= K.

If we sort the input array in ascending order, this problem can actually be rephrased as finding the kth smallest element in a sorted

matrix, where the matrix element at position (i, j) is given by matrix[i][j] = nums[j] - nums[i]

https://leetcode.com/problems/find-k-th-smallest-pair-distance/discuss/109082/Approach-the-problem-using-the-%22trial-and-error%22-algorithm

Time complexity: O(nlogn + nlogd) d is the max - min

Space complexity: O(1)

class Solution {

public int smallestDistancePair(int[] nums, int k) {

Arrays.sort(nums);

int n = nums.length;

int low = 0;

int high = nums[n-1] - nums[0];

while (low <= high){

int mid = low + (high - low) / 2;

int count = getLessEqual(nums, mid);

if (count <= k - 1){

low = mid + 1;

}else{

high = mid - 1;

}

}

return low;

}

private int getLessEqual(int[] nums, int val){

int res = 0;

for (int i = 0; i < nums.length; i++){

int j = i + 1;

while (j < nums.length && nums[j] - nums[i] <= val){

j++;

}

res += j - i - 1;

}

return res;

}

}

# 72\_EditDistance.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as

1 step.)

You have the following 3 operations permitted on a word:

Insert a character

Delete a character

Replace a character

Have you met this question in a real interview? Yes

Example

Given word1 = "mart" and word2 = "karma", return 3.

Let following be the function definition :-

f(i, j) := minimum cost (or steps) required to convert first i characters of word1 to first j characters of word2

Case 1: word1[i] == word2[j], i.e. the ith the jth character matches.

f(i, j) = f(i - 1, j - 1)

Case 2: word1[i] != word2[j], then we must either insert, delete or replace, whichever is cheaper

f(i, j) = 1 + min { f(i, j - 1), f(i - 1, j), f(i - 1, j - 1) }

f(i, j - 1) represents insert operation

f(i - 1, j) represents delete operation

f(i - 1, j - 1) represents replace operation

Here, we consider any operation from word1 to word2. It means, when we say insert operation,

we insert a new character after word1 that matches the jth character of word2.

So, now have to match i characters of word1 to j - 1 characters of word2. Same goes for other 2 operations as well.

Note that the problem is symmetric. The insert operation in one direction (i.e. from word1 to word2)

is same as delete operation in other. So, we could choose any direction.

Above equations become the recursive definitions for DP.

Base Case:

f(0, k) = f(k, 0) = k

public class Solution {

/\*

\* @param word1: A string

\* @param word2: A string

\* @return: The minimum number of steps.

\*/

public int minDistance(String word1, String word2) {

int n = word1.length();

int m = word2.length();

int[][] cost = new int[n+1][m+1];

for (int i = 0; i <= n; i++){

cost[i][0] = i;

}

for (int j = 0; j <= m; j++){

cost[0][j] = j;

}

for (int i = 1; i <= n; i++){

for (int j = 1; j <= m; j++){

// note that cost[i][j] represents the comparison at the point between i - 1 of word1 and j - 1 of word2

if (word1.charAt(i-1) == word2.charAt(j-1)){

cost[i][j] = cost[i-1][j-1];

}else{

cost[i][j] = 1 + Math.min(cost[i-1][j-1],Math.min(cost[i][j-1], cost[i-1][j]));

}

}

}

return cost[n][m];

}

}

# 720\_Longest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of strings words representing an English Dictionary, find the longest word in words that can be built one character at a time by other words in words. If there is more than one possible answer, return the longest word with the smallest lexicographical order.

If there is no answer, return the empty string.

Example 1:

Input:

words = ["w","wo","wor","worl", "world"]

Output: "world"

Explanation:

The word "world" can be built one character at a time by "w", "wo", "wor", and "worl".

Example 2:

Input:

words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]

Output: "apple"

Explanation:

Both "apply" and "apple" can be built from other words in the dictionary. However, "apple" is lexicographically smaller than "apply".

Note:

All the strings in the input will only contain lowercase letters.

The length of words will be in the range [1, 1000].

The length of words[i] will be in the range [1, 30].

Methodd 1: Trie

Similar to Map Sum Trie and Longest Word in Dictionary through Deleting

Time complexity: O(n \* log(26)) -> O(n)

Space complexity: O(n)

class Solution {

class TrieNode{

TrieNode[] children;

String str;

TrieNode (){

children = new TrieNode[26];

}

}

public String longestWord(String[] words) {

TrieNode root = new TrieNode();

for (String word : words){

TrieNode cur = root;

for (int i = 0; i < word.length(); i++){

if (cur.children[word.charAt(i) - 'a'] == null){

cur.children[word.charAt(i) - 'a'] = new TrieNode();

}

cur = cur.children[word.charAt(i) - 'a'];

}

cur.str = word;

}

String ans = "";

for (String word : words){

TrieNode cur = root;

for (int i = 0; i < word.length(); i++){

if (cur.children[word.charAt(i) - 'a'] != null){

cur = cur.children[word.charAt(i) - 'a'];

if (cur.str == null){

break;

}

}

}

if (cur.str != null && cur.str.length() >= ans.length()){;

if (ans.equals("") || cur.str.length() > ans.length() || cur.str.compareTo(ans) < 0){

ans = cur.str;

}

}

}

return ans;

}

}

class Solution {

class TrieNode{

TrieNode[] children;

boolean isEnd;

public TrieNode(){

children = new TrieNode[26];

isEnd = false;

}

}

public String longestWord(String[] words) {

TrieNode root = new TrieNode();

for(String word : words){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c- 'a'];

}

node.isEnd = true;

}

String res = "";

for (String word : words){

TrieNode node = root;

boolean found = true;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

node = node.children[c- 'a'];

if (!node.isEnd){

found = false;

break;

}

}

if (found){

if (word.length() > res.length() || word.length() == res.length() && word.compareTo(res) < 0){

res = word;

}

}

}

return res;

}

}

https://leetcode.com/problems/longest-word-in-dictionary/discuss/109114/JavaC++-Clean-Code

Best solution: build HashMap

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public String longestWord(String[] words) {

Arrays.sort(words);

Set<String> built = new HashSet<String>();

String res = "";

for (String w : words) {

if (w.length() == 1 || built.contains(w.substring(0, w.length() - 1))) {

res = w.length() > res.length() ? w : res;

built.add(w);

}

}

return res;

}

}

Method 2: Brute Force

Intuition

For each word, check if all prefixes word[:k] are present. We can use a Set structure to check this quickly.

Algorithm

Whenever our found word would be superior, we check if all it's prefixes are present, then replace our answer.

Alternatively, we could have sorted the words beforehand, so that we know the word we are considering would be the answer if all it's prefixes are present.

Time complexity: O(n\*k)

Space complexity: O(n)

class Solution {

public String longestWord(String[] words) {

String ans = "";

Set<String> wordset = new HashSet();

for (String word: words) wordset.add(word);

for (String word: words) {

if (word.length() > ans.length() ||

word.length() == ans.length() && word.compareTo(ans) < 0) {

boolean good = true;

for (int k = 1; k < word.length(); ++k) {

if (!wordset.contains(word.substring(0, k))) {

good = false;

break;

}

}

if (good) ans = word;

}

}

return ans;

}

}

class Solution {

public String longestWord(String[] words) {

Set<String> wordset = new HashSet();

for (String word: words) wordset.add(word);

Arrays.sort(words, (a, b) -> a.length() == b.length()

? a.compareTo(b) : b.length() - a.length());

for (String word: words) {

boolean good = true;

for (int k = 1; k < word.length(); ++k) {

if (!wordset.contains(word.substring(0, k))) {

good = false;

break;

}

}

if (good) return word;

}

return "";

}

}

# 721\_Accounts.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list accounts, each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name,

and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some email that

is common to both accounts. Note that even if two accounts have the same name, they may belong to different people as people

could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and

the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

Example 1:

Input:

accounts = [["John", "johnsmith@mail.com", "john00@mail.com"], ["John", "johnnybravo@mail.com"], ["John", "johnsmith@mail.com",

"john\_newyork@mail.com"], ["Mary", "mary@mail.com"]]

Output: [["John", 'john00@mail.com', 'john\_newyork@mail.com', 'johnsmith@mail.com'], ["John", "johnnybravo@mail.com"], ["Mary", "mary@mail.com"]]

Explanation:

The first and third John's are the same person as they have the common email "johnsmith@mail.com".

The second John and Mary are different people as none of their email addresses are used by other accounts.

We could return these lists in any order, for example the answer [['Mary', 'mary@mail.com'], ['John', 'johnnybravo@mail.com'],

['John', 'john00@mail.com', 'john\_newyork@mail.com', 'johnsmith@mail.com']] would still be accepted.

Note:

The length of accounts will be in the range [1, 1000].

The length of accounts[i] will be in the range [1, 10].

The length of accounts[i][j] will be in the range [1, 30].

Method 1: DFS Best solution

Intuition

Draw an edge between two emails if they occur in the same account. The problem comes down to finding connected components of this graph.

Algorithm

For each account, draw the edge from the first email to all other emails. Additionally, we'll remember a map from emails to

names on the side. After finding each connected component using a depth-first search, we'll add that to our answer.

class Solution {

public List<List<String>> accountsMerge(List<List<String>> accounts) {

Map<String, Set<String>> graph = new HashMap<>();

Map<String, String> map = new HashMap<>();

for (List<String> account : accounts){

String name = account.get(0);

for (int i = 1; i < account.size(); i++){

if (!graph.containsKey(account.get(i))){

graph.put(account.get(i), new HashSet<>());

}

map.put(account.get(i), name);

if (i == 1){

continue;

}

graph.get(account.get(i)).add(account.get(i-1));

graph.get(account.get(i-1)).add(account.get(i));

}

}

Set<String> set = new HashSet<>();

List<List<String>> res = new ArrayList<>();

for (String key : map.keySet()){

if (set.contains(key)){

continue;

}

List<String> list = new ArrayList<>();

dfs(list, set, graph, key);

Collections.sort(list);

list.add(0, map.get(key));

res.add(list);

}

return res;

}

private void dfs(List<String> list, Set<String> set, Map<String, Set<String>> graph , String key){

if (set.contains(key)){

return;

}

set.add(key);

list.add(key);

for (String nei : graph.get(key)){

dfs(list, set, graph, nei);

}

}

}

Method 2: Union Find

class Solution {

class UnionFind{

int[] parent;

public UnionFind(){

int N = 1000 \* 10;

parent = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

}

}

public int find(int x){

if (parent[x] == x){

return x;

}

return parent[x] = find(parent[x]);

}

public void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootY] = rootX;

}

}

}

public List<List<String>> accountsMerge(List<List<String>> accounts) {

List<List<String>> res = new ArrayList<>();

Map<String, String> emailToNameMap = new HashMap<>();

Map<String, Integer> emailToIDMap = new HashMap<>();

UnionFind uf = new UnionFind();

//construct union find

int id = 0;

for (List<String> account : accounts){

String name = account.get(0);

for (int i = 1; i < account.size(); i++){

if (!emailToIDMap.containsKey(account.get(i))){

emailToIDMap.put(account.get(i), id++);

}

uf.union(emailToIDMap.get(account.get(1)), emailToIDMap.get(account.get(i)));

emailToNameMap.put(account.get(i), name);

}

}

//transfer the element in union find to HashMap

Map<Integer, List<String>> graph = new HashMap<>();

for (String email : emailToNameMap.keySet()){

int p = uf.find(emailToIDMap.get(email));

if (!graph.containsKey(p)){

graph.put(p, new ArrayList<>());

}

graph.get(p).add(email);

}

for (int parent : graph.keySet()){

List<String> list = graph.get(parent);

Collections.sort(list);

String name = list.get(0);

list.add(0, emailToNameMap.get(name));

res.add(list);

}

return res;

}

}

Better UF solution

class Solution {

class UF {

int[] parent;

int[] size;

public UF (int N){

parent = new int[N];

size = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (x == parent[x]){

return x;

}

return parent[x] = find(parent[x]);

}

public void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

}

}

}

public List<List<String>> accountsMerge(List<List<String>> accounts) {

List<List<String>> res = new ArrayList<>();

if (accounts == null){

return res;

}

int n = accounts.size();

UF uf = new UF(n);

Map<String, Integer> emailToIDMap = new HashMap<>();

Map<Integer, String> idToNameMap = new HashMap<>();

for (int i = 0; i < n; i++){

List<String> account = accounts.get(i);

String name = account.get(0);

idToNameMap.put(i, name);

for (int j = 1; j < account.size(); j++){

String email = account.get(j);

if (!emailToIDMap.containsKey(email)){

emailToIDMap.put(email, i);

}else{

uf.union(i, emailToIDMap.get(email));

}

}

}

Map<Integer, List<String>> idEmailGraph = new HashMap<>();

for (String email : emailToIDMap.keySet()){

int id = emailToIDMap.get(email);

int rootId = uf.find(id);

if (!idEmailGraph.containsKey(rootId)){

idEmailGraph.put(rootId, new ArrayList<>());

}

idEmailGraph.get(rootId).add(email);

}

for (int id : idEmailGraph.keySet()){

String name = idToNameMap.get(id);

List<String> list = idEmailGraph.get(id);

Collections.sort(list);

list.add(0, name);

res.add(idEmailGraph.get(id));

}

return res;

}

}

# 722\_Remove.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a C++ program, remove comments from it. The program source is an array where source[i] is the i-th line of the source code. This represents the result of splitting the original source code string by the newline character \n.

In C++, there are two types of comments, line comments, and block comments.

The string // denotes a line comment, which represents that it and rest of the characters to the right of it in the same line should be ignored.

The string /\* denotes a block comment, which represents that all characters until the next (non-overlapping) occurrence of \*/ should be ignored. (Here, occurrences happen in reading order: line by line from left to right.) To be clear, the string /\*/ does not yet end the block comment, as the ending would be overlapping the beginning.

The first effective comment takes precedence over others: if the string // occurs in a block comment, it is ignored. Similarly, if the string /\* occurs in a line or block comment, it is also ignored.

If a certain line of code is empty after removing comments, you must not output that line: each string in the answer list will be non-empty.

There will be no control characters, single quote, or double quote characters. For example, source = "string s = "/\* Not a comment. \*/";" will not be a test case. (Also, nothing else such as defines or macros will interfere with the comments.)

It is guaranteed that every open block comment will eventually be closed, so /\* outside of a line or block comment always starts a new comment.

Finally, implicit newline characters can be deleted by block comments. Please see the examples below for details.

After removing the comments from the source code, return the source code in the same format.

Example 1:

Input:

source = ["/\*Test program \*/", "int main()", "{ ", " // variable declaration ", "int a, b, c;", "/\* This is a test", " multiline ", " comment for ", " testing \*/", "a = b + c;", "}"]

The line by line code is visualized as below:

/\*Test program \*/

int main()

{

// variable declaration

int a, b, c;

/\* This is a test

multiline

comment for

testing \*/

a = b + c;

}

Output: ["int main()","{ "," ","int a, b, c;","a = b + c;","}"]

The line by line code is visualized as below:

int main()

{

int a, b, c;

a = b + c;

}

Explanation:

The string /\* denotes a block comment, including line 1 and lines 6-9. The string // denotes line 4 as comments.

Example 2:

Input:

source = ["a/\*comment", "line", "more\_comment\*/b"]

Output: ["ab"]

Explanation: The original source string is "a/\*comment\nline\nmore\_comment\*/b", where we have bolded the newline characters. After deletion, the implicit newline characters are deleted, leaving the string "ab", which when delimited by newline characters becomes ["ab"].

Note:

The length of source is in the range [1, 100].

The length of source[i] is in the range [0, 80].

Every open block comment is eventually closed.

There are no single-quote, double-quote, or control characters in the source code.

class Solution {

public List<String> removeComments(String[] source) {

List<String> res = new ArrayList<>();

boolean comment = false;

StringBuilder sb = new StringBuilder();

for (String line : source){

for (int i = 0; i < line.length(); i++){

if (comment){

if (line.charAt(i) == '\*' && i < line.length() - 1 && line.charAt(i+1) == '/'){

comment = false;

i++;

}

}else{

if (line.charAt(i) == '/' && i < line.length() - 1 && line.charAt(i+1) == '/'){

break;

}else if (line.charAt(i) == '/' && i < line.length() - 1 && line.charAt(i+1) == '\*'){

comment = true;

i++;

}else{

sb.append(line.charAt(i));

}

}

}

if (!comment && sb.length() > 0){

res.add(sb.toString());

sb = new StringBuilder();

}

}

return res;

}

}

# 723\_Candy.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This question is about implementing a basic elimination algorithm for Candy Crush.

Given a 2D integer array board representing the grid of candy, different positive integers board[i][j] represent different types of

candies. A value of board[i][j] = 0 represents that the cell at position (i, j) is empty. The given board represents the state of

the game following the player's move. Now, you need to restore the board to a stable state by crushing candies according to the

following rules:

If three or more candies of the same type are adjacent vertically or horizontally, "crush" them all at the same time -

these positions become empty.

After crushing all candies simultaneously, if an empty space on the board has candies on top of itself, then these candies

will drop until they hit a candy or bottom at the same time. (No new candies will drop outside the top boundary.)

After the above steps, there may exist more candies that can be crushed. If so, you need to repeat the above steps.

If there does not exist more candies that can be crushed (ie. the board is stable), then return the current board.

You need to perform the above rules until the board becomes stable, then return the current board.

Example 1:

Input:

board =

[[110,5,112,113,114],[210,211,5,213,214],[310,311,3,313,314],[410,411,412,5,414],[5,1,512,3,3],[610,4,1,613,614],[710,1,2,713,714],

[810,1,2,1,1],[1,1,2,2,2],[4,1,4,4,1014]]

Output:

[[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[110,0,0,0,114],[210,0,0,0,214],[310,0,0,113,314],[410,0,0,213,414],[610,211,112,313,614],

[710,311,412,613,714],[810,411,512,713,1014]]

Explanation:

Note:

The length of board will be in the range [3, 50].

The length of board[i] will be in the range [3, 50].

Each board[i][j] will initially start as an integer in the range [1, 2000].

Time Complexity: O((R∗C)2) where R,CR, CR,C is the number of rows and columns in board. We need O(R∗C)to scan the board,

and we might crush only 3 candies repeatedly.

Space Complexity: O(1) additional complexity, as we edit the board in place.

class Solution {

public int[][] candyCrush(int[][] board) {

boolean isStable = false;

int m = board.length;

int n = board[0].length;

while (!isStable){

isStable = true;

//check row, label crushed element with negative values

for (int i = 0; i < m; i++){

for (int j = 0; j + 2 < n; j++){

int v = Math.abs(board[i][j]);

if (v != 0 && Math.abs(board[i][j+1]) == v && Math.abs(board[i][j+2]) == v){

board[i][j] = board[i][j+1] = board[i][j+2] = -v;

isStable = false;

}

}

}

//check col

for (int j = 0; j < n; j++){

for (int i = 0; i + 2 < m; i++){

int v = Math.abs(board[i][j]);

if (v != 0 && Math.abs(board[i+1][j]) == v && Math.abs(board[i+2][j]) == v){

board[i][j] = board[i+1][j] = board[i+2][j] = -v;

isStable = false;

}

}

}

//drop

for (int j = 0; j < n; j++){

int index = m - 1;

for (int i = m - 1; i >= 0; i--){

if (board[i][j] > 0){

board[index--][j] = board[i][j]; // two pointers: index is the slow pointer, i is the faster pointer

}

}

//fill zeros

while (index >= 0){

board[index--][j] = 0;

}

}

}

return board;

}

}

# 724\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers nums, write a method that returns the "pivot" index of this array.

We define the pivot index as the index where the sum of the numbers to the left of the index is equal to the sum of the numbers to the right of the index.

If no such index exists, we should return -1. If there are multiple pivot indexes, you should return the left-most pivot index.

Example 1:

Input:

nums = [1, 7, 3, 6, 5, 6]

Output: 3

Explanation:

The sum of the numbers to the left of index 3 (nums[3] = 6) is equal to the sum of numbers to the right of index 3.

Also, 3 is the first index where this occurs.

Example 2:

Input:

nums = [1, 2, 3]

Output: -1

Explanation:

There is no index that satisfies the conditions in the problem statement.

Note:

The length of nums will be in the range [0, 10000].

Each element nums[i] will be an integer in the range [-1000, 1000].

Best solution O(n)

class Solution {

public int pivotIndex(int[] nums) {

if (nums == null || nums.length == 0){

return -1;

}

int sum = 0;

for (int i = 0; i < nums.length; i++){

sum += nums[i];

}

int left = 0;

for (int i = 0; i < nums.length; i++){

int right = sum - left - nums[i];

if (left == right){

return i;

}

left += nums[i];

}

return -1;

}

}

class Solution {

public int pivotIndex(int[] nums) {

if (nums == null || nums.length == 0){

return -1;

}

int sum = 0;

for (int i = 0; i < nums.length; i++){

sum += nums[i];

}

int left = 0;

for (int i = 0; i < nums.length; i++){

if (sum - left - nums[i] == left){

return i;

}

left += nums[i];

}

return -1;

}

}

# 725\_SplitLinkedListinParts.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The length of each part should be as equal as possible: no two parts should have a size differing by more than 1.

This may lead to some parts being null.

The parts should be in order of occurrence in the input list, and parts occurring earlier should always have a size

greater than or equal parts occurring later.

Return a List of ListNode's representing the linked list parts that are formed.

Examples 1->2->3->4, k = 5 // 5 equal parts [ [1], [2], [3], [4], null ]

Example 1:

Input:

root = [1, 2, 3], k = 5

Output: [[1],[2],[3],[],[]]

Explanation:

The input and each element of the output are ListNodes, not arrays.

For example, the input root has root.val = 1, root.next.val = 2, \root.next.next.val = 3, and root.next.next.next = null.

The first element output[0] has output[0].val = 1, output[0].next = null.

The last element output[4] is null, but it's string representation as a ListNode is [].

Example 2:

Input:

root = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], k = 3

Output: [[1, 2, 3, 4], [5, 6, 7], [8, 9, 10]]

Explanation:

The input has been split into consecutive parts with size difference at most 1, and earlier parts are a larger size

than the later parts.

Note:

The length of root will be in the range [0, 1000].

Each value of a node in the input will be an integer in the range [0, 999].

k will be an integer in the range [1, 50].

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode[] splitListToParts(ListNode root, int k) {

ListNode[] result = new ListNode[k];

if (root == null){

return result;

}

ListNode curt = root;

int len = 0;

while (curt != null){

len++;

curt = curt.next;

}

int quotient = len / k;

int reminder = len - quotient \* k;

for (int i = 0; i < k; i++){

if (i < reminder){

int count = 0;

ListNode cur = root;

while (count < quotient){

cur = cur.next;

count++;

}

result[i] = root;

root = cur.next;

cur.next = null;

}else{

int count = 0;

ListNode cur = root;

while (count < quotient - 1){

cur = cur.next;

count++;

}

result[i] = root;

if (cur != null){

root = cur.next;

cur.next = null;

}

}

}

return result;

}

}

# 726\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a chemical formula (given as a string), return the count of each atom.

An atomic element always starts with an uppercase character, then zero or more lowercase letters, representing the name.

1 or more digits representing the count of that element may follow if the count is greater than 1. If the count is 1, no digits will

follow. For example, H2O and H2O2 are possible, but H1O2 is impossible.

Two formulas concatenated together produce another formula. For example, H2O2He3Mg4 is also a formula.

A formula placed in parentheses, and a count (optionally added) is also a formula. For example, (H2O2) and (H2O2)3 are formulas.

Given a formula, output the count of all elements as a string in the following form: the first name (in sorted order), followed by its

count (if that count is more than 1), followed by the second name (in sorted order), followed by its count (if that count is more than 1),

and so on.

Example 1:

Input:

formula = "H2O"

Output: "H2O"

Explanation:

The count of elements are {'H': 2, 'O': 1}.

Example 2:

Input:

formula = "Mg(OH)2"

Output: "H2MgO2"

Explanation:

The count of elements are {'H': 2, 'Mg': 1, 'O': 2}.

Example 3:

Input:

formula = "K4(ON(SO3)2)2"

Output: "K4N2O14S4"

Explanation:

The count of elements are {'K': 4, 'N': 2, 'O': 14, 'S': 4}.

Note:

All atom names consist of lowercase letters, except for the first character which is uppercase.

The length of formula will be in the range [1, 1000].

formula will only consist of letters, digits, and round parentheses, and is a valid formula as defined in the problem.

class Solution {

public String countOfAtoms(String formula) {

Map<String, Integer> res = parse(formula);

StringBuilder sb = new StringBuilder();

for (String key : res.keySet()){

sb.append(key);

int num = res.get(key);

if (num > 1){

sb.append(num);

}

}

return sb.toString();

}

private Map<String, Integer> parse(String formula) {

Map<String, Integer> map = new TreeMap<>();

int i = 0;

while (i < formula.length()) {

if (formula.charAt(i) == '(') {

int count = 0;

int j = i;

for (; j < formula.length(); j++) {//smart way to find the subproblem for brackets

if (formula.charAt(j) == '('){

count++;

}else if (formula.charAt(j) == ')'){

count--;

}

if (count == 0){

break;

}

}

Map<String, Integer> subMap = parse(formula.substring(i + 1, j));

j++;

int num = 1;

int k = j;

while (k < formula.length() && Character.isDigit(formula.charAt(k))){

k++;

}

if (k > j) {

num = Integer.parseInt(formula.substring(j, k));

}

for (String atom : subMap.keySet()) {

map.put(atom, subMap.get(atom) \* num + map.getOrDefault(atom, 0));

}

i = k;

} else {

int j = i + 1;

while (j < formula.length() && Character.isLowerCase(formula.charAt(j))){

j++;

}

int num = 1;

int k = j;

while (k < formula.length() && Character.isDigit(formula.charAt(k))){

k++;

}

if (k > j) {

num = Integer.parseInt(formula.substring(j, k));

}

String atom = formula.substring(i, j);

map.put(atom, num + map.getOrDefault(atom, 0));

i = k;

}

}

return map;

}

}

# 727\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given strings S and T, find the minimum (contiguous) substring W of S, so that T is a subsequence of W.

If there is no such window in S that covers all characters in T, return the empty string "". If there are multiple such minimum-length

windows, return the one with the left-most starting index.

Example 1:

Input:

S = "abcdebdde", T = "bde"

Output: "bcde"

Explanation:

"bcde" is the answer because it occurs before "bdde" which has the same length.

"deb" is not a smaller window because the elements of T in the window must occur in order.

Note:

All the strings in the input will only contain lowercase letters.

The length of S will be in the range [1, 20000].

The length of T will be in the range [1, 100].

Different from minmium window substring, here is the difference: T is a subsequence of W which requires to keep the order

https://github.com/optimisea/Leetcode/blob/master/Java/76\_MinimumWindowSubstring.java

Method 1: DP

Time complexity: O(mn)

Space complexity: O(mn)

class Solution {

public String minWindow(String S, String T) {

int m = T.length();

int n = S.length();

int[][] dp = new int[m+1][n+1];

// the largest starting index in String s that the first i in String T and first j in String S could match.

for (int j = 0; j <= n; j++){

dp[0][j] = j+1; //initalize by adding 1 to differentiate between index and invalid cases

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (T.charAt(i-1) == S.charAt(j-1)){

dp[i][j] = dp[i-1][j-1];

}else{

dp[i][j] = dp[i][j-1];

}

}

}

int head = 0;

int len = Integer.MAX\_VALUE;

for (int j = 1; j <= n; j++){

if (dp[m][j] != 0){

int cand = j - (dp[m][j] - 1);

if (cand < len){

head = dp[m][j] - 1;

len = cand;

}

}

}

return len == Integer.MAX\_VALUE ? "" : S.substring(head, head + len);

}

}

Best solution:

class Solution {

public String minWindow(String S, String T) {

int m = T.length();

int n = S.length();

int[][] dp = new int[m+1][n+1];

// the largest START index + 1 (or length)

//in String s that the first i (ending at i)in String T and first j (ending at j) in String S could match.

for (int i = 0; i <= m; i++){

Arrays.fill(dp[i], -1);

}

for (int j = 0; j <= n; j++){

dp[0][j] = j;

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (T.charAt(i-1) == S.charAt(j-1)){

dp[i][j] = dp[i-1][j-1];

}else{

dp[i][j] = dp[i][j-1];

}

}

}

int head = 0;

int len = Integer.MAX\_VALUE;

for (int j = 1; j <= n; j++){

if (dp[m][j] != -1){

int cand = j - dp[m][j];

if (cand < len){

head = dp[m][j];

len = cand;

}

}

}

return len == Integer.MAX\_VALUE ? "" : S.substring(head, head + len);

}

}

# 728\_Self.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A self-dividing number is a number that is divisible by every digit it contains.

For example, 128 is a self-dividing number because 128 % 1 == 0, 128 % 2 == 0, and 128 % 8 == 0.

Also, a self-dividing number is not allowed to contain the digit zero.

Given a lower and upper number bound, output a list of every possible self dividing number, including the bounds if possible.

Example 1:

Input:

left = 1, right = 22

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 22]

Note:

The boundaries of each input argument are 1 <= left <= right <= 10000.

class Solution {

public List<Integer> selfDividingNumbers(int left, int right) {

List<Integer> res = new ArrayList<>();

for (int i = left; i <= right; i++){

if (check(i)){

res.add(i);

}

}

return res;

}

private boolean check(int num){

int n = num;

while (num > 0){

int digit = num % 10;

num /= 10;

if (digit == 0 || n % digit != 0){

return false;

}

}

return true;

}

}

class Solution {

public List<Integer> selfDividingNumbers(int left, int right) {

List<Integer> list = new ArrayList<>();

for (int i = left; i <= right; i++) {

int j = i;

for (; j > 0; j /= 10) {

if ((j % 10 == 0) || (i % (j % 10) != 0)) break;

}

if (j == 0) list.add(i);

}

return list;

}

}

# 728\_ThreeDistinctFactors.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer n (1 <= n <= 10^18). Check whether a number has exactly three distinct factors,

return true if it has exactly three distinct factors, otherwise false.

Have you met this question in a real interview? Yes

Example

Given n = 9, return true

Number 9 has exactly three factors: 1, 3, 9, so return true.

Given n = 10, return false

Reference:

https://www.geeksforgeeks.org/find-divisors-natural-number-set-1/

https://stackoverflow.com/questions/9898512/how-to-test-if-a-double-is-an-integer

Method:

Time complexity: O(sqrt(n))

public class Solution {

/\*

\* @param n: the given number

\* @return: true if it has exactly three distinct factors, otherwise false

\*/

public boolean isThreeDisctFactors(long n) {

int count = 0;

for (long i = 2; i < Math.sqrt(n)+1; i++){

if (n % i == 0 && i != Math.sqrt(n)){

return false;

}

}

return true;

}

}

# 729\_My.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement a MyCalendar class to store your events. A new event can be added if adding the event will not cause a double booking.

Your class will have the method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end.

A double booking happens when two events have some non-empty intersection (ie., there is some time that is common to both events.)

For each call to the method MyCalendar.book, return true if the event can be added to the calendar successfully without causing a double booking. Otherwise, return false and do not add the event to the calendar.

Your class will be called like this: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

Example 1:

MyCalendar();

MyCalendar.book(10, 20); // returns true

MyCalendar.book(15, 25); // returns false

MyCalendar.book(20, 30); // returns true

Explanation:

The first event can be booked. The second can't because time 15 is already booked by another event.

The third event can be booked, as the first event takes every time less than 20, but not including 20.

Note:

The number of calls to MyCalendar.book per test case will be at most 1000.

In calls to MyCalendar.book(start, end), start and end are integers in the range [0, 10^9].

Method 1: Brute Force

Time complexity: O(n)

class MyCalendar {

class Interval {

int start;

int end;

public Interval (int start, int end){

this.start = start;

this.end = end;

}

}

List<Interval> intervals;

public MyCalendar() {

intervals = new ArrayList<>();

}

public boolean book(int start, int end) {

for (Interval interval : intervals){

if (interval.start < end && interval.end > start){

return false;

}

}

intervals.add(new Interval(start, end));

return true;

}

}

public class MyCalendar {

List<int[]> calendar;

MyCalendar() {

calendar = new ArrayList();

}

public boolean book(int start, int end) {

for (int[] iv: calendar) {

if (iv[0] < end && start < iv[1]) return false;

}

calendar.add(new int[]{start, end});

return true;

}

}

/\*\*

\* Your MyCalendar object will be instantiated and called as such:

\* MyCalendar obj = new MyCalendar();

\* boolean param\_1 = obj.book(start,end);

\*/

Method 2: TreeMap

Time complexity: O(logn)

class MyCalendar {

class Interval {

int start;

int end;

public Interval (int start, int end){

this.start = start;

this.end = end;

}

}

TreeMap<Integer, Interval> intervals;

public MyCalendar() {

intervals = new TreeMap<>();

}

public boolean book(int start, int end) {

if (intervals.isEmpty()){

intervals.put(start, new Interval(start, end));

return true;

}

Integer preKey = intervals.floorKey(start);

if (preKey != null && intervals.get(preKey).end > start){

return false;

}

Integer postKey = intervals.ceilingKey(start);

if (postKey != null && intervals.get(postKey).start < end){

return false;

}

intervals.put(start, new Interval(start, end));

return true;

}

}

/\*\*

\* Your MyCalendar object will be instantiated and called as such:

\* MyCalendar obj = new MyCalendar();

\* boolean param\_1 = obj.book(start,end);

\*/

Method 3: No need to create Interval class, just use treemap

Best solution

Time complexity: O(logn)

class MyCalendar {

TreeMap<Integer, Integer> intervals;

public MyCalendar() {

intervals = new TreeMap<>();

}

public boolean book(int start, int end) {

Integer preKey = intervals.floorKey(start);

if (preKey != null && intervals.get(preKey) > start){

return false;

}

Integer postKey = intervals.ceilingKey(start);

if (postKey != null && postKey < end){

return false;

}

intervals.put(start, end);

return true;

}

}

/\*\*

\* Your MyCalendar object will be instantiated and called as such:

\* MyCalendar obj = new MyCalendar();

\* boolean param\_1 = obj.book(start,end);

\*/

# 73\_SetMatrixZeroes.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a m x n matrix, if an element is 0, set its entire row and column to 0. Do it in place.

click to show follow up.

Follow up:

Did you use extra space?

A straight forward solution using O(mn) space is probably a bad idea.

A simple improvement uses O(m + n) space, but still not the best solution.

Could you devise a constant space solution?

Method 1:

space complexity: O(mn)

create a new matrix and scan the original matrix, update new matrix

Method 2:

space complexity: O(m+n)

class Solution {

public void setZeroes(int[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return;

}

int m = matrix.length;

int n = matrix[0].length;

int[] col = new int[n];

int[] row = new int[m];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (matrix[i][j] == 0){

col[j] = 1;

row[i] = 1;

}

}

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (row[i] == 1 || col[j] == 1){

matrix[i][j] = 0;

}

}

}

}

}

Method 3:

space complexity: O(1)

// fr = first row

// fc = first col

// Use first row and first column as markers.

// if matrix[i][j] = 0, mark respected row and col marker = 0; indicating

that later this respective row and col must be marked 0;

// And because you are altering first row and collumn,

you need to have two variables to track their own status.

// So, for ex, if any one of the first row is 0, fr = 0,

and at the end set all first row to 0;

class Solution {

public void setZeroes(int[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return;

}

int m = matrix.length;

int n = matrix[0].length;

boolean firstRowZero = false;

boolean firstColZero = false;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (matrix[i][j] == 0){

if (i == 0) firstRowZero = true;

if (j == 0) firstColZero = true;

matrix[i][0] = 0;

matrix[0][j] = 0;

}

}

}

for (int i = 1; i < m; i++){

for (int j = 1; j < n; j++){

if (matrix[i][0] == 0 || matrix[0][j] == 0){

matrix[i][j] = 0;

}

}

}

if (firstRowZero){

for (int i = 0; i < n; i++){

matrix[0][i] = 0;

}

}

if (firstColZero){

for (int i = 0; i < m; i++){

matrix[i][0] = 0;

}

}

}

}

# 730\_Count.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S, find the number of different non-empty palindromic subsequences in S, and return that number modulo 10^9 + 7.

A subsequence of a string S is obtained by deleting 0 or more characters from S.

A sequence is palindromic if it is equal to the sequence reversed.

Two sequences A\_1, A\_2, ... and B\_1, B\_2, ... are different if there is some i for which A\_i != B\_i.

Example 1:

Input:

S = 'bccb'

Output: 6

Explanation:

The 6 different non-empty palindromic subsequences are 'b', 'c', 'bb', 'cc', 'bcb', 'bccb'.

Note that 'bcb' is counted only once, even though it occurs twice.

Example 2:

Input:

S = 'abcdabcdabcdabcdabcdabcdabcdabcddcbadcbadcbadcbadcbadcbadcbadcba'

Output: 104860361

Explanation:

There are 3104860382 different non-empty palindromic subsequences, which is 104860361 modulo 10^9 + 7.

Note:

The length of S will be in the range [1, 1000].

Each character S[i] will be in the set {'a', 'b', 'c', 'd'}.

Intuition and Algorithm

Let dp[x][i][j] be the answer for the substring S[i...j] where S[i] == S[j] == 'a'+x. Note that since we only have 4 characters a, b, c,

d, thus 0 <= x < 4.

The DP formula goes as follows:

If S[i] != 'a'+x, then dp[x][i][j] = dp[x][i+1][j], note that here we leave the first character S[i] in the window out due

to our definition of dp[x][i][j].

If S[j] != 'a'+x, then dp[x][i][j] = dp[x][i][j-1], leaving the last character S[j] out.

If S[i] == S[j] == 'a'+x, then dp[x][i][j] = 2 + dp[0][i+1][j-1] + dp[1][i+1][j-1] + dp[2][i+1][j-1] + dp[3][i+1][j-1].

When the first and last characters are the same, we need to count all the distinct palindromes (for each of a,b,c,d) within

the sub-window S[i+1][j-1] plus the 2 palindromes contributed by the first and last characters.

Let n be the length of the input string S, The final answer would be dp[0][0][n-1] + dp[1][0][n-1] + dp[2][0][n-1] + dp[3][0][n-1]

mod 1000000007.

class Solution {

public int countPalindromicSubsequences(String S) {

int n = S.length();

int mod = 1000000007;

int[][][] dp = new int[4][n][n];

for (int i = n-1; i >= 0; --i) {

for (int j = i; j < n; ++j) {

for (int k = 0; k < 4; ++k) {

char c = (char) ('a' + k);

if (j == i) {

if (S.charAt(i) == c) dp[k][i][j] = 1;

else dp[k][i][j] = 0;

} else { // j > i

if (S.charAt(i) != c) dp[k][i][j] = dp[k][i+1][j];

else if (S.charAt(j) != c) dp[k][i][j] = dp[k][i][j-1];

else { // S[i] == S[j] == c

if (j == i+1) dp[k][i][j] = 2; // "aa" : {"a", "aa"}

else { // length is > 2

dp[k][i][j] = 2;

for (int m = 0; m < 4; ++m) { // count each one within subwindows [i+1][j-1]

dp[k][i][j] += dp[m][i+1][j-1];

dp[k][i][j] %= mod;

}

}

}

}

}

}

}

int ans = 0;

for (int k = 0; k < 4; ++k) {

ans += dp[k][0][n-1];

ans %= mod;

}

return ans;

}

}

Complexity Analysis

Time complexity : O(N^2) where N is the length of the input string SSS. It takes quadratic time to fill up the DP table.

Space complexity : O(N^2) where N is the length of the input string SSS. The DP table takes quadratic space.

Note that we ignore the constant factor 4 in the above analysis.

Conclusion

As we look back, this problem reveals a key attribute which indicates that dynamic programming might be a good fit:

overlapping sub-problems as we recall the DP formula.

https://leetcode.com/articles/count-different-palindromic-subsequences/#

https://leetcode.com/problems/count-different-palindromic-subsequences/discuss/109514/c-on2-time-on-memory-with-explanation

Best solution:

class Solution {

public int countPalindromicSubsequences(String S) {

int mod = (int)Math.pow(10, 9) + 7;

int n = S.length();

int[][][] dp = new int[4][n][n];

for (int i = n - 1; i >= 0; i--){

for (int j = i; j < n; j++){

for (int k = 0; k < 4; k++){

char c = (char)('a' + k);

if (j == i){

if (S.charAt(i) == c){

dp[k][i][j] = 1;

}else{

dp[k][i][j] = 0;

}

}else{

if (S.charAt(i) != c){

dp[k][i][j] = dp[k][i+1][j];

}else if (S.charAt(j) != c){

dp[k][i][j] = dp[k][i][j-1];

}else{

dp[k][i][j] = 2;

for (int m = 0; m < 4; m++){

dp[k][i][j] = (dp[k][i][j] + dp[m][i+1][j-1]) % mod;

}

}

}

}

}

}

int res = 0;

for (int i = 0; i < 4; i++){

res = (res + dp[i][0][n-1]) % mod;

}

return res;

}

}

# 731\_My.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement a MyCalendarTwo class to store your events. A new event can be added if adding the event will not cause a triple booking.

Your class will have one method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end.

A triple booking happens when three events have some non-empty intersection (ie., there is some time that is common to all 3 events.)

For each call to the method MyCalendar.book, return true if the event can be added to the calendar successfully without causing a triple booking. Otherwise, return false and do not add the event to the calendar.

Your class will be called like this: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

Example 1:

MyCalendar();

MyCalendar.book(10, 20); // returns true

MyCalendar.book(50, 60); // returns true

MyCalendar.book(10, 40); // returns true

MyCalendar.book(5, 15); // returns false

MyCalendar.book(5, 10); // returns true

MyCalendar.book(25, 55); // returns true

Explanation:

The first two events can be booked. The third event can be double booked.

The fourth event (5, 15) can't be booked, because it would result in a triple booking.

The fifth event (5, 10) can be booked, as it does not use time 10 which is already double booked.

The sixth event (25, 55) can be booked, as the time in [25, 40) will be double booked with the third event;

the time [40, 50) will be single booked, and the time [50, 55) will be double booked with the second event.

Note:

The number of calls to MyCalendar.book per test case will be at most 1000.

In calls to MyCalendar.book(start, end), start and end are integers in the range [0, 10^9].

Method 1: brute force

Time complexity: O(n)

class MyCalendarTwo {

class Interval {

int start;

int end;

public Interval (int start, int end){

this.start = start;

this.end = end;

}

}

List<Interval> overlap;

List<Interval> calendar;

public MyCalendarTwo() {

overlap = new ArrayList<>();

calendar = new ArrayList<>();

}

public boolean book(int start, int end) {

for (Interval interval : overlap){

if (interval.start < end && interval.end > start){

return false;

}

}

for (Interval interval : calendar){

if (interval.start < end && interval.end > start){

overlap.add(new Interval(Math.max(interval.start, start), Math.min(interval.end, end)));

}

}

calendar.add(new Interval(start, end));

return true;

}

}

/\*\*

\* Your MyCalendarTwo object will be instantiated and called as such:

\* MyCalendarTwo obj = new MyCalendarTwo();

\* boolean param\_1 = obj.book(start,end);

\*/

Method 2: sweep line (best)

Time complexity: O(n) for book

class MyCalendarTwo {

TreeMap<Integer, Integer> map;

public MyCalendarTwo() {

map = new TreeMap<>();

}

public boolean book(int start, int end) {

map.put(start, map.getOrDefault(start, 0) + 1);

map.put(end, map.getOrDefault(end, 0) - 1);

int active = 0;

for (int d : map.values()){

active += d;

if (active == 3){

map.put(start, map.get(start) - 1);

map.put(end, map.get(end) + 1);

return false;

}

}

return true;

}

}

/\*\*

\* Your MyCalendarTwo object will be instantiated and called as such:

\* MyCalendarTwo obj = new MyCalendarTwo();

\* boolean param\_1 = obj.book(start,end);

\*/

# 732\_my.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement a MyCalendarThree class to store your events. A new event can always be added.

Your class will have one method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end.

A K-booking happens when K events have some non-empty intersection (ie., there is some time that is common to all K events.)

For each call to the method MyCalendar.book, return an integer K representing the largest integer such that there exists a K-booking in the calendar.

Your class will be called like this: MyCalendarThree cal = new MyCalendarThree(); MyCalendarThree.book(start, end)

Example 1:

MyCalendarThree();

MyCalendarThree.book(10, 20); // returns 1

MyCalendarThree.book(50, 60); // returns 1

MyCalendarThree.book(10, 40); // returns 2

MyCalendarThree.book(5, 15); // returns 3

MyCalendarThree.book(5, 10); // returns 3

MyCalendarThree.book(25, 55); // returns 3

Explanation:

The first two events can be booked and are disjoint, so the maximum K-booking is a 1-booking.

The third event [10, 40) intersects the first event, and the maximum K-booking is a 2-booking.

The remaining events cause the maximum K-booking to be only a 3-booking.

Note that the last event locally causes a 2-booking, but the answer is still 3 because

eg. [10, 20), [10, 40), and [5, 15) are still triple booked.

Note:

The number of calls to MyCalendarThree.book per test case will be at most 400.

In calls to MyCalendarThree.book(start, end), start and end are integers in the range [0, 10^9]

The same as Meeting Room II

class MyCalendarThree {

TreeMap<Integer, Integer> map;

public MyCalendarThree() {

map = new TreeMap<>();

}

public int book(int start, int end) {

map.put(start, map.getOrDefault(start, 0) + 1);

map.put(end, map.getOrDefault(end, 0) - 1);

int max = 0;

int active = 0;

for (int d : map.values()){

active += d;

max = Math.max(max, active);

}

return max;

}

}

/\*\*

\* Your MyCalendarThree object will be instantiated and called as such:

\* MyCalendarThree obj = new MyCalendarThree();

\* int param\_1 = obj.book(start,end);

\*/

# 733\_FloodFill.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

An image is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, "flood fill" the image.

To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the newColor.

At the end, return the modified image.

Example 1:

Input:

image = [[1,1,1],[1,1,0],[1,0,1]]

sr = 1, sc = 1, newColor = 2

Output: [[2,2,2],[2,2,0],[2,0,1]]

Explanation:

From the center of the image (with position (sr, sc) = (1, 1)), all pixels connected

by a path of the same color as the starting pixel are colored with the new color.

Note the bottom corner is not colored 2, because it is not 4-directionally connected

to the starting pixel.

Note:

The length of image and image[0] will be in the range [1, 50].

The given starting pixel will satisfy 0 <= sr < image.length and 0 <= sc < image[0].length.

The value of each color in image[i][j] and newColor will be an integer in [0, 65535].

Method:BFS

class Solution {

class Pair{

int x;

int y;

int val;

public Pair(int x, int y, int val){

this.x = x;

this.y = y;

this.val = val;

}

}

public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {

if (image[sr][sc] == newColor){

return image;

}

Queue<Pair> queue = new LinkedList<>();

int orig = image[sr][sc];

int m = image.length;

int n = image[0].length;

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

queue.offer(new Pair(sr, sc, newColor));

image[sr][sc] = newColor;

while (!queue.isEmpty()){

Pair p = queue.poll();

for (int i = 0; i < dx.length; i++){

int cx = p.x + dx[i];

int cy = p.y + dy[i];

if (cx < m && cx >= 0 && cy < n && cy >= 0 && image[cx][cy] == orig){

image[cx][cy] = newColor;

queue.offer(new Pair(cx, cy, newColor));

}

}

}

return image;

}

}

Better:

class Solution {

public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {

if (image[sr][sc] == newColor){

return image;

}

int origColor = image[sr][sc];

int m = image.length;

int n = image[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

queue.offer(new int[]{sr, sc});

image[sr][sc] = newColor;

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && image[nx][ny] == origColor){

queue.offer(new int[]{nx, ny});

image[nx][ny] = newColor;

}

}

}

return image;

}

}

# 734\_Sentence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two sentences words1, words2 (each represented as an array of strings), and a list of similar word pairs pairs, determine if two

sentences are similar.

For example, "great acting skills" and "fine drama talent" are similar, if the similar word pairs are pairs = [["great", "fine"],

["acting","drama"], ["skills","talent"]].

Note that the similarity relation is not transitive. For example, if "great" and "fine" are similar, and "fine" and "good"

are similar, "great" and "good" are not necessarily similar.

However, similarity is symmetric. For example, "great" and "fine" being similar is the same as "fine" and "great" being similar.

Also, a word is always similar with itself. For example, the sentences words1 = ["great"], words2 = ["great"], pairs = [] are similar,

even though there are no specified similar word pairs.

Finally, sentences can only be similar if they have the same number of words. So a sentence like words1 = ["great"] can

never be similar to words2 = ["doubleplus","good"].

Note:

The length of words1 and words2 will not exceed 1000.

The length of pairs will not exceed 2000.

The length of each pairs[i] will be 2.

The length of each words[i] and pairs[i][j] will be in the range [1, 20].

Method 1:

Time complexity: O(words1.length) \* O(pairs.length) \* O(word string length for comparison)

Space complexity: O(1)

class Solution {

public boolean areSentencesSimilar(String[] words1, String[] words2, String[][] pairs) {

if (words1.length != words2.length){

return false;

}

if (pairs.length == 0){

for (int i = 0; i < words1.length; i++){

if (!words1[i].equals(words2[i])){

return false;

}

}

}

for (int i = 0; i < words1.length; i++){

if (words1[i].equals(words2[i])){

continue;

}

boolean found = false;

for (int j = 0; j < pairs.length; j++){

if (pairs[j][0].equals(words1[i]) && pairs[j][1].equals(words2[i]) || pairs[j][0].equals(words2[i]) && pairs[j][1].equals(words1[i])){

found = true;

break;

}

}

if (!found){

return false;

}

}

return true;

}

}

Method 2:

Time complexity: O(words1.length) \* O(word string length for comparison)

Space complexity: O(pairs.length)

class Solution {

public boolean areSentencesSimilar(String[] words1, String[] words2, String[][] pairs) {

if (words1.length != words2.length){

return false;

}

if (pairs.length == 0){

for (int i = 0; i < words1.length; i++){

if (!words1[i].equals(words2[i])){

return false;

}

}

}

Map<String, Set<String>> map = new HashMap<>();

for (int i = 0; i < pairs.length; i++){

if (!map.containsKey(pairs[i][0])){

map.put(pairs[i][0], new HashSet<String>());

}

if (!map.containsKey(pairs[i][1])){

map.put(pairs[i][1], new HashSet<String>());

}

map.get(pairs[i][0]).add(pairs[i][1]);

map.get(pairs[i][1]).add(pairs[i][0]);

}

for (int i = 0; i < words1.length; i++){

if (words1[i].equals(words2[i])){

continue;

}

if (map.containsKey(words1[i])){

if (map.get(words1[i]).contains(words2[i])){

continue;

}else{

return false;

}

}

// if (map.containsKey(words2[i])){

// if (map.get(words2[i]).contains(words1[i])){

// continue;

// }else{

// return false;

// }

// }

return false;

}

return true;

}

}

class Solution {

public boolean areSentencesSimilar(String[] words1, String[] words2, String[][] pairs) {

if (words1.length != words2.length){

return false;

}

if (pairs.length == 0){

for (int i = 0; i < words1.length; i++){

if (!words1[i].equals(words2[i])){

return false;

}

}

}

Map<String, Set<String>> map = new HashMap<>();

for (int i = 0; i < pairs.length; i++){

if (!map.containsKey(pairs[i][0])){

map.put(pairs[i][0], new HashSet<String>());

}

if (!map.containsKey(pairs[i][1])){

map.put(pairs[i][1], new HashSet<String>());

}

map.get(pairs[i][0]).add(pairs[i][1]);

map.get(pairs[i][1]).add(pairs[i][0]);

}

for (int i = 0; i < words1.length; i++){

if (words1[i].equals(words2[i])){

continue;

}

if (!map.containsKey(words1[i])){

return false;

}

if (!map.get(words1[i]).contains(words2[i])){

return false;

}

}

return true;

}

}

# 735\_Asteroid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given an array asteroids of integers representing asteroids in a row.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

Example 1:

Input:

asteroids = [5, 10, -5]

Output: [5, 10]

Explanation:

The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

Example 2:

Input:

asteroids = [8, -8]

Output: []

Explanation:

The 8 and -8 collide exploding each other.

Example 3:

Input:

asteroids = [10, 2, -5]

Output: [10]

Explanation:

The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

Example 4:

Input:

asteroids = [-2, -1, 1, 2]

Output: [-2, -1, 1, 2]

Explanation:

The -2 and -1 are moving left, while the 1 and 2 are moving right.

Asteroids moving the same direction never meet, so no asteroids will meet each other.

Note:

The length of asteroids will be at most 10000.

Each asteroid will be a non-zero integer in the range [-1000, 1000].

class Solution {

public int[] asteroidCollision(int[] asteroids) {

Stack<Integer> stack = new Stack<>();

for (int i : asteroids){

if (stack.isEmpty() || i \* stack.peek() > 0 || stack.peek() < 0 && i > 0){

stack.push(i);

}else if (stack.peek() > 0 && i < 0){

while (!stack.isEmpty() && stack.peek() > 0 && stack.peek() < i \* (-1)){

stack.pop();

}

if (stack.isEmpty()){

stack.push(i);

}else if (stack.peek() < 0){

stack.push(i);

}else if (stack.peek() == i \* (-1)){

stack.pop();

}

}

}

int n = stack.size();

if (n == 0){

return new int[0];

}

int[] res = new int[n];

// int index = 0; //this code also works but in reverse order

// for (int i : stack){

// res[index++] = i;

// }

int index = n - 1;

while (!stack.isEmpty()){

res[index--] = stack.pop();

}

return res;

}

}

Best solution:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int[] asteroidCollision(int[] asteroids) {

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < asteroids.length; i++){

if (stack.isEmpty() || asteroids[i] > 0){

stack.push(asteroids[i]);

}else{//asteroids[i] > 0

while (true){

int prev = stack.peek();

if (prev < 0){

stack.push(asteroids[i]);

break;

}else if (prev == -asteroids[i]){

stack.pop();

break;

}else if (prev > -asteroids[i]){

break;

}else{

stack.pop();

if (stack.isEmpty()){

stack.push(asteroids[i]);

break;

}

}

}

}

}

int n = stack.size();

int[] res = new int[n];

int index = n - 1;

while (!stack.isEmpty()){

res[index--] = stack.pop();

}

return res;

}

}

# 735\_maximumFromRight.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers, replace every element with the next greatest element

(greatest element on the right side) in the array. Since there is no element next to the last element, replace it with -1. For example,

if the array is [16, 17, 4, 3, 5, 2], then it should be modified to [17, 5, 5, 5, 2, -1].

Have you met this question in a real interview? Yes

Example

Give nums = [16, 17, 4, 3, 5, 2], change nums to [17, 5, 5, 5, 2, -1]

You should do it in place.

public class Solution {

/\*

\* @param : An array of integers.

\* @return: nothing

\*/

public void arrayReplaceWithGreatestFromRight(int[] nums) {

if (nums == null || nums.length == 0){

return;

}

int max = nums[nums.length - 1];

nums[nums.length - 1] = -1;

for (int i = nums.length - 2; i >= 0; i--){

int temp = nums[i];

nums[i] = max;

max = Math.max(max, temp);

}

}

}

# 737\_Sentence.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Based on last question I, but transitivity is allowed.

Note that the similarity relation is transitive. For example, if "great" and "good" are similar,

and "fine" and "good" are similar, then "great" and "fine" are similar.

Method 1: DFS

Time complexity: Time complexity: O(words1.length) \* O(pairs.length) \* O(word string length for comparison)

Space complexity: O(pairs.length)

class Solution {

public boolean areSentencesSimilarTwo(String[] words1, String[] words2, String[][] pairs) {

if (words1.length != words2.length){

return false;

}

if (pairs.length == 0){

for (int i = 0; i < words1.length; i++){

if (!words1[i].equals(words2[i])){

return false;

}

}

}

Map<String, Set<String>> map = new HashMap<>();

for (int i = 0; i < pairs.length; i++){

if (!map.containsKey(pairs[i][0])){

map.put(pairs[i][0], new HashSet<String>());

}

if (!map.containsKey(pairs[i][1])){

map.put(pairs[i][1], new HashSet<String>());

}

map.get(pairs[i][0]).add(pairs[i][1]);

map.get(pairs[i][1]).add(pairs[i][0]);

}

for (int i = 0; i < words1.length; i++){

if (words1[i].equals(words2[i])){

continue;

}

if (!map.containsKey(words1[i])){

return false;

}

if (!dfs(map, words1[i], words2[i], new HashSet<String>())){

return false;

}

}

return true;

}

private boolean dfs(Map<String, Set<String>> map, String source, String target, HashSet<String> visited){

if (map.get(source).contains(target)){

return true;

}

visited.add(source);

for (String str : map.get(source)){

if (!visited.contains(str) && dfs(map, str, target, visited)){

return true;

}

}

visited.remove(source);

return false;

}

}

Method 2: Union Find

Time complexity: Time complexity: O(words1.length) \* O(word string length for comparison)

Space complexity: O(pairs.length)

Time complexity with path compression: Find: O(1), Union: O(1)

class Solution {

class UF{

int[] parent;

public UF(int n){

parent = new int[n];

for (int i = 0; i < n; i++){

parent[i] = i;

}

}

public int find(int x){

if (x == parent[x]){

return x;

}

return parent[x] = find(parent[x]);

}

public void union(int x, int y){

int a = find(x);

int b = find(y);

if (a != b){

parent[a] = b;

}

}

}

public boolean areSentencesSimilarTwo(String[] words1, String[] words2, String[][] pairs) {

if (words1.length != words2.length){

return false;

}

if (pairs.length == 0){

for (int i = 0; i < words1.length; i++){

if (!words1[i].equals(words2[i])){

return false;

}

}

}

Map<String, Integer> map = new HashMap<>();

UF uf = new UF(2 \* pairs.length);

int count = 0;

for (int i = 0; i < pairs.length; i++){

if (!map.containsKey(pairs[i][0])){

map.put(pairs[i][0], count++);

}

if (!map.containsKey(pairs[i][1])){

map.put(pairs[i][1], count++);

}

uf.union(map.get(pairs[i][0]), map.get(pairs[i][1]));

}

for (int i = 0; i < words1.length; i++){

if (words1[i].equals(words2[i])){

continue;

}

if (!map.containsKey(words1[i]) || !map.containsKey(words2[i]) || uf.find(map.get(words1[i])) != uf.find(map.get(words2[i]))){

return false;

}

}

return true;

}

}

# 738\_Monotone.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-negative integer N, find the largest number that is less than or equal to N with monotone increasing digits.

(Recall that an integer has monotone increasing digits if and only if each pair of adjacent digits x and y satisfy x <= y.)

Example 1:

Input: N = 10

Output: 9

Example 2:

Input: N = 1234

Output: 1234

Example 3:

Input: N = 332

Output: 299

Note: N is an integer in the range [0, 10^9].

Intuition

One initial thought that comes to mind is we can always have a candidate answer of d999...9 (a digit 0 <= d <= 9 followed by some

number of nines.) For example if N = 432543654, we could always have an answer of at least 399999999.

We can do better. For example, when the number is 123454321, we could have a candidate of 123449999. It seems like a decent strategy

is to take a monotone increasing prefix of N, then decrease the number before the "cliff" (the index where adjacent digits decrease

for the first time) if it exists, and replace the rest of the characters with 9s.

When does that strategy fail? If N = 333222, then our strategy would give us the candidate answer of 332999 -

but this isn't monotone increasing. However, since we are looking at all indexes before the original first occurrence of a cliff,

the only place where a cliff could exist, is next to where we just decremented a digit.

Thus, we can repair our strategy, by successfully morphing our answer 332999 -> 329999 -> 299999 with a linear scan.

Algorithm

We'll find the first cliff S[i-1] > S[i]. Then, while the cliff exists, we'll decrement the appropriate digit and move i back.

Finally, we'll make the rest of the digits 9s and return our work.

We can prove our algorithm is correct because every time we encounter a cliff, the digit we decrement has to decrease by at least 1.

Then, the largest possible selection for the rest of the digits is all nines, which is always going to be monotone increasing with

respect to the other digits occurring earlier in the number.

example:

123454321 --> 123449999

332999 -> 329999 -> 299999

class Solution {

public int monotoneIncreasingDigits(int N) {

char[] S = String.valueOf(N).toCharArray();

int i = 1;

while (i < S.length && S[i] >= S[i-1]){

i++;

}

while (i > 0 && i < S.length && S[i] < S[i-1]){

i--;

S[i]--;

}

for (int j = i + 1; j < S.length; j++){

S[j] = '9';

}

return Integer.parseInt(String.valueOf(S));

}

}

Best solution:

class Solution {

public int monotoneIncreasingDigits(int N) {

String s = Integer.toString(N);

int n = s.length();

int index = -1;

for (int i = 0; i < n - 1; i++){

if (s.charAt(i) > s.charAt(i+1)){

index = i;

break;

}

}

if (index == -1){

return N;

}

while (index > 0 && s.charAt(index) == s.charAt(index-1)){ // deal with cases: 668832 should be 667999 instead of 668799 (not increasing)

index--;

}

StringBuilder sb = new StringBuilder();

for (int i = 0; i < index; i++){

sb.append(s.charAt(i));

}

sb.append((int)(s.charAt(index) - '0') - 1);

for (int i = index + 1; i < n; i++){

sb.append('9');

}

if (sb.charAt(0) == '0'){

sb.deleteCharAt(0);

}

return Integer.parseInt(sb.toString());

}

}

# 739\_Daily.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of daily temperatures, produce a list that, for each day in the input, tells you how many days you would have to wait until a warmer temperature. If there is no future day for which this is possible, put 0 instead.

For example, given the list temperatures = [73, 74, 75, 71, 69, 72, 76, 73], your output should be [1, 1, 4, 2, 1, 1, 0, 0].

Note: The length of temperatures will be in the range [1, 30000]. Each temperature will be an integer in the range [30, 100].

Similar to next greater element I

Method 1: loop from back

class Solution {

public int[] dailyTemperatures(int[] temperatures) {

Stack<Integer> stack = new Stack<>(); //montonic decreasing stack

int[] res = new int[temperatures.length];

for (int i = temperatures.length - 1; i >=0 ; i--){

while (!stack.isEmpty() && temperatures[i] >= temperatures[stack.peek()]){

stack.pop();

}

res[i] = stack.isEmpty() ? 0 : stack.peek() - i;

stack.push(i);

}

return res;

}

}

Method 2: loop from front (best)

class Solution {

public int[] dailyTemperatures(int[] temperatures) {

Stack<Integer> stack = new Stack<>(); //montonic decreasing stack

int[] res = new int[temperatures.length];

for (int i = 0; i < temperatures.length ; i++){

while (!stack.isEmpty() && temperatures[i] > temperatures[stack.peek()]){

int idx = stack.pop();

res[idx] = i - idx;

}

stack.push(i);

}

return res;

}

}

Method 3: Array stack

class Solution {

public int[] dailyTemperatures(int[] temperatures) {

int[] stack = new int[temperatures.length]; //montonic decreasing stack

int[] res = new int[temperatures.length];

int top = -1;

for (int i = 0; i < temperatures.length ; i++){

while (top > -1 && temperatures[i] > temperatures[stack[top]]){

int idx = stack[top--];

res[idx] = i - idx;

}

stack[++top] = i;

}

return res;

}

}

# 74\_FirstBadVersion.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The code base version is an integer start from 1 to n. One day, someone committed a bad version in the code case, so it caused this version and the following versions are all failed in the unit tests. Find the first bad version.

You can call isBadVersion to help you determine which version is the first bad one. The details interface can be found in the code's annotation part.

Notice

Please read the annotation in code area to get the correct way to call isBadVersion in different language. For example, Java is SVNRepo.isBadVersion(v)

Have you met this question in a real interview? Yes

Example

Given n = 5:

isBadVersion(3) -> false

isBadVersion(5) -> true

isBadVersion(4) -> true

Here we are 100% sure that the 4th version is the first bad version.

/\*\*

\* public class SVNRepo {

\* public static boolean isBadVersion(int k);

\* }

\* you can use SVNRepo.isBadVersion(k) to judge whether

\* the kth code version is bad or not.

\*/

public class Solution {

/\*

\* @param n: An integer

\* @return: An integer which is the first bad version.

\*/

public int findFirstBadVersion(int n) {

int start = 1;

int end = n;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (SVNRepo.isBadVersion(mid)){

end = mid;

}else{

start = mid;

}

}

if (SVNRepo.isBadVersion(start)){

return start;

}

return end;

}

}

/\* The isBadVersion API is defined in the parent class VersionControl.

boolean isBadVersion(int version); \*/

public class Solution extends VersionControl {

public int firstBadVersion(int n) {

int start = 1;

int end = n;

while (start <= end){

int mid = start + (end - start)/2;

if (isBadVersion(mid)){

end = mid-1;

}else{

start = mid+1;

}

}

return start;

}

}

# 74\_Search.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

Integers in each row are sorted from left to right.

The first integer of each row is greater than the last integer of the previous row.

Example 1:

Input:

matrix = [

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

target = 3

Output: true

Example 2:

Input:

matrix = [

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

target = 13

Output: false

Method: binary search

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return false;

}

int m = matrix.length;

int n = matrix[0].length;

int low = 0;

int high = m \* n - 1;

while (low <= high){

int mid = low + (high - low) / 2;

int num = converter(matrix, mid);

if (num == target){

return true;

}else if (num < target){

low = mid + 1;

}else{

high = mid - 1;

}

}

return false;

}

private int converter(int[][] matrix, int index){

int n = matrix[0].length;

int row = index / n;

int col = index % n;

return matrix[row][col];

}

}

# 74\_Search2DMatrix.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

Integers in each row are sorted from left to right.

The first integer of each row is greater than the last integer of the previous row.

For example,

Consider the following matrix:

[

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

Given target = 3, return true.

Method 1: O(logm + logn)

first binary search => target row

second binary search => target col

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return false;

}

int start = 0;

int end = matrix.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (matrix[mid][0] == target){

return true;

}else if (matrix[mid][0] > target){

end = mid;

}else{

start = mid;

}

}

int row = 0;

if (matrix[start][0] <= target){

row = start;

}

if (matrix[end][0] <= target){

row = end;

}

start = 0;

end = matrix[0].length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (matrix[row][mid] == target){

return true;

}else if (matrix[row][mid] > target){

end = mid;

}else{

start = mid;

}

}

if (matrix[row][start] == target){

return true;

}

if (matrix[row][end] == target){

return true;

}

return false;

}

}

Method 2:

Time complexity O(log(mn) = logm + logn)

// we can view this matrix as an sorted array matrix[0, m \*n - 1], given a index in array

// we can use (index / n) to get row index and use (index % n) to get column index

// then use binary search to find the target

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return false;

}

int m = matrix.length;

int n = matrix[0].length;

int start = 0;

int end = m \* n - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

int r = mid / n;

int c = mid % n;

if (matrix[r][c] == target){

return true;

}else if (matrix[r][c] < target){

start = mid;

}else{

end = mid;

}

}

if (matrix[start/n][start%n] == target || matrix[end/n][end%n] == target){

return true;

}

return false;

}

}

Better version:

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return false;

}

int m = matrix.length;

int n = matrix[0].length;

int low = 0;

int high = m \* n - 1;

while (low <= high){

int mid = low + (high - low) / 2;

int num = converter(matrix, mid);

if (num == target){

return true;

}else if (num < target){

low = mid + 1;

}else{

high = mid - 1;

}

}

return false;

}

private int converter(int[][] matrix, int index){

int n = matrix[0].length;

int row = index / n;

int col = index % n;

return matrix[row][col];

}

}

# 740\_Delete.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array nums of integers, you can perform operations on the array.

In each operation, you pick any nums[i] and delete it to earn nums[i] points. After, you must delete every element equal to

nums[i] - 1 or nums[i] + 1.

You start with 0 points. Return the maximum number of points you can earn by applying such operations.

Example 1:

Input: nums = [3, 4, 2]

Output: 6

Explanation:

Delete 4 to earn 4 points, consequently 3 is also deleted.

Then, delete 2 to earn 2 points. 6 total points are earned.

Example 2:

Input: nums = [2, 2, 3, 3, 3, 4]

Output: 9

Explanation:

Delete 3 to earn 3 points, deleting both 2's and the 4.

Then, delete 3 again to earn 3 points, and 3 again to earn 3 points.

9 total points are earned.

Note:

The length of nums is at most 20000.

Each element nums[i] is an integer in the range [1, 10000].

Similar as House Robber I

dp[i] denotes the max pointers when reach up to number i, either delete it to get dp[i-2] + i \* count[i] or keep

it to get dp[i-1]

Method 1: best solution

Time complexity: O(N)

Space complexity: O(N)

class Solution {

public int deleteAndEarn(int[] nums) {

if (nums == null || nums.length == 0){

return 0;

}

int max = 0;

for (int i : nums){

max = Math.max(max, i);

}

int[] count = new int[max+1];

for (int i : nums){

count[i]++;

}

int[] dp = new int[max+1];

dp[0] = 0;

dp[1] = 1 \* count[1];

for (int i = 2; i <= max; i++){

dp[i] = Math.max(dp[i-1], dp[i-2] + i \* count[i]);

}

return dp[max];

}

}

Better solution:

Method 2: DP + treemap

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public int deleteAndEarn(int[] nums) {

TreeMap<Integer, Integer> treemap = new TreeMap<>();

for (int i : nums){

treemap.put(i, treemap.getOrDefault(i, 0) + 1);

}

int n = treemap.size();

int[] dp = new int[n+1];

int i = 1;

for (int key : treemap.keySet()){

Integer preKey = treemap.lowerKey(key);

if (preKey == null){

dp[i] = key \* treemap.get(key);

}else{

if (preKey == key - 1){

dp[i] = Math.max(dp[i-1], dp[i-2] + key \* treemap.get(key));

}else{

dp[i] = dp[i-1] + key \* treemap.get(key);

}

}

i++;

}

return dp[n];

}

}

class Solution {

public int deleteAndEarn(int[] nums) {

TreeMap<Integer, Integer> map = new TreeMap<>();

for (int i = 0; i < nums.length; i++){

if (!map.containsKey(nums[i])){

map.put(nums[i], 1);

}else{

map.put(nums[i], map.get(nums[i]) + 1);

}

}

int n = map.size();

int[] dp = new int[n + 1];

int i = 2;

dp[0] = 0;

for (int key : map.keySet()){

if (key == map.firstKey()){

dp[1] = map.firstKey() \* map.get(map.firstKey());

}

Integer lowerKey = map.lowerKey(key);

if (lowerKey == key - 1){

dp[i] = Math.max(dp[i-2] + map.get(key) \* key, dp[i-1]);

}else{

dp[i] = dp[i-1] + map.get(key) \* key;

}

i++;

}

return dp[n];

}

}

# 742\_ClosestLeaf.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree where every node has a unique value, and a target key k, find the value of the

nearest leaf node to target k in the tree.

Here, nearest to a leaf means the least number of edges travelled on the binary tree to reach any

leaf of the tree. Also, a node is called a leaf if it has no children.

In the following examples, the input tree is represented in flattened form row by row. The actual

root tree given will be a TreeNode object.

Example 1:

Input:

root = [1, 3, 2], k = 1

Diagram of binary tree:

1

/ \

3 2

Output: 2 (or 3)

Explanation: Either 2 or 3 is the nearest leaf node to the target of 1.

Convert to undirected graph using dfs and use bfs to find the minimal path

Method 1:

Good

class Solution {

public int findClosestLeaf(TreeNode root, int k) {

Map<Integer, List<TreeNode>> graph = new HashMap<>();

buildGraph(root, null, graph);

Queue<Integer> queue = new LinkedList<>();

Set<Integer> set = new HashSet<>();

queue.offer(k);

set.add(k);

while(!queue.isEmpty()){

int curr = queue.poll();

for (TreeNode node : graph.get(curr)){

if (node.left == null && node.right == null){

return node.val;

}

if (!set.contains(node.val)){

queue.offer(node.val);

set.add(node.val);

}

}

}

return 0;

}

private void buildGraph(TreeNode root, TreeNode parent, Map<Integer, List<TreeNode>> graph){

if (root == null){

return;

}

graph.put(root.val, new ArrayList<TreeNode>());

graph.get(root.val).add(root);

if (root.left != null){

graph.get(root.val).add(root.left);

}

if (root.right != null){

graph.get(root.val).add(root.right);

}

if (parent != null){

graph.get(root.val).add(parent);

}

buildGraph(root.left, root, graph);

buildGraph(root.right, root, graph);

}

}

Method 2:

class Solution {

public int findClosestLeaf(TreeNode root, int k) {

Map<TreeNode, List<TreeNode>> graph = new HashMap();

dfs(graph, root, null);

Queue<TreeNode> queue = new LinkedList();

Set<TreeNode> seen = new HashSet();

for (TreeNode node: graph.keySet()) {

if (node != null && node.val == k) {

queue.add(node);

seen.add(node);

}

}

while (!queue.isEmpty()) {

TreeNode node = queue.poll();

if (node != null) {

if (graph.get(node).size() <= 1)

return node.val;

for (TreeNode nei: graph.get(node)) {

if (!seen.contains(nei)) {

seen.add(nei);

queue.add(nei);

}

}

}

}

throw null;

}

public void dfs(Map<TreeNode, List<TreeNode>> graph, TreeNode node, TreeNode parent) {

if (node != null) {

if (!graph.containsKey(node)) graph.put(node, new LinkedList<TreeNode>());

if (!graph.containsKey(parent)) graph.put(parent, new LinkedList<TreeNode>());

graph.get(node).add(parent);

graph.get(parent).add(node);

dfs(graph, node.left, node);

dfs(graph, node.right, node);

}

}

}

# 743\_Network.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are N network nodes, labelled 1 to N.

Given times, a list of travel times as directed edges times[i] = (u, v, w), where u is the source node, v is the target node,

and w is the time it takes for a signal to travel from source to target.

Now, we send a signal from a certain node K. How long will it take for all nodes to receive the signal? If it is impossible, return -1.

Note:

N will be in the range [1, 100].

K will be in the range [1, N].

The length of times will be in the range [1, 6000].

All edges times[i] = (u, v, w) will have 1 <= u, v <= N and 1 <= w <= 100.

https://leetcode.com/problems/network-delay-time/discuss/109968/Simple-JAVA-Djikstra's-(PriorityQueue-optimized)-Solution-with-explanation

Dijkstra’s Algorithm finds the shortest distance from the start node to every other node in the graph

Dijkstra's algorithm is based on repeatedly making the candidate move that has the least distance travelled.

In our implementations below, we showcase both O(N^2) (basic) and O(NlogN) (heap) approaches.

class Solution {

public int networkDelayTime(int[][] times, int N, int K) {

Map<Integer, Map<Integer, Integer>> graph = new HashMap<>();

for (int[] time : times){

Map<Integer, Integer> sourceMap = graph.get(time[0]);

if (sourceMap == null){

sourceMap = new HashMap<>();

graph.put(time[0], sourceMap);

}

Integer val = sourceMap.get(time[1]);

if (val == null || val > time[2]){

sourceMap.put(time[1], time[2]);

}

}

//Use PriorityQueue to get the node with shortest absolute distance

//and calculate the absolute distance of its neighbor nodes.

Map<Integer, Integer> distanceMap = new HashMap<>();//store the minimal distance from K to this point

distanceMap.put(K, 0);

PriorityQueue<int[]> pq = new PriorityQueue<>(new Comparator<int[]>(){

public int compare (int[] a, int[] b){

return a[1] - b[1];

}

});

pq.offer(new int[]{K, 0});

while (!pq.isEmpty()){

int[] cur = pq.poll();

int node = cur[0];

int distance = cur[1];

Map<Integer, Integer> sourceMap = graph.get(node);

if (sourceMap == null){

continue;

}

for (Map.Entry<Integer, Integer> entry : sourceMap.entrySet()){

int absoluteDist = distance + entry.getValue();

int targetNode = entry.getKey();

if (distanceMap.containsKey(targetNode) && distanceMap.get(targetNode) <= absoluteDist){

continue;

}

distanceMap.put(targetNode, absoluteDist);

pq.offer(new int[]{targetNode, absoluteDist});

}

}

int max = 0;

for (int val : distanceMap.values()){

max = Math.max(max, val);

}

return distanceMap.size() == N ? max : -1;

}

}

Basic:

class Solution {

Map<Integer, Integer> dist;

public int networkDelayTime(int[][] times, int N, int K) {

Map<Integer, List<int[]>> graph = new HashMap();

for (int[] edge: times) {

if (!graph.containsKey(edge[0]))

graph.put(edge[0], new ArrayList<int[]>());

graph.get(edge[0]).add(new int[]{edge[1], edge[2]});

}

dist = new HashMap();

for (int node = 1; node <= N; ++node)

dist.put(node, Integer.MAX\_VALUE);

dist.put(K, 0);

boolean[] seen = new boolean[N+1];

while (true) {

int candNode = -1;

int candDist = Integer.MAX\_VALUE;

for (int i = 1; i <= N; ++i) {

if (!seen[i] && dist.get(i) < candDist) {

candDist = dist.get(i);

candNode = i;

}

}

if (candNode < 0) break;

seen[candNode] = true;

if (graph.containsKey(candNode))

for (int[] info: graph.get(candNode))

dist.put(info[0],

Math.min(dist.get(info[0]), dist.get(candNode) + info[1]));

}

int ans = 0;

for (int cand: dist.values()) {

if (cand == Integer.MAX\_VALUE) return -1;

ans = Math.max(ans, cand);

}

return ans;

}

}

PQ:

class Solution {

public int networkDelayTime(int[][] times, int N, int K) {

Map<Integer, List<int[]>> graph = new HashMap();

for (int[] edge: times) {

if (!graph.containsKey(edge[0]))

graph.put(edge[0], new ArrayList<int[]>());

graph.get(edge[0]).add(new int[]{edge[1], edge[2]});

}

PriorityQueue<int[]> heap = new PriorityQueue<int[]>(

(info1, info2) -> info1[0] - info2[0]);

heap.offer(new int[]{0, K});

Map<Integer, Integer> dist = new HashMap();

while (!heap.isEmpty()) {

int[] info = heap.poll();

int d = info[0], node = info[1];

if (dist.containsKey(node)) continue;

dist.put(node, d);

if (graph.containsKey(node))

for (int[] edge: graph.get(node)) {

int nei = edge[0], d2 = edge[1];

if (!dist.containsKey(nei))

heap.offer(new int[]{d+d2, nei});

}

}

if (dist.size() != N) return -1;

int ans = 0;

for (int cand: dist.values())

ans = Math.max(ans, cand);

return ans;

}

}

Better version:

Time complexity: V + E

class Solution {

public int networkDelayTime(int[][] times, int N, int K) {

Map<Integer, Map<Integer, Integer>> graph = new HashMap<>();

for (int[] edge : times){

if (!graph.containsKey(edge[0])){

graph.put(edge[0], new HashMap<>());

}

graph.get(edge[0]).put(edge[1], edge[2]);

}

Map<Integer, Integer> distMap = new HashMap<>(); // store the result, key: vertex, value: shortest dist from source to key

Queue<int[]> pq = new PriorityQueue<int[]>(new Comparator<int[]>(){//e1[0]: vertex, e1[1]: shortest dist from source to key

public int compare(int[] e1, int[] e2){

return e1[1] - e2[1];

}

});

pq.offer(new int[]{K, 0});

distMap.put(K, 0);

while (!pq.isEmpty()){

int[] curr = pq.poll();

int node = curr[0];

int dist = curr[1];

if (!graph.containsKey(node)){

continue;

}

Map<Integer, Integer> map = graph.get(node);

for (int key : map.keySet()){

if (!distMap.containsKey(key) || dist + map.get(key) < distMap.get(key)){

distMap.put(key, dist + map.get(key));

pq.offer(new int[]{key, dist + map.get(key)});

}

}

}

if (distMap.size() != N){

return -1;

}

int max = 0;

for (int val : distMap.values()){

max = Math.max(max, val);

}

return max;

}

}

# 744\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of sorted characters letters containing only lowercase letters, and given a target letter target, find the smallest element in the list that is larger than the given target.

Letters also wrap around. For example, if the target is target = 'z' and letters = ['a', 'b'], the answer is 'a'.

Examples:

Input:

letters = ["c", "f", "j"]

target = "a"

Output: "c"

Input:

letters = ["c", "f", "j"]

target = "c"

Output: "f"

Input:

letters = ["c", "f", "j"]

target = "d"

Output: "f"

Input:

letters = ["c", "f", "j"]

target = "g"

Output: "j"

Input:

letters = ["c", "f", "j"]

target = "j"

Output: "c"

Input:

letters = ["c", "f", "j"]

target = "k"

Output: "c"

Note:

letters has a length in range [2, 10000].

letters consists of lowercase letters, and contains at least 2 unique letters.

target is a lowercase letter.

Method: Binary Search

class Solution {

public char nextGreatestLetter(char[] letters, char target) {

int left = 0;

int right = letters.length - 1;

while (left + 1 < right){

int mid = left + (right - left) / 2;

if (letters[mid] == target){

left = mid;

}else if (letters[mid] < target){

left = mid;

}else{

right = mid;

}

}

if (target < letters[left]){

return letters[left];

}else if (target < letters[right]){

return letters[right];

}

return right + 1 < letters.length ? letters[right+1] : letters[0];

}

}

The pattern is 0001111

binary search to find the first 1

class Solution {

public char nextGreatestLetter(char[] letters, char target) {

int n = letters.length;

int start = 0;

int end = n - 1;

while (start + 1 < end){

int mid = start + (end - start)/ 2;

char c = letters[mid];

if (c == target){

start = mid;

}else if (c > target){

end = mid;

}else{

start = mid;

}

}

if (letters[start] > target){

return letters[start];

}

if (letters[end] > target){

return letters[end];

}

return letters[0];

}

}

# 745\_Prefix.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given many words, words[i] has weight i.

Design a class WordFilter that supports one function, WordFilter.f(String prefix, String suffix). It will return the word with given prefix and suffix with maximum weight. If no word exists, return -1.

Examples:

Input:

WordFilter(["apple"])

WordFilter.f("a", "e") // returns 0

WordFilter.f("b", "") // returns -1

Note:

words has length in range [1, 15000].

For each test case, up to words.length queries WordFilter.f may be made.

words[i] has length in range [1, 10].

prefix, suffix have lengths in range [0, 10].

words[i] and prefix, suffix queries consist of lowercase letters only.

Method 1: Trie of Suffix Wrapped Words

Consider the word 'apple'. For each suffix of the word, we could insert that suffix, followed by '#', followed by the word,

all into the trie.

For example, we will insert '#apple', 'e#apple', 'le#apple', 'ple#apple', 'pple#apple', 'apple#apple' into the trie.

Then for a query like prefix = "ap", suffix = "le", we can find it by querying our trie for le#ap.

Time Complexity: O(NK^2 + QK) where N is the number of words, K is the maximum length of a word, and Q is the number of queries.

Space Complexity: O(NK^2), the size of the trie.

class WordFilter {

class TrieNode {

TrieNode[] children;

int weight;

public TrieNode (){

children = new TrieNode[27];//initialize 27 to include character '{'' which is next to 'z'

weight = 0;

}

}

TrieNode root;

public WordFilter(String[] words) {

root = new TrieNode();

for (int w = 0; w < words.length; w++){

String word = words[w] + "{";

int len = word.length();

for (int i = 0; i < len; i++){

TrieNode node = root;

for (int j = i; j < 2 \* len - 1; j++){

char c = word.charAt(j % len);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

node.weight = w;

}

}

}

}

public int f(String prefix, String suffix) {

TrieNode node = root;

String target = suffix + "{" + prefix;

for (char c : target.toCharArray()){

if (node.children[c - 'a'] == null){

return -1;

}

node = node.children[c - 'a'];

}

return node.weight;

}

}

class WordFilter {

class TrieNode {

TrieNode[] children;

int weight;

public TrieNode (){

children = new TrieNode[27];

weight = 0;

}

}

TrieNode root;

public WordFilter(String[] words) {

root = new TrieNode();

for (int w = 0; w < words.length; w++){

String word = words[w] + "{" + words[w];

int len1 = words[w].length();

int len2 = word.length();

for (int i = 0; i <= len1; i++){

TrieNode node = root;

for (int j = i; j < len2; j++){

char c = word.charAt(j);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

node.weight = w;

}

}

}

}

public int f(String prefix, String suffix) {

String query = suffix + "{" + prefix;

TrieNode node = root;

for (int i = 0; i < query.length(); i++){

char c = query.charAt(i);

if (node.children[c - 'a'] == null){

return -1;

}

node = node.children[c - 'a'];

}

return node.weight;

}

}

/\*\*

\* Your WordFilter object will be instantiated and called as such:

\* WordFilter obj = new WordFilter(words);

\* int param\_1 = obj.f(prefix,suffix);

\*/

Method 2: HashMap and others

https://leetcode.com/problems/prefix-and-suffix-search/discuss/110044/Three-ways-to-solve-this-problem-in-Java

TLE

class WordFilter {

String[] input;

public WordFilter(String[] words) {

input = words;

}

public int f(String prefix, String suffix) {

for (int i = input.length - 1; i >= 0; i--){

String word = input[i];

if (word.startsWith(prefix) && word.endsWith(suffix)){

return i;

}

}

return -1;

}

}

WordFilter: Time = O(NL)

f: Time = O(N)

Space = O(NL)

class WordFilter {

Map<String, List<Integer>> prefixMap;

Map<String, List<Integer>> suffixMap;

public WordFilter(String[] words) {

prefixMap = new HashMap<>();

suffixMap = new HashMap<>();

for (int w = 0; w < words.length; w++){

String word = words[w];

for (int i = 0; i <= word.length(); i++){

String str = word.substring(0, i);

if (!prefixMap.containsKey(str)){

prefixMap.put(str, new ArrayList<Integer>());

}

prefixMap.get(str).add(w);

}

for (int i = 0; i <= word.length(); i++){

String str = word.substring(i);

if (!suffixMap.containsKey(str)){

suffixMap.put(str, new ArrayList<Integer>());

}

suffixMap.get(str).add(w);

}

}

}

public int f(String prefix, String suffix) {

if (!prefixMap.containsKey(prefix) || !suffixMap.containsKey(suffix)){

return -1;

}

List<Integer> pList = prefixMap.get(prefix);

List<Integer> sList = suffixMap.get(suffix);

int p = pList.size() - 1;

int s = sList.size() - 1;

while (p >= 0 && s >= 0){

if (pList.get(p) > sList.get(s)){

p--;

}else if (pList.get(p) < sList.get(s)){

s--;

}else{

return pList.get(p);

}

}

return -1;

}

}

/\*\*

\* Your WordFilter object will be instantiated and called as such:

\* WordFilter obj = new WordFilter(words);

\* int param\_1 = obj.f(prefix,suffix);

\*/

# 746\_MinCostClimbingStairs.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a staircase, the i-th step has some non-negative cost cost[i] assigned (0 indexed).

Once you pay the cost, you can either climb one or two steps. You need to find minimum cost to reach the top of the floor,

and you can either start from the step with index 0, or the step with index 1.

Example 1:

Input: cost = [10, 15, 20]

Output: 15

Explanation: Cheapest is start on cost[1], pay that cost and go to the top.

Example 2:

Input: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]

Output: 6

Explanation: Cheapest is start on cost[0], and only step on 1s, skipping cost[3].

Note:

cost will have a length in the range [2, 1000].

Every cost[i] will be an integer in the range [0, 999].

To be clear, top means the index of n not n - 1;

class Solution {

public int minCostClimbingStairs(int[] cost) {

if (cost == null || cost.length == 0){

return 0;

}

int n = cost.length;

int[] dp = new int[n+1];

dp[0] = 0;

dp[1] = 0;

for (int i = 2; i <= n; i++){

dp[i] = Math.min(dp[i-1] + cost[i-1], dp[i-2] + cost[i-2]);

}

return dp[n];

}

}

747\_Largest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a given integer array nums, there is always exactly one largest element.

Find whether the largest element in the array is at least twice as much as every other number in the array.

If it is, return the index of the largest element, otherwise return -1.

Example 1:

Input: nums = [3, 6, 1, 0]

Output: 1

Explanation: 6 is the largest integer, and for every other number in the array x,

6 is more than twice as big as x. The index of value 6 is 1, so we return 1.

Example 2:

Input: nums = [1, 2, 3, 4]

Output: -1

Explanation: 4 isn't at least as big as twice the value of 3, so we return -1.

Note:

nums will have a length in the range [1, 50].

Every nums[i] will be an integer in the range [0, 99].

One pass

class Solution {

public int dominantIndex(int[] nums) {

int max = Integer.MIN\_VALUE;

int maxInd = -1;

int max2nd = Integer.MIN\_VALUE;

for (int i = 0; i < nums.length; i++){

if (max < nums[i]){

max2nd = max;

max = nums[i];

maxInd = i;

}else if (max2nd < nums[i]){

max2nd = nums[i];

}

}

if (max >= 2 \* max2nd){

return maxInd;

}

return - 1;

}

}

Two passes:

class Solution {

public int dominantIndex(int[] nums) {

int max = Integer.MIN\_VALUE;

int index = -1;

for (int i = 0; i < nums.length; i++){

if (nums[i] > max){

max = nums[i];

index = i;

}

}

for (int i : nums){

if (i == max){

continue;

}

if (max < 2 \* i){

return -1;

}

}

return index;

}

}

# 748\_Shortest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the minimum length word from a given dictionary words, which has all the letters from the string licensePlate. Such a word

is said to complete the given string licensePlate

Here, for letters we ignore case. For example, "P" on the licensePlate still matches "p" on the word.

It is guaranteed an answer exists. If there are multiple answers, return the one that occurs first in the array.

The license plate might have the same letter occurring multiple times. For example, given a licensePlate of "PP", the word "pair"

does not complete the licensePlate, but the word "supper" does.

Example 1:

Input: licensePlate = "1s3 PSt", words = ["step", "steps", "stripe", "stepple"]

Output: "steps"

Explanation: The smallest length word that contains the letters "S", "P", "S", and "T".

Note that the answer is not "step", because the letter "s" must occur in the word twice.

Also note that we ignored case for the purposes of comparing whether a letter exists in the word.

Example 2:

Input: licensePlate = "1s3 456", words = ["looks", "pest", "stew", "show"]

Output: "pest"

Explanation: There are 3 smallest length words that contains the letters "s".

We return the one that occurred first.

Note:

licensePlate will be a string with length in range [1, 7].

licensePlate will contain digits, spaces, or letters (uppercase or lowercase).

words will have a length in the range [10, 1000].

Every words[i] will consist of lowercase letters, and have length in range [1, 15].

class Solution {

public String shortestCompletingWord(String licensePlate, String[] words) {

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i < licensePlate.length(); i++){

char c = licensePlate.charAt(i);

if (Character.isLetter(c)){

char key = Character.toLowerCase(c);

map.put(key, map.getOrDefault(key, 0) + 1);

}

}

String ans = "";

int min = Integer.MAX\_VALUE;

for (String word : words){

if (word.length() < min){

boolean isValid = true;

Map<Character, Integer> wordMap = new HashMap<>();

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

wordMap.put(c, wordMap.getOrDefault(c, 0) + 1);

}

for (Character ch : map.keySet()){

if (!wordMap.containsKey(ch) || wordMap.get(ch) < map.get(ch)){

isValid = false;

break;

}

}

if (isValid){

ans = word;

min = word.length();

}

}

}

return ans;

}

}

# 75\_SortColors.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent,

with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note:

You are not suppose to use the library's sort function for this problem.

click to show follow up.

Follow up:

A rather straight forward solution is a two-pass algorithm using counting sort.

First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's,

then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

class Solution {

public void sortColors(int[] nums) {

if (nums == null || nums.length == 0){

return;

}

int left = 0;

int right = nums.length - 1;

int i = 0;

while (i <= right){

if (nums[i] == 0){

swap(nums, i, left);

i++;

left++;

}else if (nums[i] == 1){

i++;

}else{

swap(nums, i, right);

right--;

}

}

}

private void swap(int[] nums, int a, int b){

int temp = nums[a];

nums[a] = nums[b];

nums[b] = temp;

}

}

Generalized format for color from 1 to k

class Solution {

public void sortColors(int[] nums) {

int left = 0;

int right = nums.length - 1;

int i = 0;

int min = 0;

int max = 2;

while (min < max){

while (i <= right){

if (nums[i] == min){

swap(nums, i, left);

i++;

left++;

}else if (nums[i] == max){

swap(nums, i, right);

right--;

}else{

i++;

}

}

i = left;

min++;

max--;

}

}

private void swap(int[] nums, int x, int y){

int temp = nums[x];

nums[x] = nums[y];

nums[y] = temp;

}

}

# 750\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a grid where each entry is only 0 or 1, find the number of corner rectangles.

A corner rectangle is 4 distinct 1s on the grid that form an axis-aligned rectangle. Note that only the corners need to have the

value 1. Also, all four 1s used must be distinct.

Example 1:

Input: grid =

[[1, 0, 0, 1, 0],

[0, 0, 1, 0, 1],

[0, 0, 0, 1, 0],

[1, 0, 1, 0, 1]]

Output: 1

Explanation: There is only one corner rectangle, with corners grid[1][2], grid[1][4], grid[3][2], grid[3][4].

Example 2:

Input: grid =

[[1, 1, 1],

[1, 1, 1],

[1, 1, 1]]

Output: 9

Explanation: There are four 2x2 rectangles, four 2x3 and 3x2 rectangles, and one 3x3 rectangle.

Example 3:

Input: grid =

[[1, 1, 1, 1]]

Output: 0

Explanation: Rectangles must have four distinct corners.

Note:

The number of rows and columns of grid will each be in the range [1, 200].

Each grid[i][j] will be either 0 or 1.

The number of 1s in the grid will be at most 6000.

Method : This is a good example of how to use HashMap to record row/column index posiiton in matrix.

Intuition

We ask the question: for each additional row, how many more rectangles are added?

For each pair of 1s in the new row (say at new\_row[i] and new\_row[j]), we could create more rectangles where that pair forms the base.

The number of new rectangles is the number of times some previous row had row[i] = row[j] = 1.

Algorithm

Let's maintain a count count[i, j], the number of times we saw row[i] = row[j] = 1. When we process a new row, for every pair

new\_row[i] = new\_row[j] = 1, we add count[i, j] to the answer, then we increment count[i, j].

Time Complexity: O(R\*C\*C)O where R,CR, CR,C is the number of rows and columns.

Space Complexity: O(C^2) in additional space.

class Solution {

public int countCornerRectangles(int[][] grid) {

Map<Integer, Integer> map = new HashMap<>();

int ans = 0;

for (int[] row : grid){

for (int c1 = 0; c1 < row.length - 1; c1++){

if (row[c1] == 1){

for (int c2 = c1 + 1; c2 < row.length; c2++){

if (row[c2] == 1){

int pos = c1 \* row.length + c2;

int count = map.getOrDefault(pos, 0);

ans += count;

map.put(pos, count+1);

}

}

}

}

}

return ans;

}

}

# 752\_Open.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels can rotate freely and wrap around: for example we can turn '9' to be '0', or '0' to be '9'. Each move consists of turning one wheel one slot.

The lock initially starts at '0000', a string representing the state of the 4 wheels.

You are given a list of deadends dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.

Given a target representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

Example 1:

Input: deadends = ["0201","0101","0102","1212","2002"], target = "0202"

Output: 6

Explanation:

A sequence of valid moves would be "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202".

Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" -> "0202" would be invalid,

because the wheels of the lock become stuck after the display becomes the dead end "0102".

Example 2:

Input: deadends = ["8888"], target = "0009"

Output: 1

Explanation:

We can turn the last wheel in reverse to move from "0000" -> "0009".

Example 3:

Input: deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], target = "8888"

Output: -1

Explanation:

We can't reach the target without getting stuck.

Example 4:

Input: deadends = ["0000"], target = "8888"

Output: -1

Note:

The length of deadends will be in the range [1, 500].

target will not be in the list deadends.

Every string in deadends and the string target will be a string of 4 digits from the 10,000 possibilities '0000' to '9999'

Method : BFS

Best solution

class Solution {

public int openLock(String[] deadends, String target) {

Set<String> set = new HashSet<>();

for (String s : deadends){

set.add(s);

}

if (set.contains("0000")){

return -1;

}

Queue<String> queue = new LinkedList<>();

Set<String> visited = new HashSet<>();

queue.offer("0000");

visited.add("0000");

int level = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int k = 0; k < size; k++){

String curr = queue.poll();

if (target.equals(curr)){

return level;

}

for (int i = 0; i < 4; i++){

for (int j = -1; j <= 1; j+= 2){

int d = ((curr.charAt(i) - '0') + j + 10) % 10;

String s = curr.substring(0, i) + d + curr.substring(i+1);

if (!visited.contains(s) && !set.contains(s)){

visited.add(s);

queue.offer(s);

}

}

}

}

level++;

}

return -1;

}

}

class Solution {

public int openLock(String[] deadends, String target) {

Set<String> dead = new HashSet();

for (String d: deadends) dead.add(d);

Queue<String> queue = new LinkedList();

queue.offer("0000");

queue.offer(null);

Set<String> seen = new HashSet();

seen.add("0000");

int depth = 0;

while (!queue.isEmpty()) {

String node = queue.poll();

if (node == null) {

depth++;

if (queue.peek() != null)

queue.offer(null);

} else if (node.equals(target)) {

return depth;

} else if (!dead.contains(node)) {

for (int i = 0; i < 4; ++i) {

for (int d = -1; d <= 1; d += 2) {

int y = ((node.charAt(i) - '0') + d + 10) % 10;

String nei = node.substring(0, i) + ("" + y) + node.substring(i+1);

if (!seen.contains(nei)) {

seen.add(nei);

queue.offer(nei);

}

}

}

}

}

return -1;

}

}

class Solution {

public int openLock(String[] deadends, String target) {

Set<String> set = new HashSet<>();

for (String s : deadends){

set.add(s);

}

if (set.contains("0000")){

return -1;

}

Queue<String> queue = new LinkedList<>();

Set<String> visited = new HashSet<>();

queue.offer("0000");

visited.add("0000");

int[] d = {-1000, -100, -10, -1, 1000, 100, 10, 1};

int level = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int k = 0; k < size; k++){

String curr = queue.poll();

if (target.equals(curr)){

return level;

}

for (int i = 0; i < d.length; i++){

int currNum = Integer.parseInt(curr);

int newNum = currNum;

if (i < 4 && curr.charAt(i) == '0'){

newNum += -9 \* d[i];

}else if (i >= 4 && curr.charAt(i-4) == '9'){

newNum += - 9 \* d[i];

}else{

newNum += d[i];

}

String str = String.format("%04d", newNum);

if (!visited.contains(str) && !set.contains(str)){

queue.offer(str);

visited.add(str);

}

}

}

level++;

}

return -1;

}

}

for(int i = 0; i < 4; i ++) {

char c = sb.charAt(i);

String s1 = sb.substring(0, i) + (c == '9' ? 0 : c - '0' + 1) + sb.substring(i + 1);

String s2 = sb.substring(0, i) + (c == '0' ? 9 : c - '0' - 1) + sb.substring(i + 1);

if(!visited.contains(s1) && !deads.contains(s1)) {

q.offer(s1);

visited.add(s1);

}

if(!visited.contains(s2) && !deads.contains(s2)) {

q.offer(s2);

visited.add(s2);

}

}

Best solution:

class Solution {

public int openLock(String[] deadends, String target) {

Set<String> dead = new HashSet<>();

for (String s : deadends){

dead.add(s);

}

if (dead.contains("0000")){

return -1;

}

Queue<String> queue = new LinkedList<>();

Set<String> seen = new HashSet<>();

queue.offer("0000");

seen.add("0000");

int res = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

String str = queue.poll();

if (str.equals(target)){

return res;

}

for (int j = 0; j < 4; j++){

int num = (int)(str.charAt(j) - '0');

String cand = str.substring(0, j) + ((num+1)%10) + str.substring(j+1);

if (!seen.contains(cand) && !dead.contains(cand)){

queue.offer(cand);

seen.add(cand);

}

cand = str.substring(0, j) + ((num+9)%10) + str.substring(j+1);

if (!seen.contains(cand) && !dead.contains(cand)){

queue.offer(cand);

seen.add(cand);

}

}

}

res++;

}

return -1;

}

}

# 753\_Cracking.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a box protected by a password. The password is n digits, where each letter can be one of the first k digits 0, 1, ..., k-1.

You can keep inputting the password, the password will automatically be matched against the last n digits entered.

For example, assuming the password is "345", I can open it when I type "012345", but I enter a total of 6 digits.

Please return any string of minimum length that is guaranteed to open the box after the entire string is inputted.

Example 1:

Input: n = 1, k = 2

Output: "01"

Note: "10" will be accepted too.

Example 2:

Input: n = 2, k = 2

Output: "00110"

Note: "01100", "10011", "11001" will be accepted too.

Note:

n will be in the range [1, 4].

k will be in the range [1, 10].

k^n will be at most 4096.

We can think of this problem as the problem of finding an Euler path (a path visiting every edge exactly once) on the following

graph: there are $$k^{n-1}$$ nodes with each node having $$k$$ edges. It turns out this graph always has an Eulerian circuit

(path starting where it ends.) We should visit each node in "post-order" so as to not get stuck in the graph prematurely.

Backtracking

class Solution {

public String crackSafe(int n, int k) {

StringBuilder sb = new StringBuilder();

int total = (int) Math.pow(k, n);

Set<String> seen = new HashSet<>();

for (int i = 0; i < n; i++){

sb.append("0");

}

seen.add(sb.toString());

dfs(sb, seen, total, k, n);

return sb.toString();

}

private boolean dfs(StringBuilder sb, Set<String> seen, int total, int k, int n){

if (seen.size() == total){

return true;

}

String prev = sb.substring(sb.length() - n + 1);

for (int i = 0; i < k; i++){

String next = prev + i;

if (!seen.contains(next)){

seen.add(next);

sb.append(i);

if (dfs(sb,seen, total, k, n)){

return true;

}else{

sb.deleteCharAt(sb.length() - 1);

seen.remove(next);

}

}

}

return false;

}

}

# 754\_Reach.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are standing at position 0 on an infinite number line. There is a goal at position target.

On each move, you can either go left or right. During the n-th move (starting from 1), you take n steps.

Return the minimum number of steps required to reach the destination.

Example 1:

Input: target = 3

Output: 2

Explanation:

On the first move we step from 0 to 1.

On the second step we step from 1 to 3.

Example 2:

Input: target = 2

Output: 3

Explanation:

On the first move we step from 0 to 1.

On the second move we step from 1 to -1.

On the third move we step from -1 to 2.

Note:

target will be a non-zero integer in the range [-10^9, 10^9].

Method 1: BFS TLE

class Solution {

public int reachNumber(int target) {

int count = 0;

Queue<Integer> queue = new LinkedList<>();

Set<Integer> seen = new HashSet<>();

queue.offer(0);

while (!queue.isEmpty()){

int size = queue.size();

count++;

for (int i = 0; i < size; i++){

int curr = queue.poll();

int left = curr - count;

int right = curr + count;

if (left == target || right == target){

return count;

}

queue.offer(left);

queue.offer(right);

}

}

return -1;

}

}

Method 2: Math

https://leetcode.com/problems/reach-a-number/discuss/112968/Short-JAVA-Solution-with-Explanation

class Solution {

public int reachNumber(int target) {

target = Math.abs(target);

int step = 0;

int sum = 0;

while (target > sum){

step++;

sum += step;

}

while ((target - sum) % 2 != 0){

step++;

sum += step;

}

return step;

}

}

class Solution {

public int reachNumber(int target) {

target = Math.abs(target);

int step = 0;

while (target > 0){

step++;

target -= step;

}

return target % 2 == 0 ? step : step + 1 + step % 2;

}

}

# 755\_Pour.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given an elevation map, heights[i] representing the height of the terrain at that index. The width at each index is 1.

After V units of water fall at index K, how much water is at each index?

Water first drops at index K and rests on top of the highest terrain or water at that index. Then, it flows according to the

following rules:

If the droplet would eventually fall by moving left, then move left.

Otherwise, if the droplet would eventually fall by moving right, then move right.

Otherwise, rise at it's current position.

Here, "eventually fall" means that the droplet will eventually be at a lower level if it moves in that direction.

Also, "level" means the height of the terrain plus any water in that column.

We can assume there's infinitely high terrain on the two sides out of bounds of the array. Also, there could not be partial

water being spread out evenly on more than 1 grid block - each unit of water has to be in exactly one block.

Example 1:

Input: heights = [2,1,1,2,1,2,2], V = 4, K = 3

Output: [2,2,2,3,2,2,2]

Explanation:

# #

# #

## # ###

#########

0123456 <- index

The first drop of water lands at index K = 3:

# #

# w #

## # ###

#########

0123456

When moving left or right, the water can only move to the same level or a lower level.

(By level, we mean the total height of the terrain plus any water in that column.)

Since moving left will eventually make it fall, it moves left.

(A droplet "made to fall" means go to a lower height than it was at previously.)

# #

# #

## w# ###

#########

0123456

Since moving left will not make it fall, it stays in place. The next droplet falls:

# #

# w #

## w# ###

#########

0123456

Since the new droplet moving left will eventually make it fall, it moves left.

Notice that the droplet still preferred to move left,

even though it could move right (and moving right makes it fall quicker.)

# #

# w #

## w# ###

#########

0123456

# #

# #

##ww# ###

#########

0123456

After those steps, the third droplet falls.

Since moving left would not eventually make it fall, it tries to move right.

Since moving right would eventually make it fall, it moves right.

# #

# w #

##ww# ###

#########

0123456

# #

# #

##ww#w###

#########

0123456

Finally, the fourth droplet falls.

Since moving left would not eventually make it fall, it tries to move right.

Since moving right would not eventually make it fall, it stays in place:

# #

# w #

##ww#w###

#########

0123456

The final answer is [2,2,2,3,2,2,2]:

#

#######

#######

0123456

Example 2:

Input: heights = [1,2,3,4], V = 2, K = 2

Output: [2,3,3,4]

Explanation:

The last droplet settles at index 1, since moving further left would not cause it to eventually fall to a lower height.

Example 3:

Input: heights = [3,1,3], V = 5, K = 1

Output: [4,4,4]

Note:

heights will have length in [1, 100] and contain integers in [0, 99].

V will be in range [0, 2000].

K will be in range [0, heights.length - 1].

class Solution {

public int[] pourWater(int[] heights, int V, int K) {

int n = heights.length;

int[] res = new int[n];

for (int i = 0; i < n; i++){

res[i] = heights[i];

}

for (int i = 0; i < V; i++){

int lowest = K;

for (int j = K - 1; j >= 0; j--){

if (res[j] < res[lowest]){

lowest = j;

}else if (res[j] > res[lowest]){

break;

}

}

if (lowest != K){

res[lowest]++;

}else{

for (int j = K + 1; j < n; j++){

if (res[j] < res[lowest]){

lowest = j;

}else if (res[j] > res[lowest]){

break;

}

}

res[lowest]++;

}

}

return res;

}

}

# 756\_Pyramid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are stacking blocks to form a pyramid. Each block has a color which is a one letter string, like `'Z'`.

For every block of color `C` we place not in the bottom row, we are placing it on top of a left block of color `A` and right block

of color `B`. We are allowed to place the block there only if `(A, B, C)` is an allowed triple.

We start with a bottom row of bottom, represented as a single string. We also start with a list of allowed triples allowed.

Each allowed triple is represented as a string of length 3.

Return true if we can build the pyramid all the way to the top, otherwise false.

Example 1:

Input: bottom = "XYZ", allowed = ["XYD", "YZE", "DEA", "FFF"]

Output: true

Explanation:

We can stack the pyramid like this:

A

/ \

D E

/ \ / \

X Y Z

This works because ('X', 'Y', 'D'), ('Y', 'Z', 'E'), and ('D', 'E', 'A') are allowed triples.

Example 2:

Input: bottom = "XXYX", allowed = ["XXX", "XXY", "XYX", "XYY", "YXZ"]

Output: false

Explanation:

We can't stack the pyramid to the top.

Note that there could be allowed triples (A, B, C) and (A, B, D) with C != D.

Note:

bottom will be a string with length in range [2, 8].

allowed will have length in range [0, 200].

Letters in all strings will be chosen from the set {'A', 'B', 'C', 'D', 'E', 'F', 'G'}.

Method: DFS + backtracking

class Solution {

public boolean pyramidTransition(String bottom, List<String> allowed) {

Map<String, List<String>> map = new HashMap<>();

for (String str : allowed){

String key = str.substring(0,2);

if (!map.containsKey(key)){

map.put(key, new ArrayList<>());

}

map.get(key).add(str.substring(2));

}

return dfs(bottom, map);

}

private boolean dfs(String bottom, Map<String, List<String>> map){

if (bottom.length() == 1){

return true;

}

for (int i = 0; i < bottom.length() - 1; i++){

String prefix = bottom.substring(i, i+2);

if (!map.containsKey(prefix)){

return false;

}

}

List<String> list = new ArrayList<>();

getList(bottom, map, list, 0, new StringBuilder());

for (String str : list){

if (dfs(str, map)){

return true;

}

}

return false;

}

private void getList(String bottom, Map<String, List<String>> map, List<String> list, int start, StringBuilder sb){

if (start == bottom.length() - 1){

list.add(sb.toString());

return;

}

List<String> temp = map.get(bottom.substring(start, start+2));

for (String s : temp){

sb.append(s);

getList(bottom, map, list, start + 1, sb);

sb.deleteCharAt(sb.length() - 1);

}

}

}

Better version:

class Solution {

public boolean pyramidTransition(String bottom, List<String> allowed) {

Map<String, Set<String>> map = new HashMap<>();

for (String s : allowed){

String key = s.substring(0, 2);

String val = s.substring(2);

if (!map.containsKey(key)){

map.put(key, new HashSet<>());

}

map.get(key).add(val);

}

return dfs(bottom, map);

}

private boolean dfs(String target, Map<String, Set<String>> map){

if (target.length() == 1){

return true;

}

for (int i = 0; i < target.length() - 1; i++){

String sub = target.substring(i, i+2);

if (!map.containsKey(sub)){

return false;

}

}

StringBuilder sb = new StringBuilder();

List<String> list = new ArrayList<>();

backtrack(list, sb, map, target, 0);

for (String str : list){

if (dfs(str, map)){

return true;

}

}

return false;

}

private void backtrack(List<String> list, StringBuilder sb, Map<String, Set<String>> map, String target, int start){

if (sb.length() == target.length() - 1){

list.add(sb.toString());

return;

}

Set<String> set = map.get(target.substring(start, start+2));

for (String key : set){

sb.append(key);

backtrack(list, sb, map, target, start+1);

sb.deleteCharAt(sb.length() - 1);

}

}

}

# 758\_Bold.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a set of keywords words and a string S, make all appearances of all keywords in S bold. Any letters between <b> and </b>

tags become bold.

The returned string should use the least number of tags possible, and of course the tags should form a valid combination.

For example, given that words = ["ab", "bc"] and S = "aabcd", we should return "a<b>abc</b>d". Note that returning

"a<b>a<b>b</b>c</b>d" would use more tags, so it is incorrect.

Note:

words has length in range [0, 50].

words[i] has length in range [1, 10].

S has length in range [0, 500].

All characters in words[i] and S are lowercase letters.

Method 1: the same as 616. Add Bold Tag in a String

class Solution {

class Interval{

int start;

int end;

Interval(int start, int end){

this.start = start;

this.end = end;

}

public String toString() {

return "[" + start + ", " + end + "]" ;

}

}

public List<Interval> merge(List<Interval> intervals){

List<Interval> result = new ArrayList<>();

if (intervals == null || intervals.size() == 0){

return result;

}

Collections.sort(intervals, new Comparator<Interval>(){

public int compare (Interval a, Interval b){

if (a.start == b.start){

return a.end - b.end;

}

return a.start - b.start;

}

});

Interval last = null;

for (Interval interval : intervals){

if (last == null || last.end < interval.start){

result.add(interval);

last = interval;

}else{

last.end = Math.max(last.end, interval.end);

}

}

return result;

}

public String boldWords(String[] words, String S) {

List<Interval> intervals = new ArrayList<>();

for (String str : words){

int index = -1;

index = S.indexOf(str, index);

while (index != -1){

intervals.add(new Interval(index, index + str.length()));

index++;

index = S.indexOf(str, index);

}

}

intervals = merge(intervals);

StringBuilder sb = new StringBuilder();

int prev = 0;

for (Interval interval : intervals){

sb.append(S.substring(prev, interval.start));

sb.append("<b>");

sb.append(S.substring(interval.start, interval.end));

sb.append("</b>");

prev = interval.end;

}

if (prev < S.length()){

sb.append(S.substring(prev));

}

return sb.toString();

}

}

# 759\_Employee.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given a list schedule of employees, which represents the working time for each employee.

Each employee has a list of non-overlapping Intervals, and these intervals are in sorted order.

Return the list of finite intervals representing common, positive-length free time for all employees, also in sorted order.

Example 1:

Input: schedule = [[[1,2],[5,6]],[[1,3]],[[4,10]]]

Output: [[3,4]]

Explanation:

There are a total of three employees, and all common

free time intervals would be [-inf, 1], [3, 4], [10, inf].

We discard any intervals that contain inf as they aren't finite.

Example 2:

Input: schedule = [[[1,3],[6,7]],[[2,4]],[[2,5],[9,12]]]

Output: [[5,6],[7,9]]

(Even though we are representing Intervals in the form [x, y], the objects inside are Intervals, not lists or arrays. For example, schedule[0][0].start = 1, schedule[0][0].end = 2, and schedule[0][0][0] is not defined.)

Also, we wouldn't include intervals like [5, 5] in our answer, as they have zero length.

Note:

schedule and schedule[i] are lists with lengths in range [1, 50].

0 <= schedule[i].start < schedule[i].end <= 10^8.

Method: Similar as Merge Interval

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public List<Interval> employeeFreeTime(List<List<Interval>> schedule) {

List<Interval> list = new ArrayList<>();

for (List<Interval> ls : schedule){

for (Interval i : ls){

list.add(i);

}

}

//merge interval

Collections.sort(list, new Comparator<Interval>(){

public int compare (Interval a, Interval b){

return a.start - b.start;

}

});

List<Interval> mergeList = new ArrayList<>();

Interval last = null;

for (Interval item : list){

if (last == null || last.end < item.start){

mergeList.add(item);

last = item;

}else{

last.end = Math.max(last.end, item.end);

}

}

///

List<Interval> res = new ArrayList<>();

for (int i = 0; i < mergeList.size() - 1; i++){

int start = mergeList.get(i).end;

int end = mergeList.get(i+1).start;

res.add(new Interval(start, end));

}

return res;

}

}

# 760\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two lists Aand B, and B is an anagram of A. B is an anagram of A means B is made by randomizing the order of the elements in A.

We want to find an index mapping P, from A to B. A mapping P[i] = j means the ith element in A appears in B at index j.

These lists A and B may contain duplicates. If there are multiple answers, output any of them.

For example, given

A = [12, 28, 46, 32, 50]

B = [50, 12, 32, 46, 28]

We should return

[1, 4, 3, 2, 0]

as P[0] = 1 because the 0th element of A appears at B[1], and P[1] = 4 because the 1st element of A appears at B[4], and so on.

Note:

A, B have equal lengths in range [1, 100].

A[i], B[i] are integers in range [0, 10^5].

Method 1:

Time complexity: O(n^2)

Space complexity: O(1)

class Solution {

public int[] anagramMappings(int[] A, int[] B) {

int[] res = new int[A.length];

for (int i = 0; i < A.length; i++){

for (int j = 0; j < B.length; j++){

if (A[i] == B[j]){

res[i] = j;

}

}

}

return res;

}

}

Method 2:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int[] anagramMappings(int[] A, int[] B) {

int[] res = new int[A.length];

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < B.length; i++){

map.put(B[i], i);

}

for (int i = 0; i < A.length; i++){

res[i] = map.get(A[i]);

}

return res;

}

}

761\_Special.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Special binary strings are binary strings with the following two properties:

The number of 0's is equal to the number of 1's.

Every prefix of the binary string has at least as many 1's as 0's.

Given a special string S, a move consists of choosing two consecutive, non-empty, special substrings of S, and swapping them.

(Two strings are consecutive if the last character of the first string is exactly one index before the first character of the

second string.)

At the end of any number of moves, what is the lexicographically largest resulting string possible?

Example 1:

Input: S = "11011000"

Output: "11100100"

Explanation:

The strings "10" [occuring at S[1]] and "1100" [at S[3]] are swapped.

This is the lexicographically largest string possible after some number of swaps.

Note:

S has length at most 50.

S is guaranteed to be a special binary string as defined above.

Recursion

Similar as Valid Parentheses string and number of atoms

https://leetcode.com/problems/special-binary-string/discuss/113211/Easy-and-Concise-Solution-with-Explanation-C++JavaPython

https://leetcode.com/problems/special-binary-string/discuss/113212/Think-of-it-as-Valid-Parentheses

class Solution {

public String makeLargestSpecial(String S) {

int i = 0;

int count = 0;

List<String> res = new ArrayList<>();

for (int j = 0; j < S.length(); j++){

if (S.charAt(j) == '1'){

count++;

}else{

count--;

}

if (count == 0){

res.add('1' + makeLargestSpecial(S.substring(i+1, j)) + '0');

i = j + 1;

}

}

Collections.sort(res, Collections.reverseOrder());

return String.join("", res);

}

}

763\_PartitionLabels.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A string S of lowercase letters is given. We want to partition this string into as many parts as

possible so that each letter appears in at most one part, and return a list of integers representing the size of these parts.

Example 1:

Input: S = "ababcbacadefegdehijhklij"

Output: [9,7,8]

Explanation:

The partition is "ababcbaca", "defegde", "hijhklij".

This is a partition so that each letter appears in at most one part.

A partition like "ababcbacadefegde", "hijhklij" is incorrect, because it splits S into less parts.

Note:

S will have length in range [1, 500].

S will consist of lowercase letters ('a' to 'z') only.

Method: two points

Time complexity: O(n)

class Solution {

public List<Integer> partitionLabels(String S) {

List<Integer> result = new ArrayList<>();

if (S == null || S.length() == 0){

return result;

}

Map<Character, Integer> map = new HashMap<>(); // Integer stores the last index

for (int i = 0; i < S.length(); i++){

map.put(S.charAt(i), i);

}

int start = 0;

int end = 0;

int cutIndex = 0;

while (end < S.length()){

char c = S.charAt(end);

cutIndex = map.get(c);

while (end < cutIndex){

end++;

if (end < S.length() && map.get(S.charAt(end)) > cutIndex){

cutIndex = map.get(S.charAt(end));

}

}

result.add(cutIndex - start + 1);

start = cutIndex + 1;

end = start;

}

return result;

}

}

Better version:

class Solution {

public List<Integer> partitionLabels(String S) {

List<Integer> res = new ArrayList<>();

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i < S.length(); i++){

map.put(S.charAt(i), i);

}

int start = 0;

int end = 0;

int i = 0;

while (i < S.length()){

char c = S.charAt(i);

end = map.get(c);

while (i < end && map.get(S.charAt(i)) <= end){

i++;

}

if (i == end){

res.add(end - start + 1);

start = end + 1;

i = start;

}

}

return res;

}

}

764\_Largest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a 2D grid from (0, 0) to (N-1, N-1), every cell contains a 1, except those cells in the given list mines which are 0.

What is the largest axis-aligned plus sign of 1s contained in the grid? Return the order of the plus sign. If there is none, return 0.

An "axis-aligned plus sign of 1s of order k" has some center grid[x][y] = 1 along with 4 arms of length k-1 going up, down,

left, and right, and made of 1s. This is demonstrated in the diagrams below. Note that there could be 0s or 1s beyond the

arms of the plus sign, only the relevant area of the plus sign is checked for 1s.

Examples of Axis-Aligned Plus Signs of Order k:

Order 1:

000

010

000

Order 2:

00000

00100

01110

00100

00000

Order 3:

0000000

0001000

0001000

0111110

0001000

0001000

0000000

Example 1:

Input: N = 5, mines = [[4, 2]]

Output: 2

Explanation:

11111

11111

11111

11111

11011

In the above grid, the largest plus sign can only be order 2. One of them is marked in bold.

Example 2:

Input: N = 2, mines = []

Output: 1

Explanation:

There is no plus sign of order 2, but there is of order 1.

Example 3:

Input: N = 1, mines = [[0, 0]]

Output: 0

Explanation:

There is no plus sign, so return 0.

Note:

N will be an integer in the range [1, 500].

mines will have length at most 5000.

mines[i] will be length 2 and consist of integers in the range [0, N-1].

Method 1: Brute Force

Time complexity: O(N^3)

class Solution {

public int orderOfLargestPlusSign(int N, int[][] mines) {

Set<Integer> set = new HashSet<>();

for (int[] mine : mines){

set.add(mine[0] \* N + mine[1]);

}

int max = 0;

for (int i = 0; i < N; i++){

for (int j = 0; j < N; j++){

int k = 0;

while (k <= i && k + i < N && k <= j && k + j < N &&

!set.contains((i-k) \* N + j) && !set.contains((i+k) \* N + j) &&

!set.contains(i \* N + j - k) && !set.contains(i \* N + j + k)){

k++;

}

max = Math.max(max, k);

}

}

return max;

}

}

Method 2: DP

Time complexity: O(N^2)

class Solution {

public int orderOfLargestPlusSign(int N, int[][] mines) {

int max = 0;

int[][] dp = new int[N][N];//max largest plus sign at the center of (i, j)

for (int i = 0; i < N; i++){

Arrays.fill(dp[i], N);

}

Set<Integer> set = new HashSet<>();

for (int[] mine : mines){

set.add(mine[0] \* N + mine[1]);

}

for (int i = 0; i < N; i++){

int l = 0;

for (int j = 0; j < N; j++){

if (set.contains(i\*N+j)){

l = 0;

}else{

l++;

}

dp[i][j] = Math.min(dp[i][j], l);

}

int r = 0;

for (int j = N-1; j >= 0; j--){

if (set.contains(i\*N+j)){

r = 0;

}else{

r++;

}

dp[i][j] = Math.min(dp[i][j], r);

}

}

for (int j = 0; j < N; j++){

int u = 0;

for (int i = 0; i < N; i++){

if (set.contains(i\*N+j)){

u = 0;

}else{

u++;

}

dp[i][j] = Math.min(dp[i][j], u);

}

int d = 0;

for (int i = N-1; i >= 0; i--){

if (set.contains(i\*N+j)){

d = 0;

}else{

d++;

}

dp[i][j] = Math.min(dp[i][j], d);

}

}

for (int i = 0; i < N; i++){

for (int j = 0; j < N; j++){

max = Math.max(max, dp[i][j]);

}

}

return max;

}

}

class Solution {

public int orderOfLargestPlusSign(int N, int[][] mines) {

Set<Integer> banned = new HashSet();

int[][] dp = new int[N][N];

for (int[] mine: mines)

banned.add(mine[0] \* N + mine[1]);

int ans = 0, count;

for (int r = 0; r < N; ++r) {

count = 0;

for (int c = 0; c < N; ++c) {

count = banned.contains(r\*N + c) ? 0 : count + 1;

dp[r][c] = count;

}

count = 0;

for (int c = N-1; c >= 0; --c) {

count = banned.contains(r\*N + c) ? 0 : count + 1;

dp[r][c] = Math.min(dp[r][c], count);

}

}

for (int c = 0; c < N; ++c) {

count = 0;

for (int r = 0; r < N; ++r) {

count = banned.contains(r\*N + c) ? 0 : count + 1;

dp[r][c] = Math.min(dp[r][c], count);

}

count = 0;

for (int r = N-1; r >= 0; --r) {

count = banned.contains(r\*N + c) ? 0 : count + 1;

dp[r][c] = Math.min(dp[r][c], count);

ans = Math.max(ans, dp[r][c]);

}

}

return ans;

}

}

Method 3:

https://leetcode.com/problems/largest-plus-sign/discuss/113314/JavaC++Python-O(N2)-solution-using-only-one-grid-matrix

765\_Couples.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

N couples sit in 2N seats arranged in a row and want to hold hands. We want to know the minimum number of swaps so that every

couple is sitting side by side. A swap consists of choosing any two people, then they stand up and switch seats.

The people and seats are represented by an integer from 0 to 2N-1, the couples are numbered in order, the first couple being (0, 1),

the second couple being (2, 3), and so on with the last couple being (2N-2, 2N-1).

The couples' initial seating is given by row[i] being the value of the person who is initially sitting in the i-th seat.

Example 1:

Input: row = [0, 2, 1, 3]

Output: 1

Explanation: We only need to swap the second (row[1]) and third (row[2]) person.

Example 2:

Input: row = [3, 2, 0, 1]

Output: 0

Explanation: All couples are already seated side by side.

Note:

len(row) is even and in the range of [4, 60].

row is guaranteed to be a permutation of 0...len(row)-1.

https://leetcode.com/problems/couples-holding-hands/discuss/117520/Java-union-find-easy-to-understand-5-ms

Union Find Cyclic swap

class Solution {

class UF {

int[] parent;

int count;

public UF (int n){

parent = new int[n];

for (int i = 0; i < n; i++){

parent[i] = i;

}

count = n;

}

public int find (int x){

if (parent[x] == x){

return x;

}

return parent[x] = find(parent[x]);

}

public void union (int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootY] = rootX;

count--;

}

}

}

public int minSwapsCouples(int[] row) {

int N = row.length / 2;

UF uf = new UF(N);

for (int i = 0; i < row.length; i += 2){

int x = row[i];

int y = row[i+1];

uf.union(x/2, y/2);

}

return N - uf.count;

}

}

766\_Toeplitz.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A matrix is Toeplitz if every diagonal from top-left to bottom-right has the same element.

Now given an M x N matrix, return True if and only if the matrix is Toeplitz.

Example 1:

Input: matrix = [[1,2,3,4],[5,1,2,3],[9,5,1,2]]

Output: True

Explanation:

1234

5123

9512

In the above grid, the diagonals are "[9]", "[5, 5]", "[1, 1, 1]", "[2, 2, 2]", "[3, 3]", "[4]", and in each diagonal all elements are the same, so the answer is True.

Example 2:

Input: matrix = [[1,2],[2,2]]

Output: False

Explanation:

The diagonal "[1, 2]" has different elements.

Note:

matrix will be a 2D array of integers.

matrix will have a number of rows and columns in range [1, 20].

matrix[i][j] will be integers in range [0, 99].

Method 1: best

class Solution {

public boolean isToeplitzMatrix(int[][] matrix) {

int m = matrix.length;

int n = matrix[0].length;

for (int i = 1; i < m; i++){

for (int j = 1; j < n; j++){

if (matrix[i][j] != matrix[i-1][j-1]){

return false;

}

}

}

return true;

}

}

Method 2:

class Solution {

public boolean isToeplitzMatrix(int[][] matrix) {

int m = matrix.length;

int n = matrix[0].length;

int idx = 0;

int[] temp = new int[m+n-1];

for (int i = m-1; i >= 0; i--){

temp[idx++] = matrix[i][0];

}

for (int j = 1; j < n; j++){

temp[idx++] = matrix[0][j];

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

int index = j - i + (m-1);

if (matrix[i][j] != temp[index]){

return false;

}

}

}

return true;

}

}

767\_Reorganize.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S, check if the letters can be rearranged so that two characters that are adjacent to each other are not the same.

If possible, output any possible result. If not possible, return the empty string.

Example 1:

Input: S = "aab"

Output: "aba"

Example 2:

Input: S = "aaab"

Output: ""

Note:

S will consist of lowercase letters and have length in range [1, 500].

Best solution:

The same as Rearrange String k Distance Apart

https://github.com/optimisea/Leetcode/edit/master/Java/358\_Rearrange.java

class Solution {

class Pair {

char ch;

int num;

public Pair (char ch, int num){

this.ch = ch;

this.num = num;

}

}

public String reorganizeString(String S) {

Map<Character, Integer> map = new HashMap<>();

for (char c : S.toCharArray()){

map.put(c, map.getOrDefault(c, 0) + 1);

}

Queue<Pair> maxPQ = new PriorityQueue<>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p2.num - p1.num;

}

});

for (char c : map.keySet()){

maxPQ.offer(new Pair(c, map.get(c)));

}

StringBuilder sb = new StringBuilder();

Queue<Pair> waitQ = new LinkedList<>();

int K = 2;

while (!maxPQ.isEmpty()){

Pair curr = maxPQ.poll();

sb.append(curr.ch);

waitQ.offer(new Pair(curr.ch, curr.num - 1));

if (waitQ.size() < K){

continue;

}

Pair p = waitQ.poll();

if (p.num > 0){

maxPQ.offer(new Pair(p.ch, p.num));

}

}

return sb.length() == S.length() ? sb.toString() : "";

}

}

Similar as Task Scheduler, greedy + PQ

class Solution {

class Pair {

char ch;

int num;

public Pair (char ch, int num){

this.ch = ch;

this.num = num;

}

}

public String reorganizeString(String S) {

PriorityQueue<Pair> maxPQ = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

if (p1.ch == p2.ch){

return p1.ch - p2.ch;

}

return p2.num - p1.num;

}

});

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i < S.length(); i++){

map.put(S.charAt(i), map.getOrDefault(S.charAt(i), 0) + 1);

if (map.get(S.charAt(i)) > (S.length() + 1) / 2) return "";

}

for (char key : map.keySet()){

maxPQ.offer(new Pair(key, map.get(key)));

}

StringBuilder sb = new StringBuilder();

while (!maxPQ.isEmpty()){

Pair first = maxPQ.poll();

if (sb.length() == 0 || first.ch != sb.charAt(sb.length() - 1)){

sb.append(first.ch);

if (first.num > 1){

maxPQ.offer(new Pair(first.ch, first.num - 1));

}

}else{

Pair second = maxPQ.poll();

sb.append(second.ch);

if (second.num > 1){

maxPQ.offer(new Pair(second.ch, second.num - 1));

}

maxPQ.offer(new Pair(first.ch, first.num));

}

}

return sb.toString();

}

}

Better solution:

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

class Pair{

char ch;

int num;

public Pair (char ch, int num){

this.ch = ch;

this.num = num;

}

}

public String reorganizeString(String S) {

char[] chars = S.toCharArray();

Map<Character, Integer> map = new HashMap<>();

for (char c : chars){

map.put(c, map.getOrDefault(c, 0) + 1);

}

Queue<Pair> pq = new PriorityQueue<>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p2.num - p1.num;

}

});

for (char c : map.keySet()){

pq.offer(new Pair(c, map.get(c)));

}

StringBuilder sb = new StringBuilder();

while (!pq.isEmpty()){

Pair curr = pq.poll();

if (pq.isEmpty()){

if (curr.num > 1){

return "";

}else{

sb.append(curr.ch);

break;

}

}

sb.append(curr.ch);

Pair next = pq.poll();

sb.append(next.ch);

if (curr.num > 1){

pq.offer(new Pair(curr.ch, curr.num - 1));

}

if (next.num > 1){

pq.offer(new Pair(next.ch, next.num - 1));

}

}

return sb.toString();

}

}

768\_Max.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This question is the same as "Max Chunks to Make Sorted" except the integers of the given array are not necessarily distinct, the

input array could be up to length 2000, and the elements could be up to 10\*\*8.

Given an array arr of integers (not necessarily distinct), we split the array into some number of "chunks" (partitions), and

individually sort each chunk. After concatenating them, the result equals the sorted array.

What is the most number of chunks we could have made?

Example 1:

Input: arr = [5,4,3,2,1]

Output: 1

Explanation:

Splitting into two or more chunks will not return the required result.

For example, splitting into [5, 4], [3, 2, 1] will result in [4, 5, 1, 2, 3], which isn't sorted.

Example 2:

Input: arr = [2,1,3,4,4]

Output: 4

Explanation:

We can split into two chunks, such as [2, 1], [3, 4, 4].

However, splitting into [2, 1], [3], [4], [4] is the highest number of chunks possible.

Note:

arr will have length in range [1, 2000].

arr[i] will be an integer in range [0, 10\*\*8].

https://leetcode.com/problems/max-chunks-to-make-sorted-ii/discuss/113466/simple-java-solution-with-explanation

Method 1:

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public int maxChunksToSorted(int[] arr) {

int[] sorted = arr.clone();

Arrays.sort(sorted);

int ans = 0;

int[] maxArr = new int[arr.length];

maxArr[0] = arr[0];

for (int i = 1; i < arr.length; i++){

maxArr[i] = Math.max(maxArr[i-1], arr[i]);

}

int upperLimit = Integer.MAX\_VALUE;

for (int i = arr.length - 1; i >= 0; i--){

if (maxArr[i] == sorted[i]){

if (sorted[i] <= upperLimit){

ans++;

upperLimit = arr[i];

}

}

}

return ans;

}

}

Method 2: Best solution

The same as Leetcode 915: Partition Array into Disjoint Intervals

https://github.com/optimisea/Leetcode/edit/master/Java/915\_Partition.java

Iterate through the array, each time all elements to the left are smaller (or equal) to all elements to the right,

there is a new chunck.

Time complexity: O(n)

Space complexity: O(n)

https://leetcode.com/problems/max-chunks-to-make-sorted-ii/discuss/113462/Java-solution-left-max-and-right-min.

class Solution {

public int maxChunksToSorted(int[] arr) {

int n = arr.length;

int[] leftMax = new int[n+1];

int[] rightMin = new int[n+1];

leftMax[0] = Integer.MIN\_VALUE;

rightMin[n] = Integer.MAX\_VALUE;

for (int i = 1; i <= n; i++){

leftMax[i] = Math.max(leftMax[i-1], arr[i-1]);

}

for (int i = n-1; i>= 0; i--){

rightMin[i] = Math.min(rightMin[i+1], arr[i]);

}

int count = 0;

for (int i = 0; i < n; i++){

if (leftMax[i] <= rightMin[i]){

count++;

}

}

return count;

}

}

class Solution {

public int maxChunksToSorted(int[] arr) {

int n = arr.length;

int[] leftMax = new int[n+1];

int[] rightMin = new int[n+1];

leftMax[0] = Integer.MIN\_VALUE;

rightMin[n] = Integer.MAX\_VALUE;

for (int i = 1; i <= n; i++){

leftMax[i] = Math.max(leftMax[i-1], arr[i-1]);

}

for (int i = n-1; i>= 0; i--){

rightMin[i] = Math.min(rightMin[i+1], arr[i]);

}

int count = 0;

for (int i = 1; i <= n; i++){

if (leftMax[i] <= rightMin[i]){

count++;

}

}

return count;

}

}

769\_Max.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array arr that is a permutation of [0, 1, ..., arr.length - 1], we split the array into some number of "chunks" (partitions),

and individually sort each chunk. After concatenating them, the result equals the sorted array.

What is the most number of chunks we could have made?

Example 1:

Input: arr = [4,3,2,1,0]

Output: 1

Explanation:

Splitting into two or more chunks will not return the required result.

For example, splitting into [4, 3], [2, 1, 0] will result in [3, 4, 0, 1, 2], which isn't sorted.

Example 2:

Input: arr = [1,0,2,3,4]

Output: 4

Explanation:

We can split into two chunks, such as [1, 0], [2, 3, 4].

However, splitting into [1, 0], [2], [3], [4] is the highest number of chunks possible.

Note:

arr will have length in range [1, 10].

arr[i] will be a permutation of [0, 1, ..., arr.length - 1].

https://leetcode.com/problems/max-chunks-to-make-sorted/discuss/113528/Simple-Java-O(n)-Solution-with-detailed-explanation

Intuition and Algorithm

Let's try to find the smallest left-most chunk. If the first k elements are [0, 1, ..., k-1], then it can be broken into a chunk,

and we have a smaller instance of the same problem.

We can check whether k+1 elements chosen from [0, 1, ..., n-1] are [0, 1, ..., k] by checking whether the maximum of that choice is k.

class Solution {

public int maxChunksToSorted(int[] arr) {

int ans = 0, max = 0;

for (int i = 0; i < arr.length; ++i) {

max = Math.max(max, arr[i]);

if (max == i) ans++;

}

return ans;

}

}

class Solution {

public int maxChunksToSorted(int[] arr) {

int count = 0;

int i = 0;

while (i < arr.length){

int j = arr[i];

while (i <= j && arr[i] <= j){

i++;

}

if (i == j + 1){

count++;

}

}

return count == 0 ? 1 : count;

}

}

77\_Combinations.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.

For example,

If n = 4 and k = 2, a solution is:

[

[2,4],

[3,4],

[2,3],

[1,2],

[1,3],

[1,4],

]

class Solution {

public List<List<Integer>> combine(int n, int k) {

List<List<Integer>> result = new ArrayList<>();

dfs(result, new ArrayList<>(), n, k, 1);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int n, int k, int start){

if (k == 0){

result.add(new ArrayList<Integer>(item));

return;

}

for (int i = start; i <= n; i++){

if (item.contains(i)){

continue;

}

item.add(i);

dfs(result, item, n, k-1, i+1);

item.remove(item.size() - 1);

}

}

}

This is combination, not permutation, so we need have a start flag to avoid [1, 3] and [3, 1] which are the same

class Solution {

public List<List<Integer>> combine(int n, int k) {

List<List<Integer>> res = new ArrayList<>();

dfs(res, new ArrayList<Integer>(), new HashSet<Integer>(), n, k, 1);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, Set<Integer> seen, int n, int k, int start){

if (item.size() == k){

res.add(new ArrayList<>(item));

return;

}

for (int i = start; i <= n; i++){

if (seen.contains(i)){

continue;

}

item.add(i);

seen.add(i);

dfs(res, item, seen, n, k, i+1);

seen.remove(i);

item.remove(item.size() - 1);

}

}

}

Best solution: no need to add HashSet seen

class Solution {

public List<List<Integer>> combine(int n, int k) {

List<List<Integer>> res = new ArrayList<>();

dfs(res, new ArrayList<Integer>(), n, k, 1);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, int n, int k, int start){

if (item.size() == k){

res.add(new ArrayList<>(item));

return;

}

for (int i = start; i <= n; i++){

item.add(i);

dfs(res, item, n, k, i+1);

item.remove(item.size() - 1);

}

}

}

77\_LCS.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings, find the longest common subsequence (LCS).

Your code should return the length of LCS.

Have you met this question in a real interview?

Clarification

What's the definition of Longest Common Subsequence?

https://en.wikipedia.org/wiki/Longest\_common\_subsequence\_problem

http://baike.baidu.com/view/2020307.htm

Example

For "ABCD" and "EDCA", the LCS is "A" (or "D", "C"), return 1.

For "ABCD" and "EACB", the LCS is "AC", return 2.

http://www.jiuzhang.com/solutions/longest-common-subsequence/

dp[i][j] is the length of longest common subarray ending or may not end with nums[i-1] and nums[j-1]

public class Solution {

/\*\*

\* @param A: A string

\* @param B: A string

\* @return: The length of longest common subsequence of A and B

\*/

public int longestCommonSubsequence(String A, String B) {

int m = A.length();

int n = B.length();

int[][] dp = new int[m+1][n+1];

for (int i = 0; i <= m; i++){

dp[i][0] = 0;

}

for (int j = 0; j <= n; j++){

dp[0][j] = 0;

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

if (A.charAt(i-1) == B.charAt(j-1)){

dp[i][j] = Math.max(dp[i-1][j-1] + 1, Math.max(dp[i-1][j], dp[i][j-1]));

}else{

dp[i][j] = Math.max(dp[i-1][j-1], Math.max(dp[i-1][j], dp[i][j-1]));

}

}

}

return dp[m][n];

}

}

771\_JewelsandStones.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You're given strings J representing the types of stones that are jewels, and S representing the stones you have.

Each character in S is a type of stone you have. You want to know how many of the stones you have are also jewels.

The letters in J are guaranteed distinct, and all characters in J and S are letters. Letters are case sensitive,

so "a" is considered a different type of stone from "A".

Example 1:

Input: J = "aA", S = "aAAbbbb"

Output: 3

Example 2:

Input: J = "z", S = "ZZ"

Output: 0

Note:

S and J will consist of letters and have length at most 50.

The characters in J are distinct.

class Solution {

public int numJewelsInStones(String J, String S) {

final int MAX\_CHAR = 256;

int count = 0;

int[] hash = new int[MAX\_CHAR];

for (int i = 0; i < J.length(); i++){

hash[J.charAt(i)]++;

}

for (int i = 0; i < S.length(); i++){

if (hash[S.charAt(i)] > 0){

count++;

}

}

return count;

}

}

class Solution {

public int numJewelsInStones(String J, String S) {

Set<Character> set = new HashSet<>();

for (char c : J.toCharArray()){

set.add(c);

}

int res = 0;

for(char c : S.toCharArray()){

if (set.contains(c)){

res++;

}

}

return res;

}

}

772\_GroupAnagrams.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of strings, group anagrams together.

Notice

All inputs will be in lower-case.

Have you met this question in a real interview? Yes

Example

Given strs = ["eat", "tea", "tan", "ate", "nat", "bat"],

Return

[

["ate", "eat","tea"],

["nat","tan"],

["bat"]

]

public class Solution {

/\*\*

\* @param strs: the given array of strings

\* @return: The anagrams which have been divided into groups

\*/

public List<List<String>> groupAnagrams(String[] strs) {

List<List<String>> result = new ArrayList<>();

if (strs == null || strs.length == 0){

return result;

}

Map<String, List<String>> map = new HashMap<>();

for (int i = 0; i < strs.length; i++){

char[] sc = strs[i].toCharArray();

Arrays.sort(sc);

String s = String.valueOf(sc);

if (!map.containsKey(s)){

map.put(s, new ArrayList<String>());

}

map.get(s).add(strs[i]);

}

for (Map.Entry<String, List<String>> entry : map.entrySet()){

result.add(entry.getValue());

}

return result;

}

}

773\_Sliding.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a 2x3 board, there are 5 tiles represented by the integers 1 through 5, and an empty square represented by 0.

A move consists of choosing 0 and a 4-directionally adjacent number and swapping it.

The state of the board is solved if and only if the board is [[1,2,3],[4,5,0]].

Given a puzzle board, return the least number of moves required so that the state of the board is solved. If it is impossible

for the state of the board to be solved, return -1.

Examples:

Input: board = [[1,2,3],[4,0,5]]

Output: 1

Explanation: Swap the 0 and the 5 in one move.

Input: board = [[1,2,3],[5,4,0]]

Output: -1

Explanation: No number of moves will make the board solved.

Input: board = [[4,1,2],[5,0,3]]

Output: 5

Explanation: 5 is the smallest number of moves that solves the board.

An example path:

After move 0: [[4,1,2],[5,0,3]]

After move 1: [[4,1,2],[0,5,3]]

After move 2: [[0,1,2],[4,5,3]]

After move 3: [[1,0,2],[4,5,3]]

After move 4: [[1,2,0],[4,5,3]]

After move 5: [[1,2,3],[4,5,0]]

Input: board = [[3,2,4],[1,5,0]]

Output: 14

Note:

board will be a 2 x 3 array as described above.

board[i][j] will be a permutation of [0, 1, 2, 3, 4, 5].

Method 1: BFS 9ms (Best solution)

https://leetcode.com/problems/sliding-puzzle/discuss/146652/Java-8ms-BFS-with-algorithm-explained

Consider each state in the board as a graph node, we just need to find out the min distance between start node and final target

node "123450". Since it's a single point to single point questions, Dijkstra is not needed here. We can simply use BFS, and also

count the level we passed. Every time we swap 0 position in the String to find the next state. Use a hashTable to store the visited

states.

class Solution {

public int slidingPuzzle(int[][] board) {

int ans = 0;

int m = board.length;

int n = board[0].length;

int[][] dirs = {{1, 3}, {0, 2, 4}, {1,5}, {0, 4}, {1, 3, 5}, {2, 4}};

String target = "123450";

String source = "";

for (int i = 0; i < board.length; i++){

for (int j = 0; j < board[0].length; j++){

source += board[i][j];

}

}

Set<String> seen = new HashSet<>();

Queue<String> queue = new LinkedList<>();

queue.offer(source);

seen.add(source);

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

String curr = queue.poll();

if (curr.equals(target)){

return ans;

}

int index = curr.indexOf("0");

for (int dir : dirs[index]){

String str = swap(curr, index, dir);

if (!seen.contains(str)){

queue.offer(str);

seen.add(str);

}

}

}

ans++;

}

return -1;

}

private String swap(String curr, int i, int j){

StringBuilder sb = new StringBuilder(curr);

sb.setCharAt(i, curr.charAt(j));

sb.setCharAt(j, curr.charAt(i));

return sb.toString();

}

}

Method 2: DFS

class Solution {

int min = Integer.MAX\_VALUE;

public int slidingPuzzle(int[][] board) {

String source = "";

for (int i = 0; i < board.length; i++){

for (int j = 0; j < board[0].length; j++){

source += board[i][j];

}

}

String target = "123450";

int[][] dirs = {{1, 3}, {0, 2, 4}, {1, 5}, {0, 4}, {1, 3, 5}, {2, 4}};

Map<String, Integer> map = new HashMap<>();

dfs(target, source, dirs, 0, map);

return min == Integer.MAX\_VALUE ? -1 : min;

}

private void dfs(String target, String source, int[][] dirs, int moves, Map<String, Integer> map){

if (target.equals(source)){

min = Math.min(min, moves);

return;

}

if (map.containsKey(source)){

if (moves > map.get(source)){

return;

}

}

map.put(source, moves);

int index = source.indexOf("0");

for (int i = 0 ; i < dirs[index].length; i++){

String next = swap(source, dirs[index][i], index);

dfs(target, next, dirs, moves + 1, map);

}

}

private String swap (String str, int i , int j){

StringBuilder sb = new StringBuilder(str);

sb.setCharAt(i, str.charAt(j));

sb.setCharAt(j, str.charAt(i));

return sb.toString();

}

}

Method 3: backtracking 167ms

class Solution {

int min = Integer.MAX\_VALUE;

public int slidingPuzzle(int[][] board) {

int m = board.length;

int n = board[0].length;

int xStart = 0;

int yStart = 0;

Map<Integer, Integer> map = new HashMap<>();

map.put(encode(board), 0);

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (board[i][j] == 0){

xStart = i;

yStart = j;

break;

}

}

}

dfs(board, xStart, yStart, 0, map);

return min == Integer.MAX\_VALUE ? -1 : min;

}

private void dfs(int[][] board, int xStart, int yStart, int moves, Map<Integer, Integer> map){

int code = encode(board);

if (code == 123450){

min = Math.min(min, moves);

return;

}

if (map.containsKey(code)){//way to prune duplicate moves

if (moves > map.get(code)){

return;

}

}

map.put(code, moves);

int m = board.length;

int n = board[0].length;

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

for (int i = 0; i < dx.length; i++){

int x = xStart + dx[i];

int y = yStart + dy[i];

if (x >= 0 && x < m && y >= 0 && y < n){

swap(board, x, y, xStart, yStart);

dfs(board, x, y, moves + 1, map);

swap(board, x, y, xStart, yStart);

}

}

}

private void swap(int[][] board, int x, int y, int xStart, int yStart){

int temp = board[x][y];

board[x][y] = board[xStart][yStart];

board[xStart][yStart] = temp;

}

private int encode(int[][] board){

int code = 0;

for (int i = 0; i < board.length; i++){

for (int j = 0; j < board[0].length; j++){

code = code \* 10 + board[i][j];

}

}

return code;

}

}

774\_Minimize.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a horizontal number line, we have gas stations at positions stations[0], stations[1], ..., stations[N-1], where N = stations.length.

Now, we add K more gas stations so that D, the maximum distance between adjacent gas stations, is minimized.

Return the smallest possible value of D.

Example:

Input: stations = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], K = 9

Output: 0.500000

Note:

stations.length will be an integer in range [10, 2000].

stations[i] will be an integer in range [0, 10^8].

K will be an integer in range [1, 10^6].

Answers within 10^-6 of the true value will be accepted as correct.

Binary Search : O(NLogM) N -- stations.length; M --stations[N-1] - stations[0]

if we are to have mid as the min max distance between any two buildings, this count is how many more gas stations that we can

add to the array and we can compute this count by trying to inserting new gas stations between the 2 adjacent houses.

For any 2 adjacent gas stations, we divide their original distance by the min max distance (mid) we could have in the array,

to try to insert as many gas stations as we can into any 2 adjacent gas stations.

this step is computing the number of houses we can insert into this original array.

class Solution {

public double minmaxGasDist(int[] stations, int K) {

int N = stations.length;

double low = 0;

double high = stations[N-1] - stations[0];

while (low + 1e-6 < high){

double mid = (low + high) / 2.0;

if (possible(mid, stations, K)){

high = mid;

}else{

low = mid;

}

}

return low;

}

private boolean possible(double mid, int[] stations, int K){

int used = 0;

for (int i = 0; i < stations.length - 1; i++){

used += (int) (stations[i+1] - stations[i]) / mid; (note that this is double division then conver to int, it is like ceiling() - 1

}

return used <= K;

}

}

Why did I use s binary search?

In fact there are some similar problems on Leetcode so that is part of experience.

Secondly, I got a hint from "Answers within 10^-6 of the true value will be accepted as correct.". The first solution I tried

was binary search. Because binary search may not find exact value but it can approach the true answer.

Explanation of solution

Now we are using binary search to find the smallest possible value of D.

I initilze left = 0 and right = the distance between the first and the last station

count is the number of gas station we need to make it possible.

if count > K, it means mid is too small to realize using only K more stations.

if count <= K, it means mid is possible and we can continue to find a bigger one.

When left + 1e-6 >= right, it means the answer within 10^-6 of the true value and it will be accepted.

Time complexity:

O(NlogM), where N is station length and M is st[N - 1] - st[0]

Intuition

Let's ask possible(D): with K (or less) gas stations, can we make every adjacent distance between gas stations at most D?

This function is monotone, so we can apply a binary search to find D\*D^{\text{\*}}D​\*​​.

Algorithm

More specifically, there exists some D\* (the answer) for which possible(d) = False when d < D\* and possible(d) = True when d > D\*.

Binary searching a monotone function is a typical technique, so let's focus on the function possible(D).

When we have some interval like X = stations[i+1] - stations[i], we'll need to use ⌊XD⌋\lfloor \frac{X}{D} \rfloor⌊​D​​X​​⌋

gas stations to ensure every subinterval has size less than D. This is independent of other intervals, so in total we'll

need to use ∑i⌊XiD⌋\sum\_i \lfloor \frac{X\_i}{D} \rfloor∑​i​​⌊​D​​X​i​​​​⌋

gas stations. If this is at most K, then it is possible to make every adjacent distance between gas stations at most D

775\_Global\_Local\_Inversions.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have some permutation A of [0, 1, ..., N - 1], where N is the length of A.

The number of (global) inversions is the number of i < j with 0 <= i < j < N and A[i] > A[j].

The number of local inversions is the number of i with 0 <= i < N and A[i] > A[i+1].

Return true if and only if the number of global inversions is equal to the number of local inversions.

Example 1:

Input: A = [1,0,2]

Output: true

Explanation: There is 1 global inversion, and 1 local inversion.

Example 2:

Input: A = [1,2,0]

Output: false

Explanation: There are 2 global inversions, and 1 local inversion.

All local inversions are global inversions.

If the number of global inversions is equal to the number of local inversions,

it means that all global inversions in permutations are local inversions.

It also means that we can not find A[i] > A[j] with i+2<=j.

In other words, max(A[i]) < A[i+2]

class Solution {

public boolean isIdealPermutation(int[] A) {

int max = 0;

for (int i = 0; i < A.length - 2; i++){

max = Math.max(max, A[i]);

if (max > A[i+2]){

return false;

}

}

return true;

}

}

776\_SplitBST.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a Binary Search Tree (BST) with root node root, and a target value V, split the

tree into two subtrees where one subtree has nodes that are all smaller or equal to the target

value, while the other subtree has all nodes that are greater than the target value. It's not necessarily

the case that the tree contains a node with value V.

Additionally, most of the structure of the original tree should remain. Formally, for any child C

with parent P in the original tree, if they are both in the same subtree after the split, then node C

should still have the parent P.

You should output the root TreeNode of both subtrees after splitting, in any order.

Example 1:

Input: root = [4,2,6,1,3,5,7], V = 2

Output: [[2,1],[4,3,6,null,null,5,7]]

Explanation:

Note that root, output[0], and output[1] are TreeNode objects, not arrays.

The given tree [4,2,6,1,3,5,7] is represented by the following diagram:

4

/ \

2 6

/ \ / \

1 3 5 7

while the diagrams for the outputs are:

4

/ \

3 6 and 2

/ \ /

5 7 1

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode[] splitBST(TreeNode root, int V) {

if (root == null){

return new TreeNode[]{null, null};

}else if (root.val <= V){

TreeNode[] ans = splitBST(root.right, V);

root.right = ans[0];

ans[0] = root;

return ans;

}else{

TreeNode[] ans = splitBST(root.left, V);

root.left = ans[1];

ans[1] = root;

return ans;

}

}

}

ntuition and Algorithm

The root node either belongs to the first half or the second half. Let's say it belongs to the first half.

Then, because the given tree is a binary search tree (BST), the entire subtree at root.left must be in the first half.

However, the subtree at root.right may have nodes in either halves, so it needs to be split.

Diagram of tree being split

In the diagram above, the thick lines represent the main child relationships between the nodes,

while the thinner colored lines represent the subtrees after the split.

Lets say our secondary answer bns = split(root.right) is the result of such a split. Recall that bns[0]

and bns[1] will both be BSTs on either side of the split. The left half of bns must be in the first half,

and it must be to the right of root for the first half to remain a BST. The right half of bns is the right

half in the final answer.

Diagram of how root tree connects to split of subtree at root.right

The diagram above explains how we merge the two halves of split(root.right) with the main tree, and illustrates the line of code root.right = bns[0] in the implementations.

777\_Swap.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a string composed of 'L', 'R', and 'X' characters, like "RXXLRXRXL", a move consists of either replacing one occurrence of "XL"

with "LX", or replacing one occurrence of "RX" with "XR". Given the starting string start and the ending string end, return True

if and only if there exists a sequence of moves to transform one string to the other.

Example:

Input: start = "RXXLRXRXL", end = "XRLXXRRLX"

Output: True

Explanation:

We can transform start to end following these steps:

RXXLRXRXL ->

XRXLRXRXL ->

XRLXRXRXL ->

XRLXXRRXL ->

XRLXXRRLX

Note:

1 <= len(start) = len(end) <= 10000.

Both start and end will only consist of characters in {'L', 'R', 'X'}.

Method 1: Invariant

1. number of L should be equal to number of R

2. L can only shift to left; R can only shift to right

class Solution {

public boolean canTransform(String start, String end) {

if (!start.replace("X", "").equals(end.replace("X", ""))){

return false;

}

int t = 0;

for (int i = 0; i < start.length(); i++){

if (start.charAt(i) != 'L'){

continue;

}

while (end.charAt(t) != 'L'){

t++;

}

if (t > i){

return false;

}

t++;

}

t = 0;

for (int i = 0; i < start.length(); i++){

if (start.charAt(i) != 'R'){

continue;

}

while (end.charAt(t) != 'R'){

t++;

}

if (t < i){

return false;

}

t++;

}

return true;

}

}

Method 2: One pass Better solution

class Solution {

public boolean canTransform(String start, String end) {

char[] S = start.toCharArray();

char[] T = end.toCharArray();

int i = 0;

int j = 0;

int N = S.length;

while (i < N || j < N){

while (i < N && S[i] == 'X'){

i++;

}

while (j < N && T[j] == 'X'){

j++;

}

if ((i == N) ^ (j == N)){

return false;

}

if (i < N && j < N){

if (S[i] != T[j] || (S[i] == 'L' && i < j) || (S[i] == 'R' && i > j)){

return false;

}

}

i++;

j++;

}

return true;

}

}

Corner case:

"XXRXLXRXRX"

"XXRLXXXXXR"

778\_Swim.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On an N x N grid, each square grid[i][j] represents the elevation at that point (i,j).

Now rain starts to fall. At time t, the depth of the water everywhere is t. You can swim from a square to another 4-directionally

adjacent square if and only if the elevation of both squares individually are at most t. You can swim infinite distance in zero time.

Of course, you must stay within the boundaries of the grid during your swim.

You start at the top left square (0, 0). What is the least time until you can reach the bottom right square (N-1, N-1)?

Example 1:

Input: [[0,2],[1,3]]

Output: 3

Explanation:

At time 0, you are in grid location (0, 0).

You cannot go anywhere else because 4-directionally adjacent neighbors have a higher elevation than t = 0.

You cannot reach point (1, 1) until time 3.

When the depth of water is 3, we can swim anywhere inside the grid.

Example 2:

Input: [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]

Output: 16

Explanation:

0 1 2 3 4

24 23 22 21 5

12 13 14 15 16

11 17 18 19 20

10 9 8 7 6

The final route is marked in bold.

We need to wait until time 16 so that (0, 0) and (4, 4) are connected.

Note:

2 <= N <= 50.

grid[i][j] is a permutation of [0, ..., N\*N - 1].

Method 1: BFS + priority queue

Time complexity: O(N^2 \* logN) We may expand O(N^2) nodes,

and each one requires O(logN)time to perform the heap operations.

Space complexity: O(N^2)

class Solution {

class Pair {

int x;

int y;

int height;

public Pair (int x, int y, int height){

this.x = x;

this.y = y;

this.height = height;

}

}

public int swimInWater(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

Queue<Pair> pq = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p1.height - p2.height;

}

});

int max = 0;

Set<Integer> visited = new HashSet<>();

pq.offer(new Pair(0, 0, grid[0][0]));

visited.add(0);

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

while (!pq.isEmpty()){

Pair p = pq.poll();

max = Math.max(max, p.height);

if (p.x \* n + p.y == (m-1) \* n + n - 1){

return max;

}

for (int i = 0; i < dx.length; i++){

int x = p.x + dx[i];

int y = p.y + dy[i];

if (0 <= x && x < m && 0 <= y && y < n && !visited.contains(x \* n + y)){

pq.offer(new Pair(x, y, grid[x][y]));

visited.add(x \* n + y);

}

}

}

throw null;

}

}

class Solution {

class Pair {

int x;

int y;

int height;

public Pair (int x, int y, int height){

this.x = x;

this.y = y;

this.height = height;

}

}

public int swimInWater(int[][] grid) {

int N = grid.length;

Queue<Pair> pq = new PriorityQueue<>(new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p1.height - p2.height;

}

});

pq.offer(new Pair(0, 0, grid[0][0]));

boolean[][] inQueue = new boolean[N][N];

inQueue[0][0] = true;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

int max = 0;

while (!pq.isEmpty()){

Pair p = pq.poll();

max = Math.max(max, p.height);

if (p.x == N-1 && p.y == N-1 && max >= p.height){

return max;

}

for (int[] dir : dirs){

int nx = p.x + dir[0];

int ny = p.y + dir[1];

if (nx >= 0 && nx < N && ny >= 0 && ny < N && !inQueue[nx][ny]){

pq.offer(new Pair(nx, ny, grid[nx][ny]));

inQueue[nx][ny] = true;

}

}

}

return max;

}

}

Method 2: Binary Search + DFS

Intuition and Algorithm

Whether the swim is possible is a monotone function with respect to time, so we can binary search this function for the correct time:

the smallest T for which the swim is possible.

Say we guess that the correct time is T. To check whether it is possible, we perform a simple depth-first search where we can only

walk in squares that are at most T.

Time Complexity: O(N2logN) Our depth-first search during a call to possible is O(N2) and we make up to O(logN) of them.

Space Complexity: O(N2), the maximum size of the stack.

class Solution {

public int swimInWater(int[][] grid) {

int N = grid.length;

int lo = grid[0][0], hi = N \* N;

while (lo < hi) {

int mi = lo + (hi - lo) / 2;

if (!possible(mi, grid)) {

lo = mi + 1;

} else {

hi = mi;

}

}

return lo;

}

public boolean possible(int T, int[][] grid) {

int N = grid.length;

Set<Integer> seen = new HashSet();

seen.add(0);

int[] dr = new int[]{1, -1, 0, 0};

int[] dc = new int[]{0, 0, 1, -1};

Stack<Integer> stack = new Stack();

stack.add(0);

while (!stack.empty()) {

int k = stack.pop();

int r = k / N, c = k % N;

if (r == N-1 && c == N-1) return true;

for (int i = 0; i < 4; ++i) {

int cr = r + dr[i], cc = c + dc[i];

int ck = cr \* N + cc;

if (0 <= cr && cr < N && 0 <= cc && cc < N

&& !seen.contains(ck) && grid[cr][cc] <= T) {

stack.add(ck);

seen.add(ck);

}

}

}

return false;

}

}

779\_Kth.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On the first row, we write a 0. Now in every subsequent row, we look at the previous row and replace each occurrence of 0 with 01, and each occurrence of 1 with 10.

Given row N and index K, return the K-th indexed symbol in row N. (The values of K are 1-indexed.) (1 indexed).

Examples:

Input: N = 1, K = 1

Output: 0

Input: N = 2, K = 1

Output: 0

Input: N = 2, K = 2

Output: 1

Input: N = 4, K = 5

Output: 1

Explanation:

row 1: 0

row 2: 01

row 3: 0110

row 4: 01101001

Note:

N will be an integer in the range [1, 30].

K will be an integer in the range [1, 2^(N-1)].

Time complexity: O(N) or O(LogK)

class Solution {

public int kthGrammar(int N, int K) {

if (N == 1 && K == 1){

return 0;

}

if (kthGrammar(N-1, (K+1)/2) == 1){

if (K%2 == 1){

return 1;

}else{

return 0;

}

}else{

if (K%2 == 0){

return 1;

}else{

return 0;

}

}

}

}

78\_Subsets.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a set of distinct integers, nums, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

For example,

If nums = [1,2,3], a solution is:

[

[3],

[1],

[2],

[1,2,3],

[1,3],

[2,3],

[1,2],

[]

]

class Solution {

public List<List<Integer>> subsets(int[] nums) {

List<List<Integer>> result = new ArrayList<>();

dfs(result, new ArrayList<Integer>(), nums, 0);

return result;

}

private void dfs(List<List<Integer>> result, List<Integer> item, int[] nums, int start){

result.add(new ArrayList<Integer>(item));

for (int i = start; i < nums.length; i++){

item.add(nums[i]);

dfs(result, item, nums, i + 1);

item.remove(item.size() - 1);

}

}

}

780\_Reaching.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A move consists of taking a point (x, y) and transforming it to either (x, x+y) or (x+y, y).

Given a starting point (sx, sy) and a target point (tx, ty), return True if and only if a sequence of moves exists to transform the point (sx, sy) to (tx, ty). Otherwise, return False.

Examples:

Input: sx = 1, sy = 1, tx = 3, ty = 5

Output: True

Explanation:

One series of moves that transforms the starting point to the target is:

(1, 1) -> (1, 2)

(1, 2) -> (3, 2)

(3, 2) -> (3, 5)

Input: sx = 1, sy = 1, tx = 2, ty = 2

Output: False

Input: sx = 1, sy = 1, tx = 1, ty = 1

Output: True

Note:

sx, sy, tx, ty will all be integers in the range [1, 10^9].

https://leetcode.com/problems/reaching-points/discuss/114856/Easy-and-Concise-2-line-SolutionPythonC++Java

Search from target to source instead of source to target because it's always easy to get the root

because you have exact one way to get the parent node, think as tree

using ty%=tx instead of while (tx > ty) { tx -= ty }

class Solution {

public boolean reachingPoints(int sx, int sy, int tx, int ty) {

while (tx > sx && ty > sy){

if (tx > ty){

tx %= ty;

}else{

ty %= tx;

}

}

if (tx == sx && sy <= ty && (ty - sy) % sx == 0){

return true;

}

if (ty == sy && sx <= tx && (tx - sx) % sy == 0){

return true;

}

return false;

}

}

781\_Rabbits.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a forest, each rabbit has some color. Some subset of rabbits (possibly all of them) tell you how many other rabbits have the same color as them. Those answers are placed in an array.

Return the minimum number of rabbits that could be in the forest.

Examples:

Input: answers = [1, 1, 2]

Output: 5

Explanation:

The two rabbits that answered "1" could both be the same color, say red.

The rabbit than answered "2" can't be red or the answers would be inconsistent.

Say the rabbit that answered "2" was blue.

Then there should be 2 other blue rabbits in the forest that didn't answer into the array.

The smallest possible number of rabbits in the forest is therefore 5: 3 that answered plus 2 that didn't.

Input: answers = [10, 10, 10]

Output: 11

Input: answers = []

Output: 0

Note:

answers will have length at most 1000.

Each answers[i] will be an integer in the range [0, 999].

class Solution {

public int numRabbits(int[] answers) {

if (answers.length == 0){

return 0;

}

Map<Integer, Double> map = new HashMap<>();

for (int answer : answers){

map.put(answer, map.getOrDefault(answer, 0.0) + 1.0);

}

int count = 0;

for (int key : map.keySet()){

double val = map.get(key);

count += (key + 1) \* (int) Math.ceil(val / (key + 1)) ;

}

return count;

}

}

https://leetcode.com/problems/rabbits-in-forest/discuss/114721/C++JavaPython-Easy-and-Concise-Solution

If x+1 rabbits have same color, then we get x+1 rabbits who all answer x.

now n rabbits answer x.

If n % (x + 1) == 0, we need n / (x + 1) groups of x + 1 rabbits.

If n % (x + 1) != 0, we need n / (x + 1) + 1 groups of x + 1 rabbits.

the number of groups is math.ceil(n / (x + 1)) and it equals to (n + x) / (x + 1) , which is more elegant.

class Solution {

public int numRabbits(int[] answers) {

if (answers.length == 0){

return 0;

}

Map<Integer, Integer> map = new HashMap<>();

for (int answer : answers){

map.put(answer, map.getOrDefault(answer, 0) + 1);

}

int count = 0;

for (int key : map.keySet()){

int val = map.get(key);

count += (val + key) / (key + 1) \* (key + 1);

}

return count;

}

}

784\_Letter.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S, we can transform every letter individually to be lowercase or uppercase to create another string. Return a list of all possible strings we could create.

Examples:

Input: S = "a1b2"

Output: ["a1b2", "a1B2", "A1b2", "A1B2"]

Input: S = "3z4"

Output: ["3z4", "3Z4"]

Input: S = "12345"

Output: ["12345"]

Note:

S will be a string with length at most 12.

S will consist only of letters or digits.

https://leetcode.com/problems/letter-case-permutation/discuss/115508/Java-solution-using-recursion

Method 1: recursion

class Solution {

public List<String> letterCasePermutation(String S) {

List<String> res = new ArrayList<>();

dfs(res, S.toCharArray(), 0);

return res;

}

private void dfs(List<String> res, char[] chars, int pos){

if (pos == chars.length){

res.add(String.valueOf(chars));

return;

}

if (Character.isLetter(chars[pos])){

chars[pos] = Character.toLowerCase(chars[pos]);

dfs(res, chars, pos+1);

chars[pos] = Character.toUpperCase(chars[pos]);

}

dfs(res, chars, pos + 1);

}

}

class Solution {

public List<String> letterCasePermutation(String S) {

List<String> res = new ArrayList<>();

dfs(res, S.toCharArray(), 0);

return res;

}

private void dfs(List<String> res, char[] chars, int pos){

if (pos == chars.length){

res.add(String.valueOf(chars));

return;

}

dfs(res, chars, pos + 1);

if (Character.isLetter(chars[pos])){

if (Character.isUpperCase(chars[pos])){

chars[pos] = Character.toLowerCase(chars[pos]);

}else{

chars[pos] = Character.toUpperCase(chars[pos]);

}

dfs(res, chars, pos + 1);

}

}

}

Best solution:

class Solution {

public List<String> letterCasePermutation(String S) {

List<String> res = new ArrayList<>();

dfs(S, res, "", 0);

return res;

}

private void dfs(String S, List<String> res, String item, int start){

if (start == S.length()){

res.add(item);

return;

}

char c = S.charAt(start);

if (Character.isLetter(c)){

dfs(S, res, item + Character.toUpperCase(c), start + 1);

dfs(S, res, item + Character.toLowerCase(c), start + 1);

}else{

dfs(S, res, item + c, start + 1);

}

}

}

Method 2: Backtracking

class Solution {

public List<String> letterCasePermutation(String S) {

List<String> res = new ArrayList<>();

dfs(res, new StringBuilder(), S, 0);

return res;

}

private void dfs(List<String> res, StringBuilder sb, String S, int pos){

if (pos == S.length()){

res.add(sb.toString());

return;

}

char c = S.charAt(pos);

if (Character.isLetter(c)){

char lower = Character.toLowerCase(c);

sb.append(lower);

dfs(res, sb, S, pos+1);

sb.deleteCharAt(sb.length() - 1);

char upper = Character.toUpperCase(c);

sb.append(upper);

dfs(res, sb, S, pos+1);

sb.deleteCharAt(sb.length() - 1);

}else{

sb.append(c);

dfs(res, sb, S, pos+1);

sb.deleteCharAt(sb.length() - 1);

}

}

}

785\_Is.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an undirected graph, return true if and only if it is bipartite.

Recall that a graph is bipartite if we can split it's set of nodes into two independent subsets A and B such that every edge in

the graph has one node in A and another node in B.

The graph is given in the following form: graph[i] is a list of indexes j for which the edge between nodes i and j exists.

Each node is an integer between 0 and graph.length - 1. There are no self edges or parallel edges: graph[i] does not contain i,

and it doesn't contain any element twice.

Example 1:

Input: [[1,3], [0,2], [1,3], [0,2]]

Output: true

Explanation:

The graph looks like this:

0----1

| |

| |

3----2

We can divide the vertices into two groups: {0, 2} and {1, 3}.

Example 2:

Input: [[1,2,3], [0,2], [0,1,3], [0,2]]

Output: false

Explanation:

The graph looks like this:

0----1

| \ |

| \ |

3----2

We cannot find a way to divide the set of nodes into two independent subsets.

Note:

graph will have length in range [1, 100].

graph[i] will contain integers in range [0, graph.length - 1].

graph[i] will not contain i or duplicate values.

The graph is undirected: if any element j is in graph[i], then i will be in graph[j].

class Solution {

public boolean isBipartite(int[][] graph) {

int n = graph.length;

int[] color = new int[n];

for (int i = 0; i < n; i++){

if (color[i] == 0){

color[i] = -1;

}

if (!dfs(graph, color, i)){

return false;

}

}

return true;

}

private boolean dfs(int[][] graph, int[] color, int start){

for (int nei : graph[start]){

if (color[nei] == 0){

color[nei] = -color[start];

if (!dfs(graph, color, nei)){

return false;

}

}else{

if (color[nei] == color[start]){

return false;

}

}

}

return true;

}

}

Better version:

class Solution {

public boolean isBipartite(int[][] graph) {

int n = graph.length;

int[] colors = new int[n];

for (int i = 0; i < n; i++){ //This graph might be a disconnected graph. So check each unvisited node.

if (colors[i] == 0 && !isValid(graph, i, 1, colors)){

return false;

}

}

return true;

}

private boolean isValid(int[][] graph, int curr, int color, int[] colors){

colors[curr] = color;

for (int nei : graph[curr]){

if (colors[nei] == 0){

if(!isValid(graph, nei, -color, colors))

return false;

}else if (colors[nei] == color){

return false;

}

}

return true;

}

}

786\_kth.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A sorted list A contains 1, plus some number of primes. Then, for every p < q in the list, we consider the fraction p/q.

What is the K-th smallest fraction considered? Return your answer as an array of ints, where answer[0] = p and answer[1] = q.

Examples:

Input: A = [1, 2, 3, 5], K = 3

Output: [2, 5]

Explanation:

The fractions to be considered in sorted order are:

1/5, 1/3, 2/5, 1/2, 3/5, 2/3.

The third fraction is 2/5.

Input: A = [1, 7], K = 1

Output: [1, 7]

Note:

A will have length between 2 and 2000.

Each A[i] will be between 1 and 30000.

K will be between 1 and A.length \* (A.length - 1) / 2.

https://leetcode.com/problems/k-th-smallest-prime-fraction/discuss/115819/Summary-of-solutions-for-problems-%22reducible%22-to-LeetCode-378

Method 1:

Time complexity: O(klogn)

Space complexity: O(n)

class Solution {

public int[] kthSmallestPrimeFraction(int[] A, int K) {

int n = A.length;

Queue<int[]> minPQ = new PriorityQueue<int[]>(new Comparator<int[]>(){

public int compare (int[] p1, int[] p2){

return A[p1[0]] \* A[p2[1]] - A[p2[0]] \* A[p1[1]];

}

});

for (int i = 0; i < n - 1; i++){

minPQ.offer(new int[]{i, n-1});

}

for (int i = 0; i < K-1; i++){

int[] p = minPQ.poll();

if (A[p[1] - 1] > A[p[0]]){

minPQ.offer(new int[]{p[0], p[1] - 1});

}

}

int[] res = minPQ.poll();

return new int[]{A[res[0]], A[res[1]]};

}

}

Method 2:

Time complexity: O(nLogmaxProd)

Space complexity: O(1)

class Solution {

public int[] kthSmallestPrimeFraction(int[] A, int K) {

int prod = 1;

for (int i : A){

prod \*= i;

}

System.out.println(prod);

int[] res = new int[2];

int low = prod/A[A.length-1];

int high = prod;

while (low <= high){

int mid = low + (high - low) / 2;

int count = getLessEqual(A, mid, prod);

System.out.println(count);

System.out.println("low = " + low);

System.out.println("high = " + high);

if (count <= K-1){

low = mid + 1;

}else{

high = mid - 1;

}

}

System.out.println("k th smallest = " + low);

for (int j = 0; j < A.length; j++){

if ((A[j] \* low) % prod == 0){

res[0] = (A[j] \* low) / prod;

res[1] = A[j];

// break;

}

}

return res;

}

private int getLessEqual(int[] A, int val, int prod){

int res = 0;

int n = A.length;

for (int i = 0; i < A.length; i++){

int j = n-1;

while (j >= i+1 && prod \* A[i] <= A[j] \* val){

j--;

}

res += n-j-1;

}

return res;

}

}

# 787\_Cheapest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are n cities connected by m flights. Each fight starts from city u and arrives at v with a price w.

Now given all the cities and fights, together with starting city src and the destination dst, your task is to find the cheapest price from src to dst with up to k stops. If there is no such route, output -1.

Example 1:

Input:

n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]

src = 0, dst = 2, k = 1

Output: 200

Explanation:

The graph looks like this:

The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red in the picture.

Example 2:

Input:

n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]

src = 0, dst = 2, k = 0

Output: 500

Explanation:

The graph looks like this:

The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue in the picture.

Note:

The number of nodes n will be in range [1, 100], with nodes labeled from 0 to n - 1.

The size of flights will be in range [0, n \* (n - 1) / 2].

The format of each flight will be (src, dst, price).

The price of each flight will be in the range [1, 10000].

k is in the range of [0, n - 1].

There will not be any duplicated flights or self cycles.

Similar to Network Delay time but include additional dimension (record how many stops)

Dijkstra algorithm

class Solution {

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {

//build graph

Map<Integer, Map<Integer, Integer>> graph = new HashMap<>();

for (int[] flight : flights){

Map<Integer, Integer> sourceMap = graph.get(flight[0]);

if (sourceMap == null){

sourceMap = new HashMap<>();

graph.put(flight[0], sourceMap);

}

Integer d = sourceMap.get(flight[1]);

if (d == null){

sourceMap.put(flight[1], flight[2]);

}

}

//minPQ to store the shortest distance from source to the current point

PriorityQueue<int[]> pq = new PriorityQueue<>(new Comparator<int[]>(){

public int compare (int[] a, int[] b){

return a[1] - b[1];

}

});

Map<Integer, Integer> distMap = new HashMap<>(); // store the shortest distance between src and current point with different stops

// distMap.put(src, 0);

pq.offer(new int[]{src, 0, 0});

while (!pq.isEmpty()){

int[] cur = pq.poll();

int node = cur[0];

int dist = cur[1];

int stop = cur[2];

if (stop > K + 1 || distMap.getOrDefault(stop \* 1000 + node, Integer.MAX\_VALUE) < dist){

continue;

}

if (node == dst){

return dist;

}

Map<Integer, Integer> sourceMap = graph.get(node);

if (sourceMap == null){

continue;

}

for (Map.Entry<Integer, Integer> entry : sourceMap.entrySet()){

int absoluteDist = dist + entry.getValue();

int next = entry.getKey();

int key = (stop+1) \* 1000 + next;

if (!distMap.containsKey(key) || distMap.get(key) > absoluteDist){

distMap.put(key, absoluteDist);

pq.offer(new int[]{next, absoluteDist, stop+1});

}

}

}

return -1;

}

}

Better version:

class Solution {

public int findCheapestPrice(int n, int[][] flights, int src, int dst, int K) {

Map<Integer, Map<Integer, Integer>> graph = new HashMap<>();

for(int[] flight : flights){

if (!graph.containsKey(flight[0])){

graph.put(flight[0], new HashMap<>());

}

graph.get(flight[0]).put(flight[1], flight[2]);

}

Queue<int[]> pq = new PriorityQueue<int[]>(new Comparator<int[]>(){

public int compare (int[] f1, int[] f2){

return f1[1] - f2[1];

}

});//int[]{node, shortest, stop}

Map<Integer, Integer> distMap = new HashMap<>();//key: node \* n + stop, val: shortest dist

distMap.put(src \* n, 0);

pq.offer(new int[]{src, 0, -1});

while (!pq.isEmpty()){

int[] curr = pq.poll();

int node = curr[0];

int dist = curr[1];

int stop = curr[2];

if (node == dst && stop <= K){

return dist;

}

if (!graph.containsKey(node)){

continue;

}

Map<Integer, Integer> map = graph.get(node);

for (int nei : map.keySet()){

int key = nei \* n + (stop + 1);

if ((!distMap.containsKey(key) || dist + map.get(nei) < distMap.get(key)) && stop + 1 <= K){

distMap.put(key, dist + map.get(nei));

pq.offer(new int[]{nei, dist + map.get(nei), stop + 1});

}

}

}

return -1;

}

}

# 788\_Rotated.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

X is a good number if after rotating each digit individually by 180 degrees, we get a valid number that is different from X.

Each digit must be rotated - we cannot choose to leave it alone.

A number is valid if each digit remains a digit after rotation. 0, 1, and 8 rotate to themselves; 2 and 5 rotate to each other;

6 and 9 rotate to each other, and the rest of the numbers do not rotate to any other number and become invalid.

Now given a positive number N, how many numbers X from 1 to N are good?

Example:

Input: 10

Output: 4

Explanation:

There are four good numbers in the range [1, 10] : 2, 5, 6, 9.

Note that 1 and 10 are not good numbers, since they remain unchanged after rotating.

Note:

N will be in range [1, 10000].

Method 1: Brute Force (32ms)

class Solution {

public int rotatedDigits(int N) {

int ans = 0;

for (int i = 1; i <= N; i++){

String s = String.valueOf(i);

if ((s.indexOf("2") != -1 || s.indexOf("5") != -1 || s.indexOf("6") != -1 || s.indexOf("9") != -1)

&& (s.indexOf("3") == -1 && s.indexOf("4") == -1 && s.indexOf("7") == -1)){

ans++;

}

}

return ans;

}

}

class Solution {

public int rotatedDigits(int N) {

int ans = 0;

for (int i = 1; i <= N; i++){

String s = String.valueOf(i);

if ((s.contains("2") || s.contains("5") || s.contains("6") || s.contains("9"))

&& (!s.contains("3") && !s.contains("4") && !s.contains("7"))){

ans++;

}

}

return ans;

}

}

class Solution {

public int rotatedDigits(int N) {

// Count how many n in [1, N] are good.

int ans = 0;

for (int n = 1; n <= N; ++n)

if (good(n, false)) ans++;

return ans;

}

// Return true if n is good.

// The flag is true iff we have an occurrence of 2, 5, 6, 9.

public boolean good(int n, boolean flag) {

if (n == 0) return flag;

int d = n % 10;

if (d == 3 || d == 4 || d == 7) return false;

if (d == 0 || d == 1 || d == 8) return good(n / 10, flag);

return good(n / 10, true);

}

}

Method 2: DP (9ms) Better solution than brute force

class Solution {

public int rotatedDigits(int N) {

int ans = 0;

int[] dp = new int[N+1];

for (int i = 0; i <= N; i++){

if (i < 10){

if (i == 0 || i == 1 || i == 8){

dp[i] = 1;

}else if (i == 2 || i == 5 || i == 6 || i == 9){

dp[i] = 2;

ans++;

}

}else{

int a = dp[i/10];

int b = dp[i%10];

if (a == 1 && b == 1){

dp[i] = 1;

}else if (a >= 1 && b >= 1){

dp[i] = 2;

ans++;

}

}

}

return ans;

}

}

# 791\_CustomSortString.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

S and T are strings composed of lowercase letters. In S, no letter occurs more than once.

S was sorted in some custom order previously. We want to permute the characters of T so that they

match the order that S was sorted. More specifically, if x occurs before y in S, then x should occur

before y in the returned string.

Return any permutation of T (as a string) that satisfies this property.

Example :

Input:

S = "cba"

T = "abcd"

Output: "cbad"

Explanation:

"a", "b", "c" appear in S, so the order of "a", "b", "c" should be "c", "b", and "a".

Since "d" does not appear in S, it can be at any position in T. "dcba", "cdba", "cbda" are also valid

Time complexity:

Time complexity: O(n\*mlogm)

Space complexity: O(m)

class Solution {

class Pair{

char c;

int val;

public Pair(char c, int val){

this.c = c;

this.val = val;

}

}

public String customSortString(String S, String T) {

Map<Character, Integer> map = new HashMap<>();

for (int i = 0; i < S.length(); i++){

map.put(S.charAt(i), i);

}

StringBuilder sb = new StringBuilder();

Queue<Pair> pq = new PriorityQueue<Pair>(new Comparator<Pair>(){

public int compare(Pair a, Pair b){

return a.val - b.val;

}

});

for (int i = 0; i < T.length(); i++){

char c = T.charAt(i);

pq.offer(new Pair(c, map.getOrDefault(c, 26)));

}

while (!pq.isEmpty()){

sb.append(pq.poll().c);

}

return sb.toString();

}

}

Method 2:

Time complexity: O(mn)

Space complexity: O(m)

class Solution {

public String customSortString(String S, String T) {

Set<Character> set = new HashSet<>();

StringBuilder sb = new StringBuilder();

for (char c : S.toCharArray()){

for (int i = 0; i < T.length(); i++){

if (T.charAt(i) == c){

sb.append(c);

}

}

set.add(c);

}

for (char c : T.toCharArray()){

if (!set.contains(c)){

sb.append(c);

}

}

return sb.toString();

}

}

## 792\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given string S and a dictionary of words words, find the number of words[i] that is a subsequence of S.

Example :

Input:

S = "abcde"

words = ["a", "bb", "acd", "ace"]

Output: 3

Explanation: There are three words in words that are a subsequence of S: "a", "acd", "ace".

Note:

All words in words and S will only consists of lowercase letters.

The length of S will be in the range of [1, 50000].

The length of words will be in the range of [1, 5000].

The length of words[i] will be in the range of [1, 50].

Method 1: Brute Force (TLE)

class Solution {

public int numMatchingSubseq(String S, String[] words) {

int ans = 0;

for (int i = 0; i < words.length; i++){

if (isSubsequence(S, words[i])){

ans++;

}

}

return ans;

}

private boolean isSubsequence(String S, String T){

int indexS = 0;

int indexT = 0;

while (indexS < S.length()){

if (S.charAt(indexS) == T.charAt(indexT)){

indexT++;

if (indexT == T.length()){

return true;

}

}

indexS++;

}

return false;

}

}

Passed solution:

Time complexity: O(n \* m \* L) : n: length of words, m : S length, L: average word length

class Solution {

public int numMatchingSubseq(String S, String[] words) {

int count = 0;

for (String word : words){

if (isSub(word, S)){

count++;

}

}

return count;

}

private boolean isSub(String T, String S){

int prev = 0;

int index = 0;

while (index < T.length()){

prev = S.indexOf(T.charAt(index), prev);

if (prev < 0){

return false;

}

prev++;

index++;

}

return true;

}

}

Method 3: Accepted (174ms)

Refer to explanation:https://leetcode.com/problems/number-of-matching-subsequences/discuss/117634/Efficient-and-simple-go-through-words-in-parallel-with-explanation

The same time complexity as above

class Solution {

public int numMatchingSubseq(String S, String[] words) {

Map<Character, LinkedList<String>> map = new HashMap<>();

for (String word : words){

char c = word.charAt(0);

if (!map.containsKey(c)){

map.put(c, new LinkedList<String>());

}

map.get(c).offer(word);

}

int ans = 0;

for (int i = 0; i < S.length(); i++){

char c = S.charAt(i);

LinkedList<String> queue = map.get(c);

if (queue == null){

continue;

}

int size = queue.size();

for (int j = 0; j < size; j++){

String str = queue.poll();

if (str.length() == 1){

ans++;

}else{

if (!map.containsKey(str.charAt(1))){

map.put(str.charAt(1), new LinkedList<String>());

}

map.get(str.charAt(1)).offer(str.substring(1));

}

}

}

return ans;

}

}

## 793\_Preimage.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Let f(x) be the number of zeroes at the end of x!. (Recall that x! = 1 \* 2 \* 3 \* ... \* x, and by convention, 0! = 1.)

For example, f(3) = 0 because 3! = 6 has no zeroes at the end, while f(11) = 2 because 11! = 39916800 has 2 zeroes at the end. Given K, find how many non-negative integers x have the property that f(x) = K.

Example 1:

Input: K = 0

Output: 5

Explanation: 0!, 1!, 2!, 3!, and 4! end with K = 0 zeroes.

Example 2:

Input: K = 5

Output: 0

Explanation: There is no x such that x! ends in K = 5 zeroes.

Note:

K will be an integer in the range [0, 10^9].

Based on Factorial Trailing Zeroes https://leetcode.com/problems/factorial-trailing-zeroes/description/

https://leetcode.com/problems/preimage-size-of-factorial-zeroes-function/discuss/117821/Four-binary-search-solutions-based-on-different-ideas

class Solution {

public int preimageSizeFZF(int K) {

return binarySearch(K) - binarySearch(K-1);

}

private int binarySearch(int K){

long low = 0;

long high = 5L \* (K+1);

while (low <= high){

long mid = low + (high - low) / 2;

int numZeros = trailingZeroes(mid);

if (numZeros <= K){

low = mid + 1;

}else{

high = mid - 1;

}

}

return (int)low;

}

private int trailingZeroes(long n){

int ans = 0;

while (n >= 5){

n /= 5;

ans += n;

}

return ans;

}

}

## 794\_Valid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A Tic-Tac-Toe board is given as a string array board. Return True if and only if it is possible to reach this board position during the course of a valid tic-tac-toe game.

The board is a 3 x 3 array, and consists of characters " ", "X", and "O". The " " character represents an empty square.

Here are the rules of Tic-Tac-Toe:

Players take turns placing characters into empty squares (" ").

The first player always places "X" characters, while the second player always places "O" characters.

"X" and "O" characters are always placed into empty squares, never filled ones.

The game ends when there are 3 of the same (non-empty) character filling any row, column, or diagonal.

The game also ends if all squares are non-empty.

No more moves can be played if the game is over.

Example 1:

Input: board = ["O ", " ", " "]

Output: false

Explanation: The first player always plays "X".

Example 2:

Input: board = ["XOX", " X ", " "]

Output: false

Explanation: Players take turns making moves.

Example 3:

Input: board = ["XXX", " ", "OOO"]

Output: false

Example 4:

Input: board = ["XOX", "O O", "XOX"]

Output: true

Note:

board is a length-3 array of strings, where each string board[i] has length 3.

Each board[i][j] is a character in the set {" ", "X", "O"}.

class Solution {

public boolean validTicTacToe(String[] board) {

int n = board.length;

int[] rows = new int[n];

int[] cols = new int[n];

int diag = 0;

int anti = 0;

int turns = 0;

for (int i = 0; i < n; i++){

for (int j = 0; j < n; j++){

if (board[i].charAt(j) == 'X'){

turns++;

rows[i]++;

cols[j]++;

if (i==j){

diag++;

}

if (i == n - 1 - j){

anti++;

}

}else if (board[i].charAt(j) == 'O'){

turns--;

rows[i]--;

cols[j]--;

if (i==j){

diag--;

}

if (i== n - 1 - j){

anti--;

}

}

}

}

boolean xwin = false;

boolean owin = false;

for (int i = 0; i < n; i++){

if (rows[i] == 3){

xwin = true;

}

if (rows[i] == -3){

owin = true;

}

}

for (int j = 0; j < n; j++){

if (cols[j] == 3){

xwin = true;

}

if (cols[j] == -3){

owin = true;

}

}

if (diag == 3 || anti == 3){

xwin = true;

}

if (diag == -3 || anti == -3){

owin = true;

}

if (xwin && turns == 0 || owin && turns == 1){

return false;

}

return (turns == 0 || turns == 1) && (!xwin || !owin);

}

}

## 795\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given an array A of positive integers, and two positive integers L and R (L <= R).

Return the number of (contiguous, non-empty) subarrays such that the value of the maximum array element in that subarray

is at least L and at most R.

Example :

Input:

A = [2, 1, 4, 3]

L = 2

R = 3

Output: 3

Explanation: There are three subarrays that meet the requirements: [2], [2, 1], [3].

Note:

L, R and A[i] will be an integer in the range [0, 10^9].

The length of A will be in the range of [1, 50000].

class Solution {

public int numSubarrayBoundedMax(int[] A, int L, int R) {

int res = 0;

int count = 0;

int prevInd = 0;

for (int i = 0; i < A.length; i++){

if (A[i] >= L && A[i] <= R){

count = i - prevInd + 1 ;

}else if (A[i] > R){

count = 0;

prevInd = i + 1;

}

res += count;

}

return res;

}

}

class Solution {

public int numSubarrayBoundedMax(int[] A, int L, int R) {

int res = 0;

int prevInd = 0;

int count = 0;

for (int i = 0; i < A.length; i++){

if (A[i] >= L && A[i] <= R){

count = i - prevInd + 1;

res += count;

}else if (A[i] < L){//note that the incremental does not change

res += count;

}else{

count = 0;

prevInd = i + 1;

}

}

return res;

}

}

## 796\_Rotate.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given two strings, A and B.

A shift on A consists of taking string A and moving the leftmost character to the rightmost position. For example, if A = 'abcde', then it will be 'bcdea' after one shift on A. Return True if and only if A can become B after some number of shifts on A.

Example 1:

Input: A = 'abcde', B = 'cdeab'

Output: true

Example 2:

Input: A = 'abcde', B = 'abced'

Output: false

Note:

A and B will have length at most 100.

Method 1:

Time complexity: O(AB)

class Solution {

public boolean rotateString(String A, String B) {

return A.length() == B.length() && (A+A).contains(B);

}

}

Best solution

class Solution {

public boolean rotateString(String A, String B) {

if (A.length() == 0 && B.length() == 0){

return true;

}

if (A.length() != B.length()){

return false;

}

String newA = A + A;

int len = B.length();

for (int i = 0; i < len; i++){

String sub = newA.substring(i, i+len);

if (sub.equals(B)){

return true;

}

}

return false;

}

}

Method 2:

Time complexity: O(AB)

class Solution {

public boolean rotateString(String A, String B) {

if (A.length() == 0 && B.length() == 0){

return true;

}

if (A.length() != B.length()){

return false;

}

int idx = 0;

while (idx < A.length()){

while (idx < A.length() && A.charAt(idx) != B.charAt(0)){

idx++;

}

String rotated = A.substring(idx) + A.substring(0, idx);

if (rotated.equals(B)){

return true;

}

idx++;

}

return false;

}

}

## 797\_All.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a directed, acyclic graph of N nodes. Find all possible paths from node 0 to node N-1, and return them in any order.

The graph is given as follows: the nodes are 0, 1, ..., graph.length - 1. graph[i] is a list of all nodes j for which the edge (i, j) exists.

Example:

Input: [[1,2], [3], [3], []]

Output: [[0,1,3],[0,2,3]]

Explanation: The graph looks like this:

0--->1

| |

v v

2--->3

There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.

Note:

The number of nodes in the graph will be in the range [2, 15].

You can print different paths in any order, but you should keep the order of nodes inside one path.

class Solution {

public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

List<List<Integer>> res = new ArrayList<>();

List<Integer> path = new ArrayList<>();

path.add(0);

dfs(res, path, graph, 0, graph.length - 1);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> path, int[][] graph, int src, int dst){

if (src == dst){

res.add(new ArrayList<Integer>(path));

return;

}

for (int node : graph[src]){

path.add(node);

dfs(res, path, graph, node, dst);

path.remove(path.size() - 1);

}

}

}

class Solution {

public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

List<List<Integer>> res = new ArrayList<>();

Map<Integer, List<Integer>> graphMap = new HashMap<>();

for (int i = 0; i < graph.length; i++){

graphMap.put(i, new ArrayList<Integer>());

for (int j = 0; j < graph[i].length; j++){

graphMap.get(i).add(graph[i][j]);

}

}

List<Integer> path = new ArrayList<>();

path.add(0);

dfs(res, path, graphMap, 0, graph.length - 1);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> path, Map<Integer, List<Integer>> graphMap, int src, int dst){

if (src == dst){

res.add(new ArrayList<Integer>(path));

return;

}

List<Integer> list = graphMap.get(src);

for (int node : list){

path.add(node);

dfs(res, path, graphMap, node, dst);

path.remove(path.size() - 1);

}

}

}

## 798\_Smallest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A, we may rotate it by a non-negative integer K so that the array becomes A[K], A[K+1], A{K+2], ... A[A.length - 1], A[0], A[1], ..., A[K-1]. Afterward, any entries that are less than or equal to their index are worth 1 point.

For example, if we have [2, 4, 1, 3, 0], and we rotate by K = 2, it becomes [1, 3, 0, 2, 4]. This is worth 3 points because 1 > 0 [no points], 3 > 1 [no points], 0 <= 2 [one point], 2 <= 3 [one point], 4 <= 4 [one point].

Over all possible rotations, return the rotation index K that corresponds to the highest score we could receive. If there are multiple answers, return the smallest such index K.

Example 1:

Input: [2, 3, 1, 4, 0]

Output: 3

Explanation:

Scores for each K are listed below:

K = 0, A = [2,3,1,4,0], score 2

K = 1, A = [3,1,4,0,2], score 3

K = 2, A = [1,4,0,2,3], score 3

K = 3, A = [4,0,2,3,1], score 4

K = 4, A = [0,2,3,1,4], score 3

So we should choose K = 3, which has the highest score.

Example 2:

Input: [1, 3, 0, 2, 4]

Output: 0

Explanation: A will always have 3 points no matter how it shifts.

So we will choose the smallest K, which is 0.

Note:

A will have length at most 20000.

A[i] will be in the range [0, A.length].

Method 1: Brute Force (TLE)

Time complexity: O(n^2)

Space complexity: O(n)

class Solution {

public int bestRotation(int[] A) {

int len = A.length;

int max = Integer.MIN\_VALUE;

int ans = -1;

for (int i = 0; i < len; i++){

int[] temp = new int[len];

for (int j = 0; j < len; j++){

temp[j] = A[(j+i) % len];

}

int count = 0;

for (int k = 0; k < len; k++){

if (temp[k] <= k){

count++;

}

}

if (count > max){

ans = i;

max = count;

}

}

return ans;

}

}

Method 2: O(n)

https://leetcode.com/problems/smallest-rotation-with-highest-score/discuss/118725/C%2B%2BJavaPython-Solution-with-Explanation

Key note: Don't calculate the absolute score, we can calculate the relative score compared to K = 0

Get point

Each time when we rotate, we make index 0 to index N-1, then we get one more point.

We know that for sure, so I don't need to record it.

Lose point

(i - A[i] + N) % N is the value of K making A[i]'s index just equal to A[i].

For example, If A[6] = 1, then K = (6 - A[6]) % 6 = 5 making A[6] to index 1 of new array.

So when K=5, we get this point for A[6]

Then if K is bigger when K = (i - A[i] + 1) % N, we start to lose this point, making our score -= 1

All I have done is record the value of K for all A[i] where we will lose points.

A[i]=0

Rotation makes no change for it, becasue we alwars have 0 <= index.

However, it is covered in "get point" and "lose point".

class Solution {

public int bestRotation(int[] A) {

int n = A.length;

int[] loss = new int[n];

for (int i = 0; i < n; i++){

loss[(i-A[i]+n+1)%n] -= 1;

}

int max\_i = 0;

for (int i = 1; i < n; i++){

loss[i] += loss[i-1] + 1;

if (loss[i] > loss[max\_i]){

max\_i = i;

}

}

return max\_i;

}

}

## 799\_Champagne.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We stack glasses in a pyramid, where the first row has 1 glass, the second row has 2 glasses, and so on until the 100th row. Each glass holds one cup (250ml) of champagne.

Then, some champagne is poured in the first glass at the top. When the top most glass is full, any excess liquid poured will fall equally to the glass immediately to the left and right of it. When those glasses become full, any excess champagne will fall equally to the left and right of those glasses, and so on. (A glass at the bottom row has it's excess champagne fall on the floor.)

For example, after one cup of champagne is poured, the top most glass is full. After two cups of champagne are poured, the two glasses on the second row are half full. After three cups of champagne are poured, those two cups become full - there are 3 full glasses total now. After four cups of champagne are poured, the third row has the middle glass half full, and the two outside glasses are a quarter full, as pictured below.

Method 1:

In general, if a glass has flow-through X, then Q = (X - 1.0) / 2.0 quantity of champagne will equally flow left and right. We can simulate the entire pour for 100 rows of glasses. A glass at (r, c) will have excess champagne flow towards (r+1, c) and (r+1, c+1).

class Solution {

public double champagneTower(int poured, int query\_row, int query\_glass) {

double[][] dp = new double[101][101];

dp[0][0] = poured;

for (int i = 0; i < query\_row; i++){

for (int j = 0; j <= i; j++){

double q = (dp[i][j] - 1) / 2.0;

if (q > 0){

dp[i+1][j] += q;

dp[i+1][j+1] += q;

}

}

}

return Math.min(1.0, dp[query\_row][query\_glass]);

}

}

class Solution {

public double champagneTower(int poured, int query\_row, int query\_glass) {

double[][] dp = new double[101][101];

dp[0][0] = poured;

for (int i = 0; i < 100; i++){

for (int j = 0; j <= i; j++){

double extra = (dp[i][j] - 1) / 2;

if (extra > 0){

dp[i+1][j] += extra;

dp[i+1][j+1] += extra;

dp[i][j] = 1;

}

}

}

return dp[query\_row][query\_glass];

}

}

## 8\_StringtoInteger.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

8. String to Integer (atoi)

Implement atoi to convert a string to an integer.

class Solution {

public int myAtoi(String str) {

if (str == null){

return 0;

}

str = str.trim();

int i = 0;

char flag = '+';

if (str.length() == 0){

return 0;

}

if (str.charAt(0) == '-'){

flag = '-';

i++;

}else if (str.charAt(0) == '+'){

i++;

}

double res = 0;

while (i < str.length() && str.charAt(i) >= '0' && str.charAt(i) <= '9'){

res = res \* 10 + (int)(str.charAt(i) - '0');

i++;

}

if (flag == '-'){

res = -res;

}

if (res < Integer.MIN\_VALUE){

res = (int) Integer.MIN\_VALUE;

}

if (res > Integer.MAX\_VALUE){

res = (int) Integer.MAX\_VALUE;

}

return (int)res;

}

}

## 80\_RemoveDuplicatesII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Follow up for "Remove Duplicates":

What if duplicates are allowed at most twice?

For example,

Given sorted array nums = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of nums being 1, 1, 2, 2 and 3.

It doesn't matter what you leave beyond the new length.

class Solution {

public int removeDuplicates(int[] nums) {

int index = 0;

for (int i = 0; i < nums.length; i++){

if (index < 2 || nums[index-2] < nums[i]){

nums[index++] = nums[i];

}

}

return index;

}

}

class Solution {

public int removeDuplicates(int[] nums) {

int index = 0;

for (int i = 0; i < nums.length; i++){

if (index < 2 || nums[index-2] != nums[i]){

nums[index++] = nums[i];

}

}

return index;

}

}

## 800\_Similar.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In the following, every capital letter represents some hexadecimal digit from 0 to f.

The red-green-blue color "#AABBCC" can be written as "#ABC" in shorthand. For example, "#15c" is shorthand for the color "#1155cc".

Now, say the similarity between two colors "#ABCDEF" and "#UVWXYZ" is -(AB - UV)^2 - (CD - WX)^2 - (EF - YZ)^2.

Given the color "#ABCDEF", return a 7 character color that is most similar to #ABCDEF, and has a shorthand (that is, it can be

represented as some "#XYZ"

Example 1:

Input: color = "#09f166"

Output: "#11ee66"

Explanation:

The similarity is -(0x09 - 0x11)^2 -(0xf1 - 0xee)^2 - (0x66 - 0x66)^2 = -64 -9 -0 = -73.

This is the highest among any shorthand color.

Note:

color is a string of length 7.

color is a valid RGB color: for i > 0, color[i] is a hexadecimal digit from 0 to f

Any answer which has the same (highest) similarity as the best answer will be accepted.

All inputs and outputs should use lowercase letters, and the output is 7 characters.

class Solution {

public String similarRGB(String color) {

StringBuilder sb = new StringBuilder();

sb.append("#");

for (int i = 1; i < color.length(); i += 2){

String sub = color.substring(i, i+2);

String min = getSimilar(sub);

sb.append(min);

}

return sb.toString();

}

private String getSimilar(String s){

if (s.charAt(0) == s.charAt(1)){

return s;

}

String ans = "";

int num = Integer.parseInt(s, 16); //convert 16 hex to 10 decimal

char[] chars = {'f','0', '1', '2', '3', '4', '5','6','7','8','9','a','b','c','d','e','f', '0'};

for (int i = 1; i < chars.length-1; i++){

char c = s.charAt(0);

int min = Integer.MAX\_VALUE;

if (c == chars[i]){

if (Math.abs(num - Integer.parseInt(chars[i+1] + "" + chars[i+1], 16)) < min){

ans = chars[i+1] + "" + chars[i+1];

min = Math.abs(num - Integer.parseInt(ans, 16));

}

if (Math.abs(num - Integer.parseInt(chars[i] + "" + chars[i], 16)) < min){

ans = chars[i] + "" + chars[i];

min = Math.abs(num - Integer.parseInt(ans, 16));

}

if (Math.abs(num - Integer.parseInt(chars[i-1] + "" + chars[i-1], 16)) < min){

ans = chars[i-1] + "" + chars[i-1];

min = Math.abs(num - Integer.parseInt(ans, 16));

}

break;

}

}

return ans;

}

}

## 801\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have two integer sequences A and B of the same non-zero length.

We are allowed to swap elements A[i] and B[i]. Note that both elements are in the same index position in their respective sequences.

At the end of some number of swaps, A and B are both strictly increasing. (A sequence is strictly increasing if and only if A[0] < A[1] < A[2] < ... < A[A.length - 1].)

Given A and B, return the minimum number of swaps to make both sequences strictly increasing. It is guaranteed that the given input always makes it possible.

Example:

Input: A = [1,3,5,4], B = [1,2,3,7]

Output: 1

Explanation:

Swap A[3] and B[3]. Then the sequences are:

A = [1, 3, 5, 7] and B = [1, 2, 3, 4]

which are both strictly increasing.

Note:

A, B are arrays with the same length, and that length will be in the range [1, 1000].

A[i], B[i] are integer values in the range [0, 2000].

This is two alternate 1D dp

Intuition

The cost of making both sequences increasing up to the first i columns can be expressed in terms of the cost of making both sequences

increasing up to the first i-1 columns. This is because the only thing that matters to the ith column is whether the previous column

was swapped or not. This makes dynamic programming an ideal choice.

Let's remember n1 (natural1), the cost of making the first i-1 columns increasing and not swapping the i-1th column; and s1 (swapped1),

the cost of making the first i-1 columns increasing and swapping the i-1th column.

Now we want candidates n2 (and s2), the costs of making the first i columns increasing if we do not swap (or swap, respectively)

the ith column.

Algorithm

For convenience, say a1 = A[i-1], b1 = B[i-1] and a2 = A[i], b2 = B[i].

Now, if a1 < a2 and b1 < b2, then it is allowed to have both of these columns natural (unswapped), or both of these columns swapped.

This possibility leads to n2 = min(n2, n1) and s2 = min(s2, s1 + 1).

Another, (not exclusive) possibility is that a1 < b2 and b1 < a2. This means that it is allowed to have exactly one of these columns

swapped. This possibility leads to n2 = min(n2, s1) or s2 = min(s2, n1 + 1).

Note that it is important to use two if statements separately, because both of the above possibilities might be possible.

At the end, the optimal solution must leave the last column either natural or swapped, so we take the minimum number of swaps between

the two possibilities.

Method 1:

class Solution {

public int minSwap(int[] A, int[] B) {

int swapPrev = 1;

int normPrev = 0;

for (int i = 1; i < A.length; i++){

int normCurr = Integer.MAX\_VALUE;

int swapCurr = Integer.MAX\_VALUE;

if (A[i] > A[i-1] && B[i] > B[i-1]){

normCurr = Math.min(normCurr, normPrev);

swapCurr = Math.min(swapCurr, swapPrev + 1);

}

if (A[i] > B[i-1] && B[i] > A[i-1]){

normCurr = Math.min(normCurr, swapPrev);

swapCurr = Math.min(swapCurr, normPrev + 1);

}

normPrev = normCurr;

swapPrev = swapCurr;

}

return Math.min(normPrev, swapPrev);

}

}

Method 2:

https://leetcode.com/problems/minimum-swaps-to-make-sequences-increasing/discuss/119835/Java-O(n)-DP-Solution

class Solution {

public int minSwap(int[] A, int[] B) {

int n = A.length;

int[] swap = new int[n];

int[] norm = new int[n];

swap[0] = 1;

norm[0] = 0;

for (int i = 1; i < n; i++){

if (A[i-1] >= B[i] || B[i-1] >= A[i]){ //the same operation for previous and current

norm[i] = norm[i-1];

swap[i] = swap[i-1] + 1;

}else if (A[i-1] >= A[i] || B[i-1] >= B[i]){ //the opposite operation for prevoius and current

norm[i] = swap[i-1];

swap[i] = norm[i-1] + 1;

}else{ //either swap or fix is okay

int min = Math.min(norm[i-1], swap[i-1]);

norm[i] = min;

swap[i] = min + 1;

}

}

return Math.min(norm[n-1], swap[n-1]);

}

}

Similar as Wiggle Subsequence (Alternate DP) https://github.com/optimisea/Leetcode/blob/master/Java/376\_Wiggle.java

Optimize from O(n) to O(1)

Notice that every ith swapRecord and fixRecord is only relevant with the previous one.

So the algorithm should be optimized to an O(1) space version.

class Solution {

public int minSwap(int[] A, int[] B) {

int n = A.length;

int swap = 1;

int norm = 0;

for (int i = 1; i < n; i++){

if (A[i-1] >= B[i] || B[i-1] >= A[i]){

swap++;

}else if (A[i-1] >= A[i] || B[i-1] >= B[i]){

int temp = swap;

swap = norm + 1;

norm = temp;

}else{

int min = Math.min(norm, swap);

norm = min;

swap = min + 1;

}

}

return Math.min(norm, swap);

}

}

## 802\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a directed graph, we start at some node and every turn, walk along a directed edge of the graph. If we reach a node that is

terminal (that is, it has no outgoing directed edges), we stop.

Now, say our starting node is eventually safe if and only if we must eventually walk to a terminal node. More specifically, there

exists a natural number K so that for any choice of where to walk, we must have stopped at a terminal node in less than K steps.

Which nodes are eventually safe? Return them as an array in sorted order.

The directed graph has N nodes with labels 0, 1, ..., N-1, where N is the length of graph. The graph is given in the following

form: graph[i] is a list of labels j such that (i, j) is a directed edge of the graph.

Example:

Input: graph = [[1,2],[2,3],[5],[0],[5],[],[]]

Output: [2,4,5,6]

Here is a diagram of the above graph.

Illustration of graph

Note:

graph will have length at most 10000.

The number of edges in the graph will not exceed 32000.

Each graph[i] will be a sorted list of different integers, chosen within the range [0, graph.length - 1].

Intuition

As in Approach #1, the crux of the problem is whether you reach a cycle or not.

Let us perform a "brute force": a cycle-finding DFS algorithm on each node individually.

This is a classic "white-gray-black" DFS algorithm that would be part of any textbook on DFS. We mark a node gray on entry,

and black on exit. If we see a gray node during our DFS, it must be part of a cycle. In a naive view,

we'll clear the colors between each search.

Algorithm

We can improve this approach, by noticing that we don't need to clear the colors between each search.

When we visit a node, the only possibilities are that we've marked the entire subtree black (which must be eventually safe),

or it has a cycle and we have only marked the members of that cycle gray. So indeed, the invariant that gray nodes are always

part of a cycle, and black nodes are always eventually safe is maintained.

In order to exit our search quickly when we find a cycle (and not paint other nodes erroneously), we'll say the result of

visiting a node is true if it is eventually safe, otherwise false. This allows information that we've reached a cycle to

propagate up the call stack so that we can terminate our search early.

// colors:

WHITE 0: unvisited

GRAY 1: start, unsafe

BLACK 2: safe

class Solution {

public List<Integer> eventualSafeNodes(int[][] graph) {

List<Integer> res = new ArrayList<>();

int n = graph.length;

int[] colors = new int[n];

for (int i = 0; i < n; i++){

if (isSafe(graph, colors, i)){//the visited node is eventually safe if it is true

res.add(i);

}

}

return res;

}

private boolean isSafe(int[][] graph, int[] colors, int start){

if (colors[start] > 0){

return colors[start] == 2;

}

colors[start] = 1;

for (int nei : graph[start]){

if (!isSafe(graph, colors, nei)){

return false;

}

}

colors[start] = 2;

return true;

}

}

## 804\_Unique.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

International Morse Code defines a standard encoding where each letter is mapped to a series of dots and dashes, as follows: "a" maps to ".-", "b" maps to "-...", "c" maps to "-.-.", and so on.

For convenience, the full table for the 26 letters of the English alphabet is given below:

[".-","-...","-.-.","-..",".","..-.","--.","....","..",".---","-.-",".-..","--","-.","---",".--.","--.-",".-.","...","-","..-","...-",".--","-..-","-.--","--.."]

Now, given a list of words, each word can be written as a concatenation of the Morse code of each letter. For example, "cab" can be written as "-.-.-....-", (which is the concatenation "-.-." + "-..." + ".-"). We'll call such a concatenation, the transformation of a word.

Return the number of different transformations among all words we have.

Example:

Input: words = ["gin", "zen", "gig", "msg"]

Output: 2

Explanation:

The transformation of each word is:

"gin" -> "--...-."

"zen" -> "--...-."

"gig" -> "--...--."

"msg" -> "--...--."

There are 2 different transformations, "--...-." and "--...--.".

Note:

The length of words will be at most 100.

Each words[i] will have length in range [1, 12].

words[i] will only consist of lowercase letters.

class Solution {

public int uniqueMorseRepresentations(String[] words) {

Set<String> set = new HashSet<>();

for (String word : words){

String s = transform(word);

if (!set.contains(s)){

set.add(s);

}

}

return set.size();

}

private String transform(String word){

String[] morse = {".-","-...","-.-.","-..",".","..-.","--.","....","..",".---","-.-",".-..","--","-.","---",".--.","--.-",".-.","...","-","..-","...-",".--","-..-","-.--","--.."};

StringBuilder sb = new StringBuilder();

for (int i = 0; i < word.length(); i++){

sb.append(morse[word.charAt(i) - 'a']);

}

return sb.toString();

}

}

## 805\_MaximumAssociationSet.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Amazon sells books, every book has books which are strongly associated with it.

Given ListA and ListB,indicates that ListA [i] is associated with ListB [i] which represents

the book and associated books. Output the largest set associated with each other(output in any sort).

You can assume that there is only one of the largest set.

Notice

The number of books does not exceed 5000.

Have you met this question in a real interview?

Example

Given ListA = ["abc","abc","abc"], ListB = ["bcd","acd","def"], return["abc","acd","bcd","dfe"].

Explanation:

abc is associated with bcd, acd, dfe, so the largest set is the set of all books

Given ListA = ["a","b","d","e","f"], ListB = ["b","c","e","g","g"], return ["d","e","f","g"].

Explanation:

The current set are [a, b, c] and [d, e, g, f], then the largest set is [d, e, g, f]

public class Solution {

/\*\*

\* @param ListA: The relation between ListB's books

\* @param ListB: The relation between ListA's books

\* @return: The answer

\*/

public List<String> maximumAssociationSet(String[] ListA, String[] ListB) {

Map<String,Set<String>> map = new HashMap<>();

for(int i=0;i<ListA.length;i++){

if(!map.containsKey(ListA[i])){

map.put(ListA[i],new HashSet<>());

}

if(!map.containsKey(ListB[i])){

map.put(ListB[i],new HashSet<>());

}

if(!map.get(ListA[i]).contains(ListB[i]))

map.get(ListA[i]).add(ListB[i]);

if(!map.get(ListB[i]).contains(ListA[i]))

map.get(ListB[i]).add(ListA[i]);

}

List<String> res = new ArrayList<>();

Set<String> cur = new HashSet<>();

for(String str:ListA){

if(map.containsKey(str)) {

helper(cur, str, map);

if (cur.size() > res.size()) {

res.clear();

res.addAll(cur);

}

cur.clear();

}

}

return res;

}

private static void helper(Set<String> cur, String str, Map<String,Set<String>> map){

cur.add(str);

if(map.containsKey(str)) {

Set<String> next = map.get(str);

map.remove(str);

for(String s:next){

helper(cur,s,map);

}

}

}

}

## 805\_Split.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a given integer array A, we must move every element of A to either list B or list C. (B and C initially start empty.)

Return true if and only if after such a move, it is possible that the average value of B is equal to the average value of C, and B and C are both non-empty.

Example :

Input:

[1,2,3,4,5,6,7,8]

Output: true

Explanation: We can split the array into [1,4,5,8] and [2,3,6,7], and both of them have the average of 4.5.

Note:

The length of A will be in the range [1, 30].

A[i] will be in the range of [0, 10000].

https://leetcode.com/problems/split-array-with-same-average/discuss/120660/Java-accepted-recursive-solution-with-explanation

Key point: avgA = avgB = avgC

The goal is to find array B that the average of B equals to avg A.

So it will be converted to N sum problem: find lenB that sum of lenB equals to sum/n

Since only split to two arrays, so we try all the possibility of lenB from 1 to n/2

class Solution {

public boolean splitArraySameAverage(int[] A) {

int sum = 0;

for (int i : A){

sum += i;

}

Arrays.sort(A);//use sort to prune to accelerate recursion

int n = A.length;

for (int i = 1; i <= n / 2; i++){

if ((sum \* i) % n == 0){

if (canSplit(A, sum \* i / n, i, 0)){

return true;

}

}

}

return false;

}

private boolean canSplit(int[] A, int target, int len, int start){

if (len == 0){

return target == 0;

}

for (int i = start; i + len <= A.length; i++){//note that i + len < A.length is wrong because A[i] is not taken yet

if (A[start] \* len > target){

break;//or return false;

}

if (i > start && A[i] == A[i-1]){

continue;

}

if (canSplit(A, target - A[i], len - 1, i + 1)){

return true;

}

}

return false;

}

}

## 806\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are to write the letters of a given string S, from left to right into lines. Each line has maximum width 100 units, and if writing a letter would cause the width of the line to exceed 100 units, it is written on the next line. We are given an array widths, an array where widths[0] is the width of 'a', widths[1] is the width of 'b', ..., and widths[25] is the width of 'z'.

Now answer two questions: how many lines have at least one character from S, and what is the width used by the last such line? Return your answer as an integer list of length 2.

Example :

Input:

widths = [10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10]

S = "abcdefghijklmnopqrstuvwxyz"

Output: [3, 60]

Explanation:

All letters have the same length of 10. To write all 26 letters,

we need two full lines and one line with 60 units.

Example :

Input:

widths = [4,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10]

S = "bbbcccdddaaa"

Output: [2, 4]

Explanation:

All letters except 'a' have the same length of 10, and

"bbbcccdddaa" will cover 9 \* 10 + 2 \* 4 = 98 units.

For the last 'a', it is written on the second line because

there is only 2 units left in the first line.

So the answer is 2 lines, plus 4 units in the second line.

Note:

The length of S will be in the range [1, 1000].

S will only contain lowercase letters.

widths is an array of length 26.

widths[i] will be in the range of [2, 10].

class Solution {

public int[] numberOfLines(int[] widths, String S) {

int lines = 1, width = 0;

for (char c: S.toCharArray()) {

int w = widths[c - 'a'];

width += w;

if (width > 100) {

lines++;

width = w;

}

}

return new int[]{lines, width};

}

}

class Solution {

public int[] numberOfLines(int[] widths, String S) {

int[] res = new int[2];

res[0] = 1;

int i = 0;

while (i < S.length()){

int width = widths[S.charAt(i) -'a'];

res[1] += width;

if (res[1] > 100){

res[0]++;

res[1] = width;

}

i++;

}

return res;

}

}

class Solution {

public int[] numberOfLines(int[] widths, String S) {

int line = 1;

int sum = 0;

for (char c : S.toCharArray()){

if (sum + widths[c - 'a'] <= 100){

sum += widths[c - 'a'];

}else{

sum = widths[c - 'a'];

line++;

}

}

return new int[]{line, sum};

}

}

## 807\_Max.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a 2 dimensional array grid, each value grid[i][j] represents the height of a building located there. We are allowed to increase the height of any number of buildings, by any amount (the amounts can be different for different buildings). Height 0 is considered to be a building as well.

At the end, the "skyline" when viewed from all four directions of the grid, i.e. top, bottom, left, and right, must be the same as the skyline of the original grid. A city's skyline is the outer contour of the rectangles formed by all the buildings when viewed from a distance. See the following example.

What is the maximum total sum that the height of the buildings can be increased?

Example:

Input: grid = [[3,0,8,4],[2,4,5,7],[9,2,6,3],[0,3,1,0]]

Output: 35

Explanation:

The grid is:

[ [3, 0, 8, 4],

[2, 4, 5, 7],

[9, 2, 6, 3],

[0, 3, 1, 0] ]

The skyline viewed from top or bottom is: [9, 4, 8, 7]

The skyline viewed from left or right is: [8, 7, 9, 3]

The grid after increasing the height of buildings without affecting skylines is:

gridNew = [ [8, 4, 8, 7],

[7, 4, 7, 7],

[9, 4, 8, 7],

[3, 3, 3, 3] ]

Notes:

1 < grid.length = grid[0].length <= 50.

All heights grid[i][j] are in the range [0, 100].

All buildings in grid[i][j] occupy the entire grid cell: that is, they are a 1 x 1 x grid[i][j] rectangular prism.

Time complexity: O(mn)

Space compleixty: O(m) + O(n)

class Solution {

public int maxIncreaseKeepingSkyline(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int ans = 0;

int[] rowMax = new int[m];

int[] colMax = new int[n];

for (int i = 0; i < m; i++){

int max = Integer.MIN\_VALUE;

for (int j = 0; j < n; j++){

max = Math.max(max, grid[i][j]);

}

rowMax[i] = max;

}

for (int j = 0; j < n; j++){

int max = Integer.MIN\_VALUE;

for (int i = 0; i < m; i++){

max = Math.max(max, grid[i][j]);

}

colMax[j] = max;

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

ans += Math.min(rowMax[i], colMax[j]) - grid[i][j];

}

}

return ans;

}

}

Best solution:

class Solution {

public int maxIncreaseKeepingSkyline(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int ans = 0;

int[] rowMax = new int[m];

int[] colMax = new int[n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

rowMax[i] = Math.max(rowMax[i], grid[i][j]);

colMax[j] = Math.max(colMax[j], grid[i][j]);

}

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

ans += Math.min(rowMax[i], colMax[j]) - grid[i][j];

}

}

return ans;

}

}

## 809\_Expressive.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sometimes people repeat letters to represent extra feeling, such as "hello" -> "heeellooo", "hi" -> "hiiii". Here, we have groups, of adjacent letters that are all the same character, and adjacent characters to the group are different. A group is extended if that group is length 3 or more, so "e" and "o" would be extended in the first example, and "i" would be extended in the second example. As another example, the groups of "abbcccaaaa" would be "a", "bb", "ccc", and "aaaa"; and "ccc" and "aaaa" are the extended groups of that string.

For some given string S, a query word is stretchy if it can be made to be equal to S by extending some groups. Formally, we are allowed to repeatedly choose a group (as defined above) of characters c, and add some number of the same character c to it so that the length of the group is 3 or more. Note that we cannot extend a group of size one like "h" to a group of size two like "hh" - all extensions must leave the group extended - ie., at least 3 characters long.

Given a list of query words, return the number of words that are stretchy.

Example:

Input:

S = "heeellooo"

words = ["hello", "hi", "helo"]

Output: 1

Explanation:

We can extend "e" and "o" in the word "hello" to get "heeellooo".

We can't extend "helo" to get "heeellooo" because the group "ll" is not extended.

Notes:

0 <= len(S) <= 100.

0 <= len(words) <= 100.

0 <= len(words[i]) <= 100.

S and all words in words consist only of lowercase letters

class Solution {

public int expressiveWords(String S, String[] words) {

int ans = 0;

for (String word : words){

if (isExtended(S, word)){

ans++;

}

}

return ans;

}

private boolean isExtended(String S, String word){

int i = 0;

int j = 0;

while (i < S.length() && j < word.length()){

int iStart = i;

i++;

while (i < S.length() && S.charAt(i) == S.charAt(i-1)){

i++;

}

int jStart = j;

j++;

while (j < word.length() && word.charAt(j) == word.charAt(j-1)){

j++;

}

if (S.charAt(iStart) != word.charAt(jStart)){

return false;

}

int iDiff = i - iStart;

int jDiff = j - jStart;

if (jDiff > iDiff){

return false;

}

if (jDiff < iDiff && iDiff < 3){

return false;

}

}

if (i < S.length() || j < word.length()){

return false;

}

return true;

}

}

class Solution {

public int expressiveWords(String S, String[] words) {

int ans = 0;

for (String word : words){

if (isExtended(S, word)){

ans++;

}

}

return ans;

}

private boolean isExtended(String S, String word){

int i = 0;

int j = 0;

int iStart = 0;

int jStart = 0;

while (i < S.length() && j < word.length()){

if (S.charAt(iStart) != word.charAt(jStart)){

return false;

}

i = iStart + 1;

while (i < S.length() && S.charAt(i) == S.charAt(i-1)){

i++;

}

j = jStart + 1;

while (j < word.length() && word.charAt(j) == word.charAt(j-1)){

j++;

}

int iDiff = i - iStart;

int jDiff = j - jStart;

if (jDiff > iDiff){

return false;

}

if (jDiff < iDiff && iDiff < 3){

return false;

}

iStart = i;

jStart = j;

}

if ((i < S.length()) ^ (j < word.length())){

return false;

}

return true;

}

}

## 81\_DataStream.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Numbers keep coming, return the median of numbers at every time a new number added.

Have you met this question in a real interview?

Clarification

What's the definition of Median?

- Median is the number that in the middle of a sorted array. If there are n numbers in a sorted array A,

the median is A[(n - 1) / 2]. For example, if A=[1,2,3], median is 2. If A=[1,19], median is 1.

Example

For numbers coming list: [1, 2, 3, 4, 5], return [1, 1, 2, 2, 3].

For numbers coming list: [4, 5, 1, 3, 2, 6, 0], return [4, 4, 4, 3, 3, 3, 3].

For numbers coming list: [2, 20, 100], return [2, 2, 20].

Challenge

Total run time in O(nlogn).

public class Solution {

/\*

\* @param nums: A list of integers

\* @return: the median of numbers

\*/

public int[] medianII(int[] nums) {

if (nums == null || nums.length == 0){

return null;

}

Queue<Integer> minHeap = new PriorityQueue<Integer>();

Queue<Integer> maxHeap = new PriorityQueue<Integer>(1, new Comparator<Integer>(){

public int compare(Integer a, Integer b){

return b - a;

}

});

int[] result = new int[nums.length];

result[0] = nums[0];

maxHeap.offer(nums[0]);

for (int i = 1; i < nums.length; i++){

int prevMedian = maxHeap.peek();

if (nums[i] > prevMedian){

minHeap.offer(nums[i]);

}else{

maxHeap.offer(nums[i]);

}

if (maxHeap.size() > minHeap.size() + 1){

minHeap.offer(maxHeap.poll());

}

if (maxHeap.size() < minHeap.size()){

maxHeap.offer(minHeap.poll());

}

result[i] = maxHeap.peek();

}

return result;

}

}

# 81\_searchRotated.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Follow up for "Search in Rotated Sorted Array":

What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Write a function to determine if a given target is in the array.

The array may contain duplicates.

https://leetcode.com/problems/search-in-rotated-sorted-array-ii/discuss/28218/My-8ms-C++-solution-(o(logn)-on-average-o(n)-worst-case)

class Solution {

public boolean search(int[] nums, int target) {

if (nums == null || nums.length == 0){

return false;

}

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return true;

}

if (nums[mid] > nums[start]){

if (nums[start] <= target && target < nums[mid]){

end = mid;

}else{

start = mid;

}

}else if (nums[mid] < nums[end]){

if (nums[mid] < target && target <= nums[end]){

start = mid;

}else{

end = mid;

}

}else{//this case is the only difference due to the duplicates

if (nums[start] == nums[mid]){//when eqaul, we don't know which direction to do binary

start++;

}

if (nums[end] == nums[mid]){//when eqaul, we don't know which direction to do binary

end--;

}

}

}

if (nums[start] == target || nums[end] == target){

return true;

}

return false;

}

}

Take this:

class Solution {

public boolean search(int[] nums, int target) {

if (nums == null || nums.length == 0){

return false;

}

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return true;

}

if (nums[mid] > nums[end]){

if (nums[mid] > target && target > nums[end]){

end = mid;

}else{

start = mid;

}

}else if (nums[mid] < nums[end]){

if (nums[mid] < target && target <= nums[end]){

start = mid;

}else{

end = mid;

}

}else{ //this case is the only difference due to the duplicates

if (nums[mid] == nums[start]){//when eqaul, we don't know which direction to do binary

start++;

}

if (nums[mid] == nums[end]){//when eqaul, we don't know which direction to do binary

end--;

}

}

}

if (nums[start] == target){

return true;

}else if (nums[end] == target){

return true;

}

return false;

}

}

class Solution {

public boolean search(int[] nums, int target) {

if (nums == null || nums.length == 0){

return false;

}

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return true;

}

int last = nums[end];

if (target == last){

return true;

}else if (target > last){

if (nums[mid] == last){

end--;

if (nums[mid] == nums[start]){

start++;

}

}else if (nums[mid] > last && nums[mid] < target){

start = mid;

}else{

end = mid;

}

}else {

if (nums[mid] == last){

end--;

if (nums[mid] == nums[start]){

start++;

}

}else if (nums[mid] < last && nums[mid] > target){

end = mid;

}else{

start = mid;

}

}

}

if (nums[start] == target || nums[end] == target){

return true;

}

return false;

}

}

Best solution:

class Solution {

public boolean search(int[] nums, int target) {

if (nums == null || nums.length == 0){

return false;

}

int start = 0;

int end = nums.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (nums[mid] == target){

return true;

}

if (nums[mid] > nums[end]){

if (nums[mid] > target && target > nums[end]){

end = mid;

}else{

start = mid;

}

}else if (nums[mid] < nums[end]){

if (nums[mid] < target && target <= nums[end]){

start = mid;

}else{

end = mid;

}

}else{

// if (nums[mid] == nums[start]){

// start++;

// }

// if (nums[mid] == nums[end]){

end--;

// }

}

}

if (nums[start] == target){

return true;

}else if (nums[end] == target){

return true;

}

return false;

}

}

# 810\_Chalkboard.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given non-negative integers nums[i] which are written on a chalkboard. Alice and Bob take turns erasing exactly one number from the chalkboard, with Alice starting first. If erasing a number causes the bitwise XOR of all the elements of the chalkboard to become 0, then that player loses. (Also, we'll say the bitwise XOR of one element is that element itself, and the bitwise XOR of no elements is 0.)

Also, if any player starts their turn with the bitwise XOR of all the elements of the chalkboard equal to 0, then that player wins.

Return True if and only if Alice wins the game, assuming both players play optimally.

Example:

Input: nums = [1, 1, 2]

Output: false

Explanation:

Alice has two choices: erase 1 or erase 2.

If she erases 1, the nums array becomes [1, 2]. The bitwise XOR of all the elements of the chalkboard is 1 XOR 2 = 3. Now Bob can remove any element he wants, because Alice will be the one to erase the last element and she will lose.

If Alice erases 2 first, now nums becomes [1, 1]. The bitwise XOR of all the elements of the chalkboard is 1 XOR 1 = 0. Alice will lose.

Notes:

1 <= N <= 1000.

0 <= nums[i] <= 2^16.

class Solution {

public boolean xorGame(int[] nums) {

int x = 0;

for (int i : nums){

x ^= i;

}

return x == 0 || nums.length % 2 == 0;

}

}

# 811\_Subdomain.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A website domain like "discuss.leetcode.com" consists of various subdomains. At the top level, we have "com", at the next level, we have "leetcode.com", and at the lowest level, "discuss.leetcode.com". When we visit a domain like "discuss.leetcode.com", we will also visit the parent domains "leetcode.com" and "com" implicitly.

Now, call a "count-paired domain" to be a count (representing the number of visits this domain received), followed by a space, followed by the address. An example of a count-paired domain might be "9001 discuss.leetcode.com".

We are given a list cpdomains of count-paired domains. We would like a list of count-paired domains, (in the same format as the input, and in any order), that explicitly counts the number of visits to each subdomain.

Example 1:

Input:

["9001 discuss.leetcode.com"]

Output:

["9001 discuss.leetcode.com", "9001 leetcode.com", "9001 com"]

Explanation:

We only have one website domain: "discuss.leetcode.com". As discussed above, the subdomain "leetcode.com" and "com" will also be visited. So they will all be visited 9001 times.

Example 2:

Input:

["900 google.mail.com", "50 yahoo.com", "1 intel.mail.com", "5 wiki.org"]

Output:

["901 mail.com","50 yahoo.com","900 google.mail.com","5 wiki.org","5 org","1 intel.mail.com","951 com"]

Explanation:

We will visit "google.mail.com" 900 times, "yahoo.com" 50 times, "intel.mail.com" once and "wiki.org" 5 times. For the subdomains, we will visit "mail.com" 900 + 1 = 901 times, "com" 900 + 50 + 1 = 951 times, and "org" 5 times.

Notes:

The length of cpdomains will not exceed 100.

The length of each domain name will not exceed 100.

Each address will have either 1 or 2 "." characters.

The input count in any count-paired domain will not exceed 10000.

The answer output can be returned in any order.

Method 1:

class Solution {

public List<String> subdomainVisits(String[] cpdomains) {

List<String> res = new ArrayList<>();

Map<String, Integer> map = new HashMap<>();

for (String str : cpdomains){

String[] arr = str.split("\\s+");

String subdomain = arr[1];

int num = Integer.parseInt(arr[0]);

map.put(subdomain, map.getOrDefault(subdomain, 0) + num);

int start = subdomain.lastIndexOf(".");

while (start != -1){

String sub = subdomain.substring(start+1);

map.put(sub, map.getOrDefault(sub, 0) + num);

start = subdomain.lastIndexOf(".", start-1);

}

}

for (String key : map.keySet()){

String str = map.get(key) + " " + key;

res.add(str);

}

return res;

}

}

Method 2:

note to split ".", must use str.split("\\.");

there are 12 characters with special meanings: the backslash \, the caret ^, the dollar sign $, the period or dot .,

the vertical bar or pipe symbol |, the question mark ?, the asterisk or star \*, the plus sign +, the opening parenthesis (,

the closing parenthesis ), and the opening square bracket [, the opening curly brace {, These special characters are often

called "metacharacters".

https://stackoverflow.com/questions/3481828/how-to-split-a-string-in-java

class Solution {

public List<String> subdomainVisits(String[] cpdomains) {

List<String> res = new ArrayList<>();

Map<String, Integer> map = new HashMap<>();

for (String str : cpdomains){

String[] arr = str.split("\\s+");

String subdomain = arr[1];

int num = Integer.parseInt(arr[0]);

String[] frags = subdomain.split("\\."); // can't use "."

String curr = "";

for (int i = frags.length - 1; i >= 0; i--){

curr = frags[i] + (i == frags.length - 1 ? "" : ".") + curr;

map.put(curr, map.getOrDefault(curr, 0) + num);

}

}

for (String key : map.keySet()){

String str = map.get(key) + " " + key;

res.add(str);

}

return res;

}

}

Best solution:

class Solution {

public List<String> subdomainVisits(String[] cpdomains) {

Map<String, Integer> map = new HashMap<>();

for (String domain : cpdomains){

String[] strs = domain.split(" ");

int num = Integer.parseInt(strs[0]);

String str = strs[1];

for (int i = 0 ; i < str.length(); i++){

if (str.charAt(i) == '.'){

String sub = str.substring(i+1);

map.put(sub, map.getOrDefault(sub, 0) + num);

}

}

map.put(str, map.getOrDefault(str, 0) + num);

}

List<String> res = new ArrayList<>();

for (String key : map.keySet()){

StringBuilder sb = new StringBuilder();

sb.append(map.get(key) + " ");

sb.append(key);

res.add(sb.toString());

}

return res;

}

}

# 812\_Largest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have a list of points in the plane. Return the area of the largest triangle that can be formed by any 3 of the points.

Example:

Input: points = [[0,0],[0,1],[1,0],[0,2],[2,0]]

Output: 2

Explanation:

The five points are show in the figure below. The red triangle is the largest.

https://leetcode.com/problems/largest-triangle-area/discuss/122711/C++JavaPython-Solution-with-Explanation-and-Prove

class Solution {

public double largestTriangleArea(int[][] points) {

double max = 0;

for (int i = 0; i < points.length; i++){

for (int j = i+1; j < points.length; j++){

for (int k = j+1; k < points.length; k++){

max = Math.max(max, area(points[i], points[j], points[k]));

}

}

}

return max;

}

private double area(int[] p, int[] q, int[] r){

return 0.5 \* Math.abs(p[0]\*q[1] + q[0]\*r[1] + r[0]\*p[1] - p[0]\*r[1] - r[0]\*q[1] - q[0]\*p[1]);

}

}

class Solution {

public double largestTriangleArea(int[][] points) {

double res = 0;

for (int[] i: points)

for (int[] j: points)

for (int[] k: points)

res = Math.max(res, 0.5 \* Math.abs(i[0] \* j[1] + j[0] \* k[1] + k[0] \* i[1]- j[0] \* i[1] - k[0] \* j[1] - i[0] \* k[1]));

return res;

}

}

# 813\_Largest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We partition a row of numbers A into at most K adjacent (non-empty) groups, then our score is the sum of the average of each group. What is the largest score we can achieve?

Note that our partition must use every number in A, and that scores are not necessarily integers.

Example:

Input:

A = [9,1,2,3,9]

K = 3

Output: 20

Explanation:

The best choice is to partition A into [9], [1, 2, 3], [9]. The answer is 9 + (1 + 2 + 3) / 3 + 9 = 20.

We could have also partitioned A into [9, 1], [2], [3, 9], for example.

That partition would lead to a score of 5 + 2 + 6 = 13, which is worse.

Note:

1 <= A.length <= 100.

1 <= A[i] <= 10000.

1 <= K <= A.length.

Answers within 10^-6 of the correct answer will be accepted as correct.

Intuition

The best score partitioning A[i:] into at most K parts depends on answers to paritioning A[j:] (j > i) into less parts.

We can use dynamic programming as the states form a directed acyclic graph.

Algorithm

Let dp(i, k) be the best score partioning A[i:] into at most K parts.

If the first group we partition A[i:] into ends before j, then our candidate partition has score average(i, j) + dp(j, k-1)),

where average(i, j) = (A[i] + A[i+1] + ... + A[j-1]) / (j - i) (floating point division). We take the highest score of these,

keeping in mind we don't necessarily need to partition - dp(i, k) can also be just average(i, N).

In total, our recursion in the general case is dp(i, k) = max(average(i, N), max\_{j > i}(average(i, j) + dp(j, k-1))).

We can calculate average a little bit faster by remembering prefix sums. If P[x+1] = A[0] + A[1] + ... + A[x],

then average(i, j) = (P[j] - P[i]) / (j - i).

Our implementation showcases a "bottom-up" style of dp. Here at loop number k in our outer-most loop,

dp[i] represents dp(i, k) from the discussion above, and we are calculating the next layer dp(i, k+1).

The end of our second loop for i = 0..N-1 represents finishing the calculation of the correct value for dp(i, t),

and the inner-most loop performs the calculation max\_{j > i}(average(i, j) + dp(j, k)).

class Solution {

public double largestSumOfAverages(int[] A, int K) {

int N = A.length;

//build prefixSum array

double[] prefix = new double[N+1];

for (int i = 0; i < N; i++){

prefix[i+1] = prefix[i] + A[i];

}

//initialize dp

double[] dp = new double[N];

for (int i = 0; i < N; i++){

dp[i] = (prefix[N] - prefix[i]) / (N - i);

}

//dp iteration

for (int k = 0; k < K - 1; k++){//note that here we only do K-1 iteration since initlaization does once

for (int i = 0; i < N; i++){

for (int j = i + 1; j < N; j++){

dp[i] = Math.max(dp[i], (prefix[j] - prefix[i]) / (j - i) + dp[j]);

}

}

}

return dp[0];

}

}

Note that this is 2D DP with built-in 1D LCS DP

so time complexity is O(K \* N \* N)

https://leetcode.com/problems/largest-sum-of-averages/discuss/122775/Java-bottom-up-DP-with-Explanation

https://leetcode.com/problems/largest-sum-of-averages/discuss/126003/C++-DP-Simple-Solution-use-O(N\*K)-Space-and-O(K\*N2)-Time-With-Detailed-Explanation

class Solution {

public double largestSumOfAverages(int[] A, int K) {

int N = A.length;

double[][] dp = new double[K+1][N+1];

double[] preSum = new double[N+1];

for (int i = 1; i <= N; i++){

preSum[i] = preSum[i-1] + A[i-1];

}

if(K <= 1){

return preSum[N] / N;

}

for (int i = 1; i <= N; i++){

dp[1][i] = preSum[i] / i;

}

double max = 0.0;

for (int k = 2; k <= K; k++){

for (int i = k; i <= N; i++){

for (int j = k-1; j < i; j++){

double avg = (preSum[i] - preSum[j]) / (i-j);

dp[k][i] = Math.max(dp[k][i], dp[k-1][j] + avg);

}

}

max = Math.max(max, dp[k][N]);

}

return max;

}

}

class Solution {

public double largestSumOfAverages(int[] A, int K) {

int N = A.length;

double[][] dp = new double[K+1][N+1];

double[] preSum = new double[N+1];

for (int i = 1; i <= N; i++){

preSum[i] = preSum[i-1] + A[i-1];

}

if(K <= 1){

return preSum[N] / N;

}

for (int i = 1; i <= N; i++){

dp[1][i] = preSum[i] / i;

}

double max = 0.0;

for (int k = 2; k <= K; k++){

for (int i = 0; i <= N; i++){

for (int j = 0; j < i; j++){

double avg = (preSum[i] - preSum[j]) / (i-j);

dp[k][i] = Math.max(dp[k][i], dp[k-1][j] + avg);

}

}

max = Math.max(max, dp[k][N]);

}

return max;

}

}

# 814\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given the head node root of a binary tree, where additionally every node's value is either a 0 or a 1.

Return the same tree where every subtree (of the given tree) not containing a 1 has been removed.

(Recall that the subtree of a node X is X, plus every node that is a descendant of X.)

Example 1:

Input: [1,null,0,0,1]

Output: [1,null,0,null,1]

Explanation:

Only the red nodes satisfy the property "every subtree not containing a 1".

The diagram on the right represents the answer.

Example 2:

Input: [1,0,1,0,0,0,1]

Output: [1,null,1,null,1]

Example 3:

Input: [1,1,0,1,1,0,1,0]

Output: [1,1,0,1,1,null,1]

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

Method 1: must use post order traversal because it checks both children before checking root

class Solution {

public TreeNode pruneTree(TreeNode root) {

TreeNode parent = new TreeNode(0);

parent.left = root;

postOrder(root, parent);

return parent.left;

}

private void postOrder(TreeNode root, TreeNode parent){

if (root == null){

return;

}

postOrder(root.left, root);

postOrder(root.right, root);

if (root.left == null && root.right == null && root.val == 0){

if (root == parent.right){

parent.right = null;

}else{

parent.left = null;

}

}

}

}

Method 2: divide conquer (Best solution)

class Solution {

public TreeNode pruneTree(TreeNode root) {

if (root == null){

return null;

}

root.left = pruneTree(root.left);

root.right = pruneTree(root.right);

if (root.left == null && root.right == null && root.val == 0){

return null;

}

return root;

}

}

class Solution {

public TreeNode pruneTree(TreeNode root) {

if (root == null){

return root;

}

root.left = pruneTree(root.left);

root.right = pruneTree(root.right);

if (root.left == null && root.right == null && root.val == 0){

root = null;

}

return root;

}

}

# 815\_Bus.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have a list of bus routes. Each routes[i] is a bus route that the i-th bus repeats forever. For example if routes[0] = [1, 5, 7],

this means that the first bus (0-th indexed) travels in the sequence 1->5->7->1->5->7->1->... forever.

We start at bus stop S (initially not on a bus), and we want to go to bus stop T. Travelling by buses only, what is the least number

of buses we must take to reach our destination? Return -1 if it is not possible.

Example:

Input:

routes = [[1, 2, 7], [3, 6, 7]]

S = 1

T = 6

Output: 2

Explanation:

The best strategy is take the first bus to the bus stop 7, then take the second bus to the bus stop 6.

Note:

1 <= routes.length <= 500.

1 <= routes[i].length <= 500.

0 <= routes[i][j] < 10 ^ 6.

class Solution {

public int numBusesToDestination(int[][] routes, int S, int T) {

Queue<Integer> queue = new LinkedList<>();//bus stops

Set<Integer> visited = new HashSet<>();//bus stops

Map<Integer, Set<Integer>> graph = new HashMap<>();//key: bus stop, value: buses that pass the bus stop

for (int i = 0; i < routes.length; i++){

for (int j = 0; j < routes[i].length; j++){

if (!graph.containsKey(routes[i][j])){

graph.put(routes[i][j], new HashSet<Integer>());

}

Set<Integer> set = graph.get(routes[i][j]);

set.add(i);

}

}

queue.offer(S);//record stop not bus

visited.add(S);//record stop, not bus

int count = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

int stop = queue.poll();

if (stop == T){

return count;

}

Set<Integer> buses = graph.get(stop);

for (int bus : buses){

for (int j = 0; j < routes[bus].length; j++){

if (!visited.contains(routes[bus][j])){

visited.add(routes[bus][j]);

queue.offer(routes[bus][j]);

}

}

}

}

count++;

}

return -1;

}

}

Method 2: Faster

Use bus as the visited set instead of stopo as the visited set

class Solution {

public int numBusesToDestination(int[][] routes, int S, int T) {

if (S == T){

return 0;

}

Map<Integer, Set<Integer>> graph = new HashMap<>();//key : stop, values are the set of buses

for (int i = 0; i < routes.length; i++){

for (int j = 0; j < routes[i].length; j++){

if (!graph.containsKey(routes[i][j])){

graph.put(routes[i][j], new HashSet<>());

}

graph.get(routes[i][j]).add(i);

}

}

Queue<Integer> queue = new LinkedList<>(); //record stop not bus

Set<Integer> set = new HashSet<>(); //record bus not stop, so we don't do set.add(S)

queue.offer(S);

int res = 0;

while (!queue.isEmpty()){

int size = queue.size();

res++;

for (int i = 0; i < size; i++){

int stop = queue.poll();

Set<Integer> buses = graph.get(stop);

for (int bus : buses){

if (set.contains(bus)){

continue;

}

set.add(bus);

for (int j = 0; j < routes[bus].length; j++){

if (routes[bus][j] == T){

return res;

}

queue.offer(routes[bus][j]);

}

}

}

}

return -1;

}

}

# 816\_Ambiguous.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We had some 2-dimensional coordinates, like "(1, 3)" or "(2, 0.5)". Then, we removed all commas, decimal points, and spaces, and ended up with the string S. Return a list of strings representing all possibilities for what our original coordinates could have been.

Our original representation never had extraneous zeroes, so we never started with numbers like "00", "0.0", "0.00", "1.0", "001", "00.01", or any other number that can be represented with less digits. Also, a decimal point within a number never occurs without at least one digit occuring before it, so we never started with numbers like ".1".

The final answer list can be returned in any order. Also note that all coordinates in the final answer have exactly one space between them (occurring after the comma.)

Example 1:

Input: "(123)"

Output: ["(1, 23)", "(12, 3)", "(1.2, 3)", "(1, 2.3)"]

Example 2:

Input: "(00011)"

Output: ["(0.001, 1)", "(0, 0.011)"]

Explanation:

0.0, 00, 0001 or 00.01 are not allowed.

Example 3:

Input: "(0123)"

Output: ["(0, 123)", "(0, 12.3)", "(0, 1.23)", "(0.1, 23)", "(0.1, 2.3)", "(0.12, 3)"]

Example 4:

Input: "(100)"

Output: [(10, 0)]

Explanation:

1.0 is not allowed.

Note:

4 <= S.length <= 12.

S[0] = "(", S[S.length - 1] = ")", and the other elements in S are digits.

class Solution {

public List<String> ambiguousCoordinates(String S) {

List<String> res = new ArrayList<>();

String str = S.substring(1, S.length() - 1);

int len = str.length();

for (int i = 1 ; i < len ; i++){

List<String> left = getList(str.substring(0, i));

List<String> right = getList(str.substring(i));

for (String l : left){//if left is empty, it will not go into loop

for (String r : right){

res.add("(" + l + ", " + r + ")");

}

}

}

return res;

}

private List<String> getList(String S){

int len = S.length();

List<String> res = new ArrayList<>();

if (len == 1){//case 1: length == 1

res.add(S);

return res;

}

if (S.charAt(0) == '0' && S.charAt(len-1) == '0'){// case 2: both start and end with zero, 0012300

return res;

}

if (S.charAt(0) == '0'){// case 3: start with zero 0001234

res.add("0." + S.substring(1));

return res;

}

res.add(S);

if (S.charAt(len-1) == '0'){//case 4: end with zero, 1230

return res;

}

for (int i = 1; i < len; i++){//case 5: neither start nor end with zero

res.add(S.substring(0, i) + "." + S.substring(i));

}

return res;

}

}

# 819\_Most.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a paragraph and a list of banned words, return the most frequent word that is not in the list of banned words. It is guaranteed there is at least one word that isn't banned, and that the answer is unique.

Words in the list of banned words are given in lowercase, and free of punctuation. Words in the paragraph are not case sensitive. The answer is in lowercase.

Example:

Input:

paragraph = "Bob hit a ball, the hit BALL flew far after it was hit."

banned = ["hit"]

Output: "ball"

Explanation:

"hit" occurs 3 times, but it is a banned word.

"ball" occurs twice (and no other word does), so it is the most frequent non-banned word in the paragraph.

Note that words in the paragraph are not case sensitive,

that punctuation is ignored (even if adjacent to words, such as "ball,"),

and that "hit" isn't the answer even though it occurs more because it is banned.

Note:

1 <= paragraph.length <= 1000.

1 <= banned.length <= 100.

1 <= banned[i].length <= 10.

The answer is unique, and written in lowercase (even if its occurrences in paragraph may have uppercase symbols, and even if it is a proper noun.)

paragraph only consists of letters, spaces, or the punctuation symbols !?',;.

Different words in paragraph are always separated by a space.

There are no hyphens or hyphenated words.

Words only consist of letters, never apostrophes or other punctuation symbols.

Method 1:

class Solution {

public String mostCommonWord(String paragraph, String[] banned) {

Map<String, Integer> map = new HashMap<>();

Set<String> set = new HashSet<>(Arrays.asList(banned));

String[] words = paragraph.replaceAll("[!?',;.]", "").toLowerCase().split("\\s+");

int count = 0;

String ans = "";

for (String word : words){

if (!set.contains(word)){

map.put(word, map.getOrDefault(word, 0) + 1);

if (count < map.get(word)){

count = map.get(word);

ans = word;

}

}

}

return ans;

}

}

class Solution {

public String mostCommonWord(String paragraph, String[] banned) {

Map<String, Integer> map = new HashMap<>();

Set<String> set = new HashSet<>(Arrays.asList(banned));

String[] words = paragraph.replaceAll("[!?',;.]", "").toLowerCase().split("\\s+");

int count = 0;

String ans = "";

for (String word : words){

if (!set.contains(word)){

map.put(word, map.getOrDefault(word, 0) + 1);

}

}

for (String key : map.keySet()){

if (count < map.get(key)){

count = map.get(key);

ans = key;

}

}

return ans;

}

}

Best solution:

class Solution {

public String mostCommonWord(String paragraph, String[] banned) {

Set<String> ban = new HashSet<>();

for (String s : banned){

ban.add(s);

}

Map<String, Integer> map = new HashMap<>();

StringBuilder sb = new StringBuilder();

int max = 0;

String res = "";

paragraph += '.';

for (char c : paragraph.toCharArray()){

if (Character.isLetter(c)){

sb.append(Character.toLowerCase(c));

}else{

if (sb.length() > 0){

String str = sb.toString();

sb = new StringBuilder();

if (!ban.contains(str)){

map.put(str, map.getOrDefault(str, 0) + 1);

int count = map.get(str);

if (count > max){

max = count;

res = str;

}

}

}

}

}

return res;

}

}

class Solution {

public String mostCommonWord(String paragraph, String[] banned) {

Set<String> ban = new HashSet<>();

for (String s : banned){

ban.add(s);

}

Map<String, Integer> map = new HashMap<>();

StringBuilder sb = new StringBuilder();

int max = 0;

String res = "";

for (int i = 0; i <= paragraph.length(); i++){

char c = i < paragraph.length() ? paragraph.charAt(i) : '.';

if (Character.isLetter(c)){

sb.append(Character.toLowerCase(c));

}else{

if (sb.length() > 0){

String str = sb.toString();

sb = new StringBuilder();

if (!ban.contains(str)){

map.put(str, map.getOrDefault(str, 0) + 1);

int count = map.get(str);

if (count > max){

max = count;

res = str;

}

}

}

}

}

return res;

}

}

# 82\_Remove.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct

numbers from the original list.

For example,

Given 1->2->3->3->4->4->5, return 1->2->5.

Given 1->1->1->2->3, return 2->3.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

slow: denotes the pointer just before the dup

fast: denotes the pointer at the last dup

class Solution {

public ListNode deleteDuplicates(ListNode head) {

if (head == null){

return head;

}

ListNode dummy = new ListNode(-1);

dummy.next = head;

ListNode slow = dummy;

ListNode fast = head;

while (fast != null){

while (fast.next != null && fast.val == fast.next.val){

fast = fast.next;

}

if (slow.next != fast ){ //note that we are not only comparing the values

slow.next = fast.next;

fast = slow.next;

}else{

slow = slow.next;

fast = fast.next;

}

}

return dummy.next;

}

}

# 820\_Short.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a list of words, we may encode it by writing a reference string S and a list of indexes A.

For example, if the list of words is ["time", "me", "bell"], we can write it as S = "time#bell#" and indexes = [0, 2, 5].

Then for each index, we will recover the word by reading from the reference string from that index until we reach a "#" character.

What is the length of the shortest reference string S possible that encodes the given words?

Example:

Input: words = ["time", "me", "bell"]

Output: 10

Explanation: S = "time#bell#" and indexes = [0, 2, 5].

Note:

1 <= words.length <= 2000.

1 <= words[i].length <= 7.

Each word has only lowercase letters.

Method 1: Trie Best solution

class Solution {

class TrieNode {

TrieNode[] children;

int childrenNum;

public TrieNode(){

children = new TrieNode[26];

childrenNum = 0; //use childrenNum to directly find number of children

}

}

public int minimumLengthEncoding(String[] words) {

Map<TrieNode, Integer> map = new HashMap<>(); // use map to directly find the leaves

//build Trie

TrieNode root = new TrieNode();

for (int j = 0; j < words.length; j++){

TrieNode cur = root;

for (int i = words[j].length() - 1; i >= 0; i--){

char c = words[j].charAt(i);

if (cur.children[c - 'a'] == null){

cur.children[c - 'a'] = new TrieNode();

cur.childrenNum++;

}

cur = cur.children[c - 'a'];

}

map.put(cur, j);

}

int res = 0;

for (TrieNode node : map.keySet()){

if (node.childrenNum == 0){

res += words[map.get(node)].length() + 1; //plus 1 for additional #

}

}

return res;

}

}

Method 2:

class Solution {

public int minimumLengthEncoding(String[] words) {

Set<String> good = new HashSet<>(Arrays.asList(words));

for (String word : words){

for (int i = 1; i < word.length(); i++){

good.remove(word.substring(i));

}

}

int res = 0;

for (String word : good){

res += word.length() + 1;

}

return res;

}

}

# 821\_Shortest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S and a character C, return an array of integers representing the shortest distance from the character C in the string.

Example 1:

Input: S = "loveleetcode", C = 'e'

Output: [3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0]

Note:

S string length is in [1, 10000].

C is a single character, and guaranteed to be in string S.

All letters in S and C are lowercase.

Method 1:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int[] shortestToChar(String S, char C) {

int[] res = new int[S.length()];

LinkedList<Integer> queue = new LinkedList<>();

for (int i = 0; i < S.length(); i++){

if (S.charAt(i) == C){

queue.offer(i);

}

}

int curr = -S.length();

int next = queue.poll();

for (int i = 0; i < S.length(); i++){

if (i > next){

curr = next;

if (!queue.isEmpty()){

next = queue.poll();

}else{

next = 2 \* S.length();

}

}

res[i] = Math.min(Math.abs(i-curr), Math.abs(next-i));

}

return res;

}

}

Method 2: Best solution

Similar question: https://github.com/optimisea/Leetcode/blob/master/Java/849\_Maximize.java

Time complexity: O(n)

Space complexity: O(1)

Loop twice on the string S.

First forward pass to find shortest distant to character on left.

Second backward pass to find shortest distant to character on right.

class Solution {

public int[] shortestToChar(String S, char C) {

int n = S.length();

int[] res = new int[n];

int pos = -n;

for (int i = 0; i < n; i++){

if (S.charAt(i) == C){

pos = i;

}

res[i] = i - pos;

}

pos = 2 \* n;

for (int i = n - 1; i >= 0; i--){

if (S.charAt(i) == C){

pos = i;

}

res[i] = Math.min(res[i], pos - i);

}

return res;

}

}

Method 3: more general solution as Leetcode

Maximize Distance to Closest Person

Longest Mountain in Array

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int[] shortestToChar(String S, char C) {

int n = S.length();

int[] left = new int[n];

int[] right = new int[n];

Arrays.fill(left, n);

Arrays.fill(right, n);

for (int i = 0; i < n; i++){

if (S.charAt(i) == C){

left[i] = 0;

}else if (i > 0){

left[i] = left[i-1] + 1;

}

}

for (int i = n-1; i >= 0; i--){

if (S.charAt(i) == C){

right[i] = 0;

}else if (i < n-1){

right[i] = right[i+1] + 1;

}

}

int[] res = new int[n];

for (int i = 0; i < n; i++){

res[i] = Math.min(left[i], right[i]);

}

return res;

}

}

## 822\_Card.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a table are N cards, with a positive integer printed on the front and back of each card (possibly different).

We flip any number of cards, and after we choose one card.

If the number X on the back of the chosen card is not on the front of any card, then this number X is good.

What is the smallest number that is good? If no number is good, output 0.

Here, fronts[i] and backs[i] represent the number on the front and back of card i.

A flip swaps the front and back numbers, so the value on the front is now on the back and vice versa.

Example:

Input: fronts = [1,2,4,4,7], backs = [1,3,4,1,3]

Output: 2

Explanation: If we flip the second card, the fronts are [1,3,4,4,7] and the backs are [1,2,4,1,3].

We choose the second card, which has number 2 on the back, and it isn't on the front of any card, so 2 is good.

Note:

1 <= fronts.length == backs.length <= 1000.

1 <= fronts[i] <= 2000.

1 <= backs[i] <= 2000.

class Solution {

public int flipgame(int[] fronts, int[] backs) {

Set<Integer> same = new HashSet<>();

for (int i = 0; i < fronts.length; i++){

if (fronts[i] == backs[i]){

same.add(fronts[i]);

}

}

int res = Integer.MAX\_VALUE;

for (int i : fronts){

if (!same.contains(i)){

res = Math.min(res, i);

}

}

for (int i : backs){

if (!same.contains(i)){

res = Math.min(res, i);

}

}

return res % Integer.MAX\_VALUE;

}

}

## 823\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of unique integers, each integer is strictly greater than 1.

We make a binary tree using these integers and each number may be used for any number of times.

Each non-leaf node's value should be equal to the product of the values of it's children.

How many binary trees can we make? Return the answer modulo 10 \*\* 9 + 7.

Example 1:

Input: A = [2, 4]

Output: 3

Explanation: We can make these trees: [2], [4], [4, 2, 2]

Example 2:

Input: A = [2, 4, 5, 10]

Output: 7

Explanation: We can make these trees: [2], [4], [5], [10], [4, 2, 2], [10, 2, 5], [10, 5, 2].

Note:

1 <= A.length <= 1000.

2 <= A[i] <= 10 ^ 9.

DP:

Sort the list A at first.

DP equation:

dp[i] = sum(dp[j] \* dp[i / j])

res = sum(dp[i])

with i, j, i / j in the list L

Time complexity: O(N^2)

Space complexity: O(N)

class Solution {

public int numFactoredBinaryTrees(int[] A) {

long mod = (long) Math.pow(10, 9) + 7;

Arrays.sort(A);

Map<Integer, Integer> index = new HashMap<>();

for (int i = 0; i < A.length; i++){

index.put(A[i], i);

}

long[] dp = new long[A.length];

for (int i = 0; i < A.length; i++){

dp[i] = 1;

for (int j = 0; j < i; j++){

if (A[i] % A[j] == 0 && index.containsKey(A[i] / A[j])){

dp[i] += dp[j] \* dp[index.get(A[i] / A[j])];

}

}

}

long sum = 0;

for (int i = 0; i < A.length; i++){

sum += dp[i];

}

return (int) (sum % mod);

}

}

class Solution {

public int numFactoredBinaryTrees(int[] A) {

long mod = (long) Math.pow(10, 9) + 7;

Arrays.sort(A);

Map<Integer, Integer> index = new HashMap<>();

for (int i = 0; i < A.length; i++){

index.put(A[i], i);

}

long[] dp = new long[A.length];

for (int i = 0; i < A.length; i++){

dp[i] = 1;

for (int j = 0; j < i; j++){

if (A[i] % A[j] == 0 && index.containsKey(A[i] / A[j])){

dp[i] = (dp[i] + dp[j] \* dp[index.get(A[i] / A[j])]) % mod;

}

}

}

long sum = 0;

for (int i = 0; i < A.length; i++){

sum += dp[i];

}

return (int) (sum % mod);

}

}

Best solution:

class Solution {

public int numFactoredBinaryTrees(int[] A) {

int mod = (int) Math.pow(10, 9) + 7;

int n = A.length;

Arrays.sort(A);

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < n; i++){

map.put(A[i], i);

}

long[] dp = new long[n];

long sum = 0;

for (int i = 0; i < n; i++){

dp[i] = 1;

for (int j = 0; j < i; j++){

if (A[i] % A[j] == 0 && map.containsKey( A[i] / A[j] )){

int ind = map.get(A[i]/A[j]);

dp[i] = (dp[i] + (dp[j] \* dp[ind]) % mod)%mod;

}

}

sum = (sum + dp[i]) % mod;

}

return (int)sum;

}

}

Use one hashmap

class Solution {

public int numFactoredBinaryTrees(int[] A) {

long mod = (long) Math.pow(10, 9) + 7;

Arrays.sort(A);

Map<Integer, Long> dp = new HashMap<>();

for (int i = 0; i < A.length; i++){

dp.put(A[i], 1L);

for (int j = 0; j < i; j++){

if (A[i] % A[j] == 0 && dp.containsKey(A[i] / A[j])){

dp.put(A[i], (dp.get(A[i]) + dp.get(A[j]) \* dp.get(A[i] / A[j])) % mod);

}

}

}

long sum = 0;

for (long val : dp.values()){

sum += val;

}

return (int) (sum % mod);

}

}

The largest value v used must be the root node in the tree. Say dp(v) is the number of ways to make a tree with root node v.

If the root node of the tree (with value v) has children with values x and y (and x \* y == v), then there are dp(x) \* dp(y) ways to make this tree.

In total, there are ∑x∗y=vdp(x)∗dp(y) ways to make a tree with root node v.

Algorithm

Actually, let dp[i] be the number of ways to have a root node with value A[i].

Since in the above example we always have x < v and y < v, we can calculate the values of dp[i] in increasing order using dynamic programming.

For some root value A[i], let's try to find candidates for the children with values A[j] and A[i] / A[j] (so that evidently A[j] \* (A[i] / A[j]) = A[i]). To do this quickly, we will need index which looks up this value: if A[k] = A[i] / A[j], then index[A[i] / A[j]] = k`.

After, we'll add all possible dp[j] \* dp[k] (with j < i, k < i) to our answer dp[i]. In our Java implementation, we carefully used long so avoid overflow issues.

## 824\_Goat.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A sentence S is given, composed of words separated by spaces. Each word consists of lowercase and uppercase letters only.

We would like to convert the sentence to "Goat Latin" (a made-up language similar to Pig Latin.)

The rules of Goat Latin are as follows:

If a word begins with a vowel (a, e, i, o, or u), append "ma" to the end of the word.

For example, the word 'apple' becomes 'applema'.

If a word begins with a consonant (i.e. not a vowel), remove the first letter and append it to the end, then add "ma".

For example, the word "goat" becomes "oatgma".

Add one letter 'a' to the end of each word per its word index in the sentence, starting with 1.

For example, the first word gets "a" added to the end, the second word gets "aa" added to the end and so on.

Return the final sentence representing the conversion from S to Goat Latin.

Example 1:

Input: "I speak Goat Latin"

Output: "Imaa peaksmaaa oatGmaaaa atinLmaaaaa"

Example 2:

Input: "The quick brown fox jumped over the lazy dog"

Output: "heTmaa uickqmaaa rownbmaaaa oxfmaaaaa umpedjmaaaaaa overmaaaaaaa hetmaaaaaaaa azylmaaaaaaaaa ogdmaaaaaaaaaa"

Notes:

S contains only uppercase, lowercase and spaces. Exactly one space between each word.

1 <= S.length <= 150.

class Solution {

public String toGoatLatin(String S) {

StringBuilder sb = new StringBuilder();

String[] arr = S.split("\\s+");

String vowels = "aeiouAEIOU";

for (int i = 0; i < arr.length; i++){

String str = arr[i];

if (vowels.contains(str.substring(0, 1))){

sb.append(str);

}else{

if (str.length() > 1){

sb.append(str.substring(1));

sb.append(str.substring(0, 1));

}else{

sb.append(str);

}

}

sb.append("ma");

for (int j = 0; j <= i; j++){

sb.append("a");

}

sb.append(" ");

}

sb.deleteCharAt(sb.length() - 1);

return sb.toString();

}

}

## 825\_Friends.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Some people will make friend requests. The list of their ages is given and ages[i] is the age of the ith person.

Person A will NOT friend request person B (B != A) if any of the following conditions are true:

age[B] <= 0.5 \* age[A] + 7

age[B] > age[A]

age[B] > 100 && age[A] < 100

Otherwise, A will friend request B.

Note that if A requests B, B does not necessarily request A. Also, people will not friend request themselves.

How many total friend requests are made?

Example 1:

Input: [16,16]

Output: 2

Explanation: 2 people friend request each other.

Example 2:

Input: [16,17,18]

Output: 2

Explanation: Friend requests are made 17 -> 16, 18 -> 17.

Example 3:

Input: [20,30,100,110,120]

Output:

Explanation: Friend requests are made 110 -> 100, 120 -> 110, 120 -> 100.

Notes:

1 <= ages.length <= 20000.

1 <= ages[i] <= 120.

Method 1: Brute Force: TLE

Time complexity:O(N^2)

Space complexity: O(1)

class Solution {

public int numFriendRequests(int[] ages) {

int count = 0;

for (int i = 0; i < ages.length; i++){

for (int j = 0; j < ages.length; j++){

if (i != j && ages[j] > 0.5 \*ages[i] + 7 && ages[j] <= ages[i] && (ages[j] <= 100 || ages[i] >= 100)){

count++;

}

}

}

return count;

}

}

Method 2: counting

Time complexity: O(A^2 + N), where N is the number of people, and A is the number of ages.

Space complexity: O(A)

class Solution {

public int numFriendRequests(int[] ages) {

int ans = 0;

int[] counts = new int[121];

for (int age : ages){

counts[age]++;

}

for (int i = 1; i < counts.length; i++){

for (int j = 1; j < counts.length; j++){

if (j <= 0.5 \* i + 7 || j > i || (j > 100 && i < 100)){

continue;

}

if (i != j){

ans += counts[i] \* counts[j]/ 2 \* 2;

}else{

ans += counts[i] \* (counts[i] - 1);

}

}

}

return ans;

}

}

Intuition

Instead of processing all 20000 people, we can process pairs of (age, count) representing how many people are that age.

Since there are only 120 possible ages, this is a much faster loop.

Algorithm

For each pair (ageA, countA), (ageB, countB), if the conditions are satisfied with respect to age, then countA \* countB

pairs of people made friend requests.

If ageA == ageB, then we overcounted: we should have countA \* (countA - 1) pairs of people making friend requests instead, as you cannot friend request yourself.

## 826\_Most.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have jobs: difficulty[i] is the difficulty of the ith job, and profit[i] is the profit of the ith job.

Now we have some workers. worker[i] is the ability of the ith worker, which means that this worker can only complete a job with

difficulty at most worker[i].

Every worker can be assigned at most one job, but one job can be completed multiple times.

For example, if 3 people attempt the same job that pays $1, then the total profit will be $3. If a worker cannot complete any job,

his profit is $0.

What is the most profit we can make?

Example 1:

Input: difficulty = [2,4,6,8,10], profit = [10,20,30,40,50], worker = [4,5,6,7]

Output: 100

Explanation: Workers are assigned jobs of difficulty [4,4,6,6] and they get profit of [20,20,30,30] seperately.

Notes:

1 <= difficulty.length = profit.length <= 10000

1 <= worker.length <= 10000

difficulty[i], profit[i], worker[i] are in range [1, 10^5]

Method 1: two points

Time complexity:O(N\*M), N-- number of worker, M -- number of difficulty

Space complexity: O(1)

class Solution {

public int maxProfitAssignment(int[] difficulty, int[] profit, int[] worker) {

int ans = 0;

for (int i = 0; i < worker.length; i++){

int max = 0;

for (int j = 0; j < difficulty.length; j++){

if (difficulty[j] <= worker[i]){

max = Math.max(max, profit[j]);

}

}

ans += max;

}

return ans;

}

}

Method 2: Best solution TreeMap

Time complexity: O(mlogm + nlogm)

Space complexity: O(m)

class Solution {

public int maxProfitAssignment(int[] difficulty, int[] profit, int[] worker) {

int n = worker.length;

int m = profit.length;

int res = 0;

TreeMap<Integer, Integer> treemap = new TreeMap<>();//key: diff, value : the max profit at this key

for (int i = 0; i < m; i++){

treemap.put(difficulty[i], Math.max(profit[i], treemap.getOrDefault(difficulty[i], 0))); // in case the same diff but different profit

}

int max = 0;

// here is the trick about how to get the max value below or equal the key

for (int diff : treemap.keySet()){

max = Math.max(max, treemap.get(diff));

treemap.put(diff, max); // change treemap to key: diff, value: the max profit below or equal this key

}

/////////////

for (int i = 0; i < n; i++){

Integer key = treemap.floorKey(worker[i]);

if (key != null){

res += treemap.get(key);

}

}

return res;

}

}

## 827\_Making.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a 2D grid of 0s and 1s, we change at most one 0 to a 1.

After, what is the size of the largest island? (An island is a 4-directionally connected group of 1s).

Example 1:

Input: [[1, 0], [0, 1]]

Output: 3

Explanation: Change one 0 to 1 and connect two 1s, then we get an island with area = 3.

Example 2:

Input: [[1, 1], [1, 0]]

Output: 4

Explanation: Change the 0 to 1 and make the island bigger, only one island with area = 1.

Example 3:

Input: [[1, 1], [1, 1]]

Output: 4

Explanation: Can't change any 0 to 1, only one island with area = 1.

Notes:

1 <= grid.length = grid[0].length <= 50.

0 <= grid[i][j] <= 1.

Method 1: DFS/BFS TLE

Time complexity: O(N^4)

Space complexity: O(N^2)

For each 0 in the grid, let's temporarily change it to a 1, then count the size of the group from that square.

class Solution {

int[] dr = new int[]{-1, 0, 1, 0};

int[] dc = new int[]{0, -1, 0, 1};

public int largestIsland(int[][] grid) {

int N = grid.length;

int ans = 0;

boolean hasZero = false;

for (int r = 0; r < N; ++r)

for (int c = 0; c < N; ++c)

if (grid[r][c] == 0) {

hasZero = true;

grid[r][c] = 1;

ans = Math.max(ans, check(grid, r, c));

grid[r][c] = 0;

}

return hasZero ? ans : N\*N;

}

public int check(int[][] grid, int r0, int c0) {

int N = grid.length;

Queue<Integer> queue = new LinkedList();

Set<Integer> seen = new HashSet();

queue.offer(r0 \* N + c0);

seen.add(r0 \* N + c0);

while (!queue.isEmpty()) {

int code = queue.poll();

int r = code / N, c = code % N;

for (int k = 0; k < 4; ++k) {

int nr = r + dr[k], nc = c + dc[k];

if (!seen.contains(nr \* N + nc) && 0 <= nr && nr < N &&

0 <= nc && nc < N && grid[nr][nc] == 1) {

queue.offer(nr \* N + nc);

seen.add(nr \* N + nc);

}

}

}

return seen.size();

}

}

Method 2:

Two Steps:

1. Explore every island using DFS/BFS, count its area, give it an island index and save the result to a {index: area} map.

2. Loop every cell == 0, check its connected islands and calculate total islands area.

Time complexity: O(N^2)

Space complexity: O(N^2)

class Solution {

public int largestIsland(int[][] grid) {

int ans = 0;

int m = grid.length;

int n = grid[0].length;

Map<Integer, Integer> map = new HashMap<>();

int index = 2;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1){

int area = bfs(grid, i, j, index);

map.put(index, area);

index++;

ans = Math.max(ans, area);

}

}

}

//traverse every 0 cell and count biggest island it can conntect

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 0){

int count = 1;

Set<Integer> seen = new HashSet<>();

for (int k = 0; k < dx.length; k++){

int x = i + dx[k];

int y = j + dy[k];

if (0 <= x && x < m && 0 <= y && y < n){

int key = grid[x][y];

if (map.containsKey(key) && !seen.contains(key)){

seen.add(key);

count += map.get(key);

}

}

}

ans = Math.max(ans, count);

}

}

}

return ans;

}

private int bfs(int[][] grid, int x, int y, int index){

grid[x][y] = index;

int m = grid.length;

int n = grid[0].length;

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

Queue<Integer> qx = new LinkedList<>();

Queue<Integer> qy = new LinkedList<>();

qx.offer(x);

qy.offer(y);

int count = 1;

while (!qx.isEmpty()){

int cx = qx.poll();

int cy = qy.poll();

for (int i = 0 ; i < dx.length; i++){

int nx = cx + dx[i];

int ny = cy + dy[i];

if (0 <= nx && nx < m && 0 <= ny && ny < n && grid[nx][ny] == 1){

grid[nx][ny] = index;

qx.offer(nx);

qy.offer(ny);

count++;

}

}

}

return count;

}

}

Method 3: Union Find

O(M\*N)

class Solution {

class UF {

int[] parent;

int[] size;

int count;

public UF (int N){

parent = new int[N];

size = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

count = N;

}

public int find(int x){

if (parent[x] == x){

return x;

}

return parent[x] = find(parent[x]);

}

public void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

count--;

}

}

}

public int largestIsland(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

UF uf = new UF(m\*n);

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

for (int i = 0; i< m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1){

int curr = i \* n + j;

for (int[] dir : dirs){

int nx = i + dir[0];

int ny = j + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] == 1){

int nei = nx \* n + ny;

uf.union(curr, nei);

}

}

}

}

}

int max = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 0){

int count = 1;

int curr = i \* n + j;

Set<Integer> seen = new HashSet<>();

for (int[] dir : dirs){

int nx = i + dir[0];

int ny = j + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] == 1){

int nei = nx \* n + ny;

int root = uf.find(nei);

if (!seen.contains(root)){

count += uf.size[root];

seen.add(root);

}

}

}

max = Math.max(max, count);

}

}

}

return max == 0 ? m \* n : max;

}

}

## 828\_Unique.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A character is unique in string S if it occurs exactly once in it.

For example, in string S = "LETTER", the only unique characters are "L" and "R".

Let's define UNIQ(S) as the number of unique characters in string S.

For example, UNIQ("LETTER") = 2.

Given a string S with only uppercases, calculate the sum of UNIQ(substring) over all non-empty substrings of S.

If there are two or more equal substrings at different positions in S, we consider them different.

Since the answer can be very large, return the answer modulo 10 ^ 9 + 7.

Example 1:

Input: "ABC"

Output: 10

Explanation: All possible substrings are: "A","B","C","AB","BC" and "ABC".

Evey substring is composed with only unique letters.

Sum of lengths of all substring is 1 + 1 + 1 + 2 + 2 + 3 = 10

Example 2:

Input: "ABA"

Output: 8

Explanation: The same as example 1, except uni("ABA") = 1.

Note: 0 <= S.length <= 10000.

Method 1: Brute Force, TLE

Time complexity: O(N^3)

Space complexity: O(N)

class Solution {

public int uniqueLetterString(String S) {

int count = 0;

char[] arr = S.toCharArray();

for (int i = 0; i < arr.length; i++){

for (int j = i; j < arr.length; j++){

count += unique(arr, i, j) % (Math.pow(10, 9) + 7);

}

}

return count;

}

private int unique(char[] arr, int start, int end){

Map<Character, Integer> map = new HashMap<>();

for (int i = start; i <= end; i++){

map.put(arr[i], map.getOrDefault(arr[i], 0) + 1);

}

int count = 0;

for (char key : map.keySet()){

if (map.get(key) == 1){

count++;

}

}

return count;

}

}

Method 2:

Instead of counting all unique characters and struggling with all possible substrings,

we can count for every char in S, how many ways to be found as a unique char.

We count and sum, and it will be out answer.

Time complexity: O(n)

Space complexity: O(n)

https://leetcode.com/problems/unique-letter-string/discuss/128952/One-pass-O(N)-Straight-Forward

class Solution {

public int uniqueLetterString(String S) {

Map<Character, List<Integer>> map = new HashMap<>();

for (int i = 0; i < S.length(); i++){

if (!map.containsKey(S.charAt(i))){

map.put(S.charAt(i), new ArrayList<Integer>());

}

List<Integer> list = map.get(S.charAt(i));

list.add(i);

}

long count = 0;

for (char c : map.keySet()){

List<Integer> list = map.get(c);

for (int i = 0; i < list.size() ; i++){

long prev = i > 0 ? list.get(i-1) : -1;

long next = i < list.size() - 1 ? list.get(i+1) : S.length();

count += (list.get(i) - prev) \* (next - list.get(i)) % (Math.pow(10, 9) + 7);

}

}

return (int) (count % (Math.pow(10, 9) + 7));

}

}

class Solution {

public int uniqueLetterString(String S) {

int mod = (int) Math.pow(10, 9) + 7;

Map<Character, List<Integer>> map = new HashMap<>();

for (int i = 0; i < S.length(); i++){

char c = S.charAt(i);

if (!map.containsKey(c)){

map.put(c, new ArrayList<>());

}

map.get(c).add(i);

}

long res = 0;

for (char key : map.keySet()){

List<Integer> list = map.get(key);

for (int i = 0; i < list.size(); i++){

long prevInd = i == 0 ? -1 : list.get(i-1);

long nextInd = i == list.size() - 1 ? S.length() : list.get(i+1);

long currInd = list.get(i);

res = (res + ((currInd - prevInd) \* (nextInd - currInd)) % mod) % mod;

}

}

return (int) res;

}

}

## 829\_Consecutive.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer N, how many ways can we write it as a sum of consecutive positive integers?

Example 1:

Input: 5

Output: 2

Explanation: 5 = 5 = 2 + 3

Example 2:

Input: 9

Output: 3

Explanation: 9 = 9 = 4 + 5 = 2 + 3 + 4

Example 3:

Input: 15

Output: 4

Explanation: 15 = 15 = 8 + 7 = 4 + 5 + 6 = 1 + 2 + 3 + 4 + 5

Note: 1 <= N <= 10 ^ 9.

https://leetcode.com/problems/consecutive-numbers-sum/discuss/129015/5-lines-C++-solution-with-detailed-mathematical-explanation.

Time complexity: O(sqrt(n))

class Solution {

public int consecutiveNumbersSum(int N) {

int count = 1;

for (int k = 2; k \* k < 2\*N; k++){

if ((N - (k\*(k-1)/2))%k == 0){

count++;

}

}

return count;

}

}

https://leetcode.com/problems/consecutive-numbers-sum/discuss/209317/topic

class Solution {

public int consecutiveNumbersSum(int N) {

int count = 0;

int m = 1; //number of items that can consecutively sum up to N

while (true){

int mx = N - m\*(m-1)/2;

if (mx <= 0){

break;

}else if (mx % m == 0){

count++;

}

m++;

}

return count;

}

}

# 830\_Positions.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a string S of lowercase letters, these letters form consecutive groups of the same character.

For example, a string like S = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z" and "yy".

Call a group large if it has 3 or more characters. We would like the starting and ending positions of every large group.

The final answer should be in lexicographic order.

Example 1:

Input: "abbxxxxzzy"

Output: [[3,6]]

Explanation: "xxxx" is the single large group with starting 3 and ending positions 6.

Example 2:

Input: "abc"

Output: []

Explanation: We have "a","b" and "c" but no large group.

Example 3:

Input: "abcdddeeeeaabbbcd"

Output: [[3,5],[6,9],[12,14]]

Note: 1 <= S.length <= 1000

Method 1:

class Solution {

public List<List<Integer>> largeGroupPositions(String S) {

List<List<Integer>> res = new ArrayList<>();

int i = 0;

while (i < S.length()){

i++;

int count = 1;

int start = i;

while (i < S.length() && S.charAt(i-1) == S.charAt(i)){

count++;

i++;

}

if (count >= 3){

List<Integer> item = new ArrayList<>();

item.add(start-1);

item.add(i-1);

res.add(new ArrayList<Integer>(item));

}

}

return res;

}

}

Method 2: two pointers

class Solution {

public List<List<Integer>> largeGroupPositions(String S) {

List<List<Integer>> res = new ArrayList<>();

int i = 0;

int j = 0;

while (i < S.length()){

while (j < S.length() && S.charAt(j) == S.charAt(i)){

j++;

}

if (j - i >= 3){

res.add(Arrays.asList(i, j-1));

}

i = j;

}

return res;

}

}

Similar as Group Binary String

https://github.com/optimisea/Leetcode/blob/master/Java/696\_Count.java

Best solution:

class Solution {

public List<List<Integer>> largeGroupPositions(String S) {

List<List<Integer>> res = new ArrayList<>();

int curr = 1;

int currStart = 0;

for (int i = 1; i < S.length(); i++){

if (S.charAt(i) == S.charAt(i-1)){

curr++;

}else{

if (curr >= 3){

List<Integer> list = new ArrayList<>();

list.add(currStart);

list.add(i-1);

res.add(list);

}

currStart = i;

curr = 1;

}

}

if (curr >= 3){

List<Integer> list = new ArrayList<>();

list.add(currStart);

list.add(S.length()-1);

res.add(list);

}

return res;

}

}

## 831\_Masking.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given a personal information string S, which may represent either an email address or a phone number.

We would like to mask this personal information according to the following rules:

1. Email address:

We define a name to be a string of length ≥ 2 consisting of only lowercase letters a-z or uppercase letters A-Z.

An email address starts with a name, followed by the symbol '@', followed by a name, followed by the dot '.' and followed by a name.

All email addresses are guaranteed to be valid and in the format of "name1@name2.name3".

To mask an email, all names must be converted to lowercase and all letters between the first and last letter of the first name must be replaced by 5 asterisks '\*'.

2. Phone number:

A phone number is a string consisting of only the digits 0-9 or the characters from the set {'+', '-', '(', ')', ' '}. You may assume a phone number contains 10 to 13 digits.

The last 10 digits make up the local number, while the digits before those make up the country code. Note that the country code is optional. We want to expose only the last 4 digits and mask all other digits.

The local number should be formatted and masked as "\*\*\*-\*\*\*-1111", where 1 represents the exposed digits.

To mask a phone number with country code like "+111 111 111 1111", we write it in the form "+\*\*\*-\*\*\*-\*\*\*-1111". The '+' sign and the first '-' sign before the local number should only exist if there is a country code. For example, a 12 digit phone number mask should start with "+\*\*-".

Note that extraneous characters like "(", ")", " ", as well as extra dashes or plus signs not part of the above formatting scheme should be removed.

Return the correct "mask" of the information provided.

Example 1:

Input: "LeetCode@LeetCode.com"

Output: "l\*\*\*\*\*e@leetcode.com"

Explanation: All names are converted to lowercase, and the letters between the

first and last letter of the first name is replaced by 5 asterisks.

Therefore, "leetcode" -> "l\*\*\*\*\*e".

Example 2:

Input: "AB@qq.com"

Output: "a\*\*\*\*\*b@qq.com"

Explanation: There must be 5 asterisks between the first and last letter

of the first name "ab". Therefore, "ab" -> "a\*\*\*\*\*b".

Example 3:

Input: "1(234)567-890"

Output: "\*\*\*-\*\*\*-7890"

Explanation: 10 digits in the phone number, which means all digits make up the local number.

Example 4:

Input: "86-(10)12345678"

Output: "+\*\*-\*\*\*-\*\*\*-5678"

Explanation: 12 digits, 2 digits for country code and 10 digits for local number.

Notes:

S.length <= 40.

Emails have length at least 8.

Phone numbers have length at least 10.

Method 1:

class Solution {

public String maskPII(String S) {

StringBuilder sb = new StringBuilder();

int index = S.indexOf("@");

if (index > -1){//email

sb.append(Character.toLowerCase(S.charAt(0)));

sb.append("\*\*\*\*\*");

sb.append(Character.toLowerCase(S.charAt(index-1)));

sb.append("@");

for (int i = index + 1; i < S.length(); i++){

sb.append(Character.toLowerCase(S.charAt(i)));

}

}else{//phone number

StringBuilder temp = new StringBuilder();

for (int i = 0; i < S.length(); i++){

char c = S.charAt(i);

if (Character.isDigit(c)){

temp.append(c);

}

}

int len = temp.length();

if (len > 10){

sb.append("+");

for (int i = 0; i < len - 10; i++){

sb.append("\*");

}

sb.append("-");

}

sb.append("\*\*\*-\*\*\*-");

sb.append(temp.substring(len-4));

}

return sb.toString();

}

}

Method 2: use RegEx

class Solution {

public String maskPII(String S) {

int index = S.indexOf("@");

if (index > -1){//email

return (S.substring(0, 1) + "\*\*\*\*\*" + S.substring(index-1)).toLowerCase();

}

String numStr = S.replaceAll("[^0-9]", "");

int len = numStr.length();

StringBuilder sb = new StringBuilder();

if (len > 10){

sb.append("+");

for (int i = 0; i < len - 10; i++){

sb.append("\*");

}

sb.append("-");

}

sb.append("\*\*\*-\*\*\*-");

sb.append(numStr.substring(len-4));

return sb.toString();

}

}

## 832\_Flipping.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary matrix A, we want to flip the image horizontally, then invert it, and return the resulting image.

To flip an image horizontally means that each row of the image is reversed. For example, flipping [1, 1, 0] horizontally results in [0, 1, 1].

To invert an image means that each 0 is replaced by 1, and each 1 is replaced by 0. For example, inverting [0, 1, 1] results in [1, 0, 0].

Example 1:

Input: [[1,1,0],[1,0,1],[0,0,0]]

Output: [[1,0,0],[0,1,0],[1,1,1]]

Explanation: First reverse each row: [[0,1,1],[1,0,1],[0,0,0]].

Then, invert the image: [[1,0,0],[0,1,0],[1,1,1]]

Example 2:

Input: [[1,1,0,0],[1,0,0,1],[0,1,1,1],[1,0,1,0]]

Output: [[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]

Explanation: First reverse each row: [[0,0,1,1],[1,0,0,1],[1,1,1,0],[0,1,0,1]].

Then invert the image: [[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]

Notes:

1 <= A.length = A[0].length <= 20

0 <= A[i][j] <= 1

class Solution {

public int[][] flipAndInvertImage(int[][] A) {

int m = A.length;

int n = A[0].length;

for (int i = 0; i < m; i++){

for (int j = 0; j < (n+1)/2; j++){

int temp = A[i][j];

A[i][j] = A[i][n-1-j] == 0 ? 1: 0;

A[i][n-1-j] = temp == 0 ? 1 : 0;

}

}

return A;

}

}

class Solution {

public int[][] flipAndInvertImage(int[][] A) {

int m = A.length;

int n = A[0].length;

for (int i = 0; i < m; i++){

for (int j = 0; j < (n+1)/2; j++){

int temp = A[i][j];

A[i][j] = A[i][n-1-j] ^ 1;

A[i][n-1-j] = temp ^ 1;

}

}

return A;

}

}

## 833\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To some string S, we will perform some replacement operations that replace groups of letters with new ones (not necessarily the same

size). Each replacement operation has 3 parameters: a starting index i, a source word x and a target word y. The rule is that

if x starts at position i in the original string S, then we will replace that occurrence of x with y. If not, we do nothing.

For example, if we have S = "abcd" and we have some replacement operation i = 2, x = "cd", y = "ffff", then because "cd" starts at

position 2 in the original string S, we will replace it with "ffff".

Using another example on S = "abcd", if we have both the replacement operation i = 0, x = "ab", y = "eee", as well as

another replacement operation i = 2, x = "ec", y = "ffff", this second operation does nothing because in the original

string S[2] = 'c', which doesn't match x[0] = 'e'.

All these operations occur simultaneously. It's guaranteed that there won't be any overlap in replacement: for example,

S = "abc", indexes = [0, 1], sources = ["ab","bc"] is not a valid test case.

Example 1:

Input: S = "abcd", indexes = [0,2], sources = ["a","cd"], targets = ["eee","ffff"]

Output: "eeebffff"

Explanation: "a" starts at index 0 in S, so it's replaced by "eee".

"cd" starts at index 2 in S, so it's replaced by "ffff".

Example 2:

Input: S = "abcd", indexes = [0,2], sources = ["ab","ec"], targets = ["eee","ffff"]

Output: "eeecd"

Explanation: "ab" starts at index 0 in S, so it's replaced by "eee".

"ec" doesn't starts at index 2 in the original S, so we do nothing.

Notes:

0 <= indexes.length = sources.length = targets.length <= 100

0 < indexes[i] < S.length <= 1000

All characters in given inputs are lowercase letters.

class Solution {

public String findReplaceString(String S, int[] indexes, String[] sources, String[] targets) {

StringBuilder sb = new StringBuilder();

int n = S.length();

int[] match = new int[n];//record the index in targets array that needs to be replaced

Arrays.fill(match, -1);

for (int i = 0; i < indexes.length; i++){

int len = sources[i].length();

int ind = indexes[i];

String str = S.substring(ind, ind + len);

if (sources[i].equals(str)){

match[ind] = i;

}

}

for (int i = 0; i < n; i++){

if (match[i] >= 0){

sb.append(targets[match[i]]);

i += sources[match[i]].length() - 1;

}else{

sb.append(S.charAt(i));

}

}

return sb.toString();

}

}

class Solution {

public String findReplaceString(String S, int[] indexes, String[] sources, String[] targets) {

int n = S.length();

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < indexes.length; i++){

String source = sources[i];

String sub = S.substring(indexes[i], indexes[i] + source.length());

if (sub.equals(source)){

map.put(indexes[i], i);

}

}

StringBuilder sb = new StringBuilder();

for (int i = 0; i < n; i++){

if (map.containsKey(i)){

sb.append(targets[map.get(i)]);

i += sources[map.get(i)].length() - 1;

}else{

sb.append(S.charAt(i));

}

}

return sb.toString();

}

}

## 834\_Sum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

An undirected, connected tree with N nodes labelled 0...N-1 and N-1 edges are given.

The ith edge connects nodes edges[i][0] and edges[i][1] together.

Return a list ans, where ans[i] is the sum of the distances between node i and all other nodes.

Example 1:

Input: N = 6, edges = [[0,1],[0,2],[2,3],[2,4],[2,5]]

Output: [8,12,6,10,10,10]

Explanation:

Here is a diagram of the given tree:

0

/ \

1 2

/|\

3 4 5

We can see that dist(0,1) + dist(0,2) + dist(0,3) + dist(0,4) + dist(0,5)

equals 1 + 1 + 2 + 2 + 2 = 8. Hence, answer[0] = 8, and so on.

Note: 1 <= N <= 10000

Method 1: Brute Force TLE

Time complexity: O((V+E)\*V^2)

Space complexity: O(V+E)

class Solution {

public int[] sumOfDistancesInTree(int N, int[][] edges) {

if (edges.length == 0){

return new int[N];

}

Map<Integer, Set<Integer>> graph = new HashMap<>();

//build graph

for (int[] edge : edges){

if (!graph.containsKey(edge[0])){

graph.put(edge[0], new HashSet<Integer>());

}

Set<Integer> set0 = graph.get(edge[0]);

set0.add(edge[1]);

if (!graph.containsKey(edge[1])){

graph.put(edge[1], new HashSet<Integer>());

}

Set<Integer> set1 = graph.get(edge[1]);

set1.add(edge[0]);

}

//dfs

int[] ans = new int[N];

for (int i = 0; i < N; i++){

for (int j = 0; j < N; j++){

ans[i] += bfs(graph, i, j);

}

}

return ans;

}

private int bfs(Map<Integer, Set<Integer>> graph, int start, int end){

if (start == end){

return 0;

}

Queue<Integer> queue = new LinkedList<>();

Set<Integer> seen = new HashSet<>();

queue.offer(start);

seen.add(start);

int dist = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

int node = queue.poll();

if (node == end){

return dist;

}

Set<Integer> set = graph.get(node);

for (int j : set){

if (!seen.contains(j)){

queue.offer(j);

seen.add(j);

}

}

}

dist++;

}

return -10000;

}

}

Method 2: Best solution

Time complexity: O(V+E)

Space complexity: O(V+E)

class Solution {

public int[] sumOfDistancesInTree(int N, int[][] edges) {

int[] res = new int[N]; //first stores the sum distance of curr node to its subtree nodes, then stores the answer

int[] count = new int[N]; //store the number of subtree nodes (including i) at node i

Arrays.fill(count, 1);

Map<Integer, Set<Integer>> graph = new HashMap<>();

for (int i = 0; i < N; i++){

graph.put(i, new HashSet<Integer>());

}

for (int[] edge : edges){

graph.get(edge[0]).add(edge[1]);

graph.get(edge[1]).add(edge[0]);

}

postOrder(graph, res, count, 0, new HashSet<Integer>()); //calculate the distance from 0 to all nodes

preOrder(graph, res, count, 0, new HashSet<Integer>()); //calculate others

return res;

}

//calculate the sum of all the distance from root to nodes

private void postOrder(Map<Integer, Set<Integer>> graph, int[] res, int[] count, int parent, Set<Integer> seen){

seen.add(parent);

for (int child : graph.get(parent)){

if (!seen.contains(child)){

seen.add(child);

postOrder(graph, res, count, child, seen);

count[parent] += count[child];

res[parent] += res[child] + count[child];

}

}

}

//calculate the sum of all the distance from node to all other nodes

private void preOrder(Map<Integer, Set<Integer>> graph, int[] res, int[] count, int parent, Set<Integer> seen){

seen.add(parent);

for (int child : graph.get(parent)){

if (!seen.contains(child)){

res[child] = res[parent] - count[child] + res.length - count[child];

preOrder(graph, res, count, child, seen);

}

}

}

}

https://leetcode.com/problems/sum-of-distances-in-tree/solution/

class Solution {

int[] ans, count;

List<Set<Integer>> graph;

int N;

public int[] sumOfDistancesInTree(int N, int[][] edges) {

this.N = N;

graph = new ArrayList<Set<Integer>>();

ans = new int[N];

count = new int[N];

Arrays.fill(count, 1);

for (int i = 0; i < N; ++i)

graph.add(new HashSet<Integer>());

for (int[] edge: edges) {

graph.get(edge[0]).add(edge[1]);

graph.get(edge[1]).add(edge[0]);

}

dfs(0, -1);

dfs2(0, -1);

return ans;

}

public void dfs(int node, int parent) {

for (int child: graph.get(node))

if (child != parent) {

dfs(child, node);

count[node] += count[child];

ans[node] += ans[child] + count[child];

}

}

public void dfs2(int node, int parent) {

for (int child: graph.get(node))

if (child != parent) {

ans[child] = ans[node] - count[child] + N - count[child];

dfs2(child, node);

}

}

}

## 835\_Image.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Two images A and B are given, represented as binary, square matrices of the same size. (A binary matrix has only 0s and 1s as values.)

We translate one image however we choose (sliding it left, right, up, or down any number of units), and place it on top of the other image. After, the overlap of this translation is the number of positions that have a 1 in both images.

(Note also that a translation does not include any kind of rotation.)

What is the largest possible overlap?

Example 1:

Input: A = [[1,1,0],

[0,1,0],

[0,1,0]]

B = [[0,0,0],

[0,1,1],

[0,0,1]]

Output: 3

Explanation: We slide A to right by 1 unit and down by 1 unit.

Notes:

1 <= A.length = A[0].length = B.length = B[0].length <= 30

0 <= A[i][j], B[i][j] <= 1

Method:

Time complexity: O(N^2)

Spapce complexity: O(N)

https://leetcode.com/problems/image-overlap/discuss/130623/C++JavaPython-Straight-Forward

note that in order to avoid collison, the factor must be greater than 2\*n - 1.

avoid case :[0,1], [1, 1]

[1, 1], [1, 0]

class Solution {

public int largestOverlap(int[][] A, int[][] B) {

int n = A.length;

int factor = 2 \* n;

List<Integer> listA = new ArrayList<>();

List<Integer> listB = new ArrayList<>();

for (int i = 0; i < n; i++){

for (int j = 0; j < n; j++){

if (A[i][j] == 1){

listA.add(i \* factor + j);

}

if (B[i][j] == 1){

listB.add(i \* factor + j);

}

}

}

Map<Integer, Integer> map = new HashMap<>();

for (int i : listA){

for (int j : listB){

map.put(i-j, map.getOrDefault(i-j, 0) + 1);

}

}

int max = 0;

for (int value : map.values()){

max = Math.max(max, value);

}

return max;

}

}

## 837\_New.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Alice plays the following game, loosely based on the card game "21".

Alice starts with 0 points, and draws numbers while she has less than K points. During each draw, she gains an integer number of points randomly from the range [1, W], where W is an integer. Each draw is independent and the outcomes have equal probabilities.

Alice stops drawing numbers when she gets K or more points. What is the probability that she has N or less points?

Example 1:

Input: N = 10, K = 1, W = 10

Output: 1.00000

Explanation: Alice gets a single card, then stops.

Example 2:

Input: N = 6, K = 1, W = 10

Output: 0.60000

Explanation: Alice gets a single card, then stops.

In 6 out of W = 10 possibilities, she is at or below N = 6 points.

Example 3:

Input: N = 21, K = 17, W = 10

Output: 0.73278

Note:

0 <= K <= N <= 10000

1 <= W <= 10000

Answers will be accepted as correct if they are within 10^-5 of the correct answer.

The judging time limit has been reduced for this question.

Method: DP + Sliding window

https://leetcode.com/problems/new-21-game/discuss/132334/One-Pass-DP-O(N)

When the game ends, the point is between K and K-1+W

What is the probability that the the point is less than N?

- If N is greater than K-1+W, probability is 1

- If N is less than K, probability is 0

If W == 3 and we want to find the probability to get a 5

- You can get a card with value 1, 2, or 3 with equal probability (1/3)

- If you had a 4 and you get a 1: prob(4) \* (1/3)

- If you had a 3 and you get a 2: prob(3) \* (1/3)

- If you had a 2 and you get a 3: prob(2) \* (1/3)

- If you had a 1, you can never reach a 5 in the next draw

- prob(5) = prob(4) / 3 + prob(3) / 3 + prob(2) /3

To generalize:

The probability to get point K is

p(K) = p(K-1) / W + p(K-2) / W + p(K-3) / W + ... p(K-W) / W

let wsum = p(K-1) + p(K-2) + ... + p(K-W)

p(K) = wsum / W

class Solution {

public double new21Game(int N, int K, int W) {

if (K == 0){

return 1.0;

}

double res = 0.0;

double[] dp = new double[N+1];//the probability to get i point

dp[0] = 1.0;

double Wsum = 1.0;

for (int i = 1; i <= N; i++){

dp[i] = Wsum / W;

if (i < K){

Wsum += dp[i];

}else{//in this case, no card draw

res += dp[i];

}

if (i >= W){

Wsum -= dp[i-W];

}

}

return res;

}

}

## 838\_Push.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are N dominoes in a line, and we place each domino vertically upright.

In the beginning, we simultaneously push some of the dominoes either to the left or to the right.

After each second, each domino that is falling to the left pushes the adjacent domino on the left.

Similarly, the dominoes falling to the right push their adjacent dominoes standing on the right.

When a vertical domino has dominoes falling on it from both sides, it stays still due to the balance of the forces.

For the purposes of this question, we will consider that a falling domino expends no additional force to a falling or already fallen domino.

Given a string "S" representing the initial state. S[i] = 'L', if the i-th domino has been pushed to the left; S[i] = 'R', if the i-th domino has been pushed to the right; S[i] = '.', if the i-th domino has not been pushed.

Return a string representing the final state.

Example 1:

Input: ".L.R...LR..L.."

Output: "LL.RR.LLRRLL.."

Example 2:

Input: "RR.L"

Output: "RR.L"

Explanation: The first domino expends no additional force on the second domino.

Note:

0 <= N <= 10^5

String dominoes contains only 'L', 'R' and '.'

Method 1:

class Solution {

public String pushDominoes(String dominoes) {

int len = dominoes.length();

StringBuilder right = new StringBuilder();

for (int i = 0; i < len; i++){

if (dominoes.charAt(i) == '.'){

if (i > 0 && right.charAt(i-1) == 'R'){

right.append('R');

}else{

right.append('.');

}

}else{

right.append(dominoes.charAt(i));

}

}

StringBuilder left = new StringBuilder();

for (int i = 0; i < len; i++){

if (dominoes.charAt(len - 1 - i) == '.'){

if (i > 0 && left.charAt(i-1) == 'L'){

left.append('L');

}else{

left.append('.');

}

}else{

left.append(dominoes.charAt(len - 1 - i));

}

}

left = left.reverse();

StringBuilder sb = new StringBuilder();

int start = 0;

for (int i = 0; i < len; i++){

if (left.charAt(i) == right.charAt(i) || left.charAt(i) == '.' || right.charAt(i) == '.'){

sb.append(left.charAt(i) != '.' ? left.charAt(i) : right.charAt(i));

}else{

start = i;

int count = 0;

while (i < len && left.charAt(i) != right.charAt(i)){

count++;

i++;

}

if (count % 2 == 0){

for (int j = start; j < i; j++){

if (j - start < count / 2){

sb.append('R');

}else{

sb.append('L');

}

}

}else{

for (int j = start; j < i; j++){

if (j - start < count / 2){

sb.append('R');

}else if (j - start == count / 2){

sb.append('.');

}else{

sb.append('L');

}

}

}

i = start + count - 1;

}

}

return sb.toString();

}

}

Method 2: Best solution:

https://leetcode.com/problems/push-dominoes/discuss/132482/Java-one-pass-in-place-13ms

Two pointers:

key:

1. four cases: L .... L

R .... L

L .... R

R .... R

2. Record last seen L and R to find out the prevous letter

3. Pay attention to corner case at the start (prevL = -1, prevR = -1) and at the end (i == n)

class Solution {

public String pushDominoes(String dominoes) {

char[] arr = dominoes.toCharArray();

int n = dominoes.length();

int prevL = -1; //previous L position

int prevR = -1;

for (int i = 0; i <= n; i++){

if (i < n && arr[i] == 'L'){

if (prevL > prevR || prevL == -1 && prevR == -1){//L....L

while (prevL < i){

prevL++;

arr[prevL] = 'L';

}

}else{//R...L

int low = prevR + 1;

int high = i - 1;

while (low < high){

arr[low] = 'R';

arr[high] = 'L';

low++;

high--;

}

prevL = i;

}

}else if (i == n || arr[i] == 'R'){

if (prevR > prevL){//R...R

while (prevR < i){

arr[prevR] = 'R';

prevR++;

}

}else{//L...R

prevR = i;

}

}

}

return new String(arr);

}

}

## 839\_Similar.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Two strings X and Y are similar if we can swap two letters (in different positions) of X, so that it equals Y.

For example, "tars" and "rats" are similar (swapping at positions 0 and 2), and "rats" and "arts" are similar, but "star" is not similar to "tars", "rats", or "arts".

Together, these form two connected groups by similarity: {"tars", "rats", "arts"} and {"star"}. Notice that "tars" and "arts" are in the same group even though they are not similar. Formally, each group is such that a word is in the group if and only if it is similar to at least one other word in the group.

We are given a list A of strings. Every string in A is an anagram of every other string in A. How many groups are there?

Example 1:

Input: ["tars","rats","arts","star"]

Output: 2

Note:

A.length <= 2000

A[i].length <= 1000

A.length \* A[i].length <= 20000

All words in A consist of lowercase letters only.

All words in A have the same length and are anagrams of each other.

The judging time limit has been increased for this question.

Method 1: Union Find

Time complexity: O(n^2 \* m)

class Solution {

class UF {

int[] parent;

int[] size;

int count;

public UF (int N){

parent = new int[N];

size = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

count = N;

}

public int find (int x){

if (parent[x] == x){

return x;

}

return find(parent[x]);

}

public void union (int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootY] = rootX;

size[x] += size[y];

count--;

}

}

}

public int numSimilarGroups(String[] A) {

int N = A.length;

UF uf = new UF(N);

for (int i = 0; i < N; i++){

for (int j = i; j < N; j++){

if (isSimilar(A[i], A[j])){

uf.union(i, j);

}

}

}

return uf.count;

}

private boolean isSimilar(String a, String b){

if (a.length() != b.length()){

return false;

}

int n = a.length();

int firstInd = n;

int secondInd = n;

for (int i = 0; i < n; i++){

if (a.charAt(i) != b.charAt(i)){

if (firstInd == n){

firstInd = i;

}else if (firstInd != n && secondInd == n){

if (a.charAt(firstInd) != b.charAt(i) || a.charAt(i) != b.charAt(firstInd)){

return false;

}

secondInd = i;

}else{

return false;

}

}

}

return true;

}

}

## 84\_LargestRectangle.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1,

find the area of largest rectangle in the histogram.

Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].

The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example,

Given heights = [2,1,5,6,2,3],

return 10.

Method: monotonic decreasing stack: top element is the largest one in the stack

O(n)

class Solution {

public int largestRectangleArea(int[] heights) {

if (heights == null || heights.length == 0){

return 0;

}

Stack<Integer> stack = new Stack<>();

int max = 0;

for (int i = 0; i <= heights.length; i++){

int curr = (i == heights.length) ? -1 : heights[i];

while (!stack.isEmpty() && curr < heights[stack.peek()]){

int h = heights[stack.pop()];

int w = stack.isEmpty() ? i : i - stack.peek() - 1;

max = Math.max(max, h \* w);

}

stack.push(i);

}

return max;

}

}

## 840\_Magic.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A 3 x 3 magic square is a 3 x 3 grid filled with distinct numbers from 1 to 9 such that each row, column, and both diagonals all have the same sum.

Given an grid of integers, how many 3 x 3 "magic square" subgrids are there? (Each subgrid is contiguous).

Example 1:

Input: [[4,3,8,4],

[9,5,1,9],

[2,7,6,2]]

Output: 1

Explanation:

The following subgrid is a 3 x 3 magic square:

438

951

276

while this one is not:

384

519

762

In total, there is only one magic square inside the given grid.

Note:

1 <= grid.length <= 10

1 <= grid[0].length <= 10

0 <= grid[i][j] <= 15

class Solution {

public int numMagicSquaresInside(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int count = 0;

for (int i = 0; i < m-2; i++){

for (int j = 0; j < n-2; j++){

if (validMagic(grid, i, j)){

count++;

}

}

}

return count;

}

private boolean validMagic(int[][] grid, int x, int y){

//check distinct

Set<Integer> set = new HashSet<>();

for (int i = x; i < x+3; i++){

for (int j = y; j < y+3; j++){

if (grid[i][j] < 1 || grid[i][j] > 9){

return false;

}

if (!set.add(grid[i][j])){

return false;

}

}

}

//check horizal

for (int i = x; i < x+3; i++){

int rowSum = 0;

for (int j = y; j < y+3; j++){

rowSum += grid[i][j];

}

if (rowSum != 15){

return false;

}

}

//check vertical

for (int j = y; j < y+3; j++){

int colSum = 0;

for (int i = x; i < x+3; i++){

colSum += grid[i][j];

}

if (colSum != 15){

return false;

}

}

//check diagonal

int diagSum = grid[x][y] + grid[x+1][y+1] + grid[x+2][y+2];

if (diagSum != 15){

return false;

}

//check anti-diagonal

int antidiagSum = grid[x+2][y] + grid[x+1][y+1] + grid[x][y+2];

if (antidiagSum != 15){

return false;

}

return true;

}

}

## 841\_Keys.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are N rooms and you start in room 0. Each room has a distinct number in 0, 1, 2, ..., N-1, and each room may have some keys to access the next room.

Formally, each room i has a list of keys rooms[i], and each key rooms[i][j] is an integer in [0, 1, ..., N-1] where N = rooms.length. A key rooms[i][j] = v opens the room with number v.

Initially, all the rooms start locked (except for room 0).

You can walk back and forth between rooms freely.

Return true if and only if you can enter every room.

Example 1:

Input: [[1],[2],[3],[]]

Output: true

Explanation:

We start in room 0, and pick up key 1.

We then go to room 1, and pick up key 2.

We then go to room 2, and pick up key 3.

We then go to room 3. Since we were able to go to every room, we return true.

Example 2:

Input: [[1,3],[3,0,1],[2],[0]]

Output: false

Explanation: We can't enter the room with number 2.

Note:

1 <= rooms.length <= 1000

0 <= rooms[i].length <= 1000

The number of keys in all rooms combined is at most 3000.

Method 1: DFS

class Solution {

public boolean canVisitAllRooms(List<List<Integer>> rooms) {

List<Integer> seen = new ArrayList<>();

dfs(seen, rooms, 0);

return seen.size() == rooms.size();

}

private void dfs(List<Integer> seen, List<List<Integer>> rooms, int start){

seen.add(start);

for (int room : rooms.get(start)){

if (!seen.contains(room)){

dfs(seen, rooms, room);

}

}

}

}

Method 2: BFS

class Solution {

public boolean canVisitAllRooms(List<List<Integer>> rooms) {

List<Integer> seen = new ArrayList<>();

Queue<Integer> queue = new LinkedList<>();

queue.offer(0);

seen.add(0);

while (!queue.isEmpty()){

int curr = queue.poll();

for (int room : rooms.get(curr)){

if (!seen.contains(room)){

seen.add(room);

queue.offer(room);

}

}

}

return seen.size() == rooms.size();

}

}

## 842\_Split.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S of digits, such as S = "123456579", we can split it into a Fibonacci-like sequence [123, 456, 579].

Formally, a Fibonacci-like sequence is a list F of non-negative integers such that:

0 <= F[i] <= 2^31 - 1, (that is, each integer fits a 32-bit signed integer type);

F.length >= 3;

and F[i] + F[i+1] = F[i+2] for all 0 <= i < F.length - 2.

Also, note that when splitting the string into pieces, each piece must not have extra leading zeroes, except if the piece is the number 0 itself.

Return any Fibonacci-like sequence split from S, or return [] if it cannot be done.

Example 1:

Input: "123456579"

Output: [123,456,579]

Example 2:

Input: "11235813"

Output: [1,1,2,3,5,8,13]

Example 3:

Input: "112358130"

Output: []

Explanation: The task is impossible.

Example 4:

Input: "0123"

Output: []

Explanation: Leading zeroes are not allowed, so "01", "2", "3" is not valid.

Example 5:

Input: "1101111"

Output: [110, 1, 111]

Explanation: The output [11, 0, 11, 11] would also be accepted.

Note:

1 <= S.length <= 200

S contains only digits.

Method 1: backtracking

class Solution {

public List<Integer> splitIntoFibonacci(String S) {

List<Integer> res = new ArrayList<>();

dfs(res, new ArrayList<>(), S, 0);

return res;

}

private void dfs(List<Integer> res, List<Integer> list, String S, int start){

if (!res.isEmpty()){

return;

}

if (start == S.length() && list.size() >= 3){

for (int i : list){

res.add(i);

}

return;

}

for (int i = start; i < S.length(); i++){

if (S.charAt(start) == '0' && i > start){//no leading 0

break;

}

long num = Long.parseLong(S.substring(start, i+1));

if (num > Integer.MAX\_VALUE){//must be signed integer

break;

}

int size = list.size();

if (size >= 2 && num > list.get(size - 2) + list.get(size - 1)){//early termination

break;

}

if (size < 2 || num == list.get(size - 2) + list.get(size - 1)){

list.add((int)num);

dfs(res, list, S, i + 1);

list.remove(list.size() - 1);

}

}

}

}

Best solution:

class Solution {

public List<Integer> splitIntoFibonacci(String S) {

List<List<Integer>> res = new ArrayList<>();

backtrack(res, S, new ArrayList<>(), 0);

if (res.size() == 0){

return new ArrayList<Integer>();

}

return res.get(0);

}

private void backtrack(List<List<Integer>> res, String S, List<Integer> item, int start){

if (start == S.length()){

if (item.size() >= 3){

res.add(new ArrayList<Integer>(item));

}

return;

}

for (int i = start; i < S.length(); i++){

String sub = S.substring(start, i+1);

if (sub.length() > 1 && sub.charAt(0) == '0'){//note that this condition should break

break;

}

long temp = Long.parseLong(sub);

if (temp > Integer.MAX\_VALUE){

break;

}

int num = (int)temp;

if (item.size() <= 1){

item.add(num);

backtrack(res, S, item, i+1);

item.remove(item.size() - 1);

}else{

int first = item.get(item.size() - 2);

int second = item.get(item.size() - 1);

if (first + second == num){

item.add(num);

backtrack(res, S, item, i+1);

item.remove(item.size() - 1);

}

}

}

}

}

## 843\_Guess.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This problem is an interactive problem new to the LeetCode platform.

We are given a word list of unique words, each word is 6 letters long, and one word in this list is chosen as secret.

You may call master.guess(word) to guess a word. The guessed word should have type string and must be from the original list with 6 lowercase letters.

This function returns an integer type, representing the number of exact matches (value and position) of your guess to the secret word. Also, if your guess is not in the given wordlist, it will return -1 instead.

For each test case, you have 10 guesses to guess the word. At the end of any number of calls, if you have made 10 or less calls to master.guess and at least one of these guesses was the secret, you pass the testcase.

Besides the example test case below, there will be 5 additional test cases, each with 100 words in the word list. The letters of each word in those testcases were chosen independently at random from 'a' to 'z', such that every word in the given word lists is unique.

Example 1:

Input: secret = "acckzz", wordlist = ["acckzz","ccbazz","eiowzz","abcczz"]

Explanation:

master.guess("aaaaaa") returns -1, because "aaaaaa" is not in wordlist.

master.guess("acckzz") returns 6, because "acckzz" is secret and has all 6 matches.

master.guess("ccbazz") returns 3, because "ccbazz" has 3 matches.

master.guess("eiowzz") returns 2, because "eiowzz" has 2 matches.

master.guess("abcczz") returns 4, because "abcczz" has 4 matches.

We made 5 calls to master.guess and one of them was the secret, so we pass the test case.

Note: Any solutions that attempt to circumvent the judge will result in disqualification.

Refer to Google Doc https://docs.google.com/document/d/1qxA2wps0IhVRWULulQ55W4SGPMu2AE5MkBB37h8Dr58/edit

当一个单词和其他单词match number为0的次数越多，那么这个单词越不好，因为match number为0时我们减少搜索空间的速度最慢。

假如现在有无限多长度为6的单词，对于word X，和他match number为0的单词有25^6这么多个，然而和X match number为1的单词则减少到了25^5 \* 6这么多个，

为2时为 C(6, 2) \* 25^4，以此类推，match number越大我们下一轮的搜索空间会越小，所以这里我们每一轮都挑选出当前搜索空间中和其他单词match number

为0的次数最少的单词作为guess word来猜，这样minimize了每次猜词的worse case

/\*\*

\* // This is the Master's API interface.

\* // You should not implement it, or speculate about its implementation

\* interface Master {

\* public int guess(String word) {}

\* }

\*/

class Solution {

class Pair {

String str;

int freq;

public Pair (String str, int freq){

this.str = str;

this.freq = freq;

}

}

public void findSecretWord(String[] wordlist, Master master) {

List<String> list = new ArrayList<>();

for (String word : wordlist){

list.add(word);

}

for (int i = 0; i < 10; i++){

Map<String, Integer> zeroMatch = new HashMap<>();

for (String s1 : list){

zeroMatch.putIfAbsent(s1, 0);

for (String s2 : list){

if (match(s1, s2) == 0){

zeroMatch.put(s1, zeroMatch.get(s1) + 1);

}

}

}

Pair p = new Pair("", 1000);

for (String key : zeroMatch.keySet()){

if (p.freq > zeroMatch.get(key)){

p = new Pair(key, p.freq);

}

}

int matchNum = master.guess(p.str);

if (matchNum == 6){

return;

}

List<String> temp = new ArrayList<>();

for (String str : list){

if (match(str, p.str) == matchNum){

temp.add(str);

}

}

list = temp;

}

}

private int match(String s1, String s2){

int count = 0;

for (int i = 0; i < s1.length(); i++){

if (s1.charAt(i) == s2.charAt(i)){

count++;

}

}

return count;

}

}

## 844\_Backspace.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings S and T, return if they are equal when both are typed into empty text editors. # means a backspace character.

Example 1:

Input: S = "ab#c", T = "ad#c"

Output: true

Explanation: Both S and T become "ac".

Example 2:

Input: S = "ab##", T = "c#d#"

Output: true

Explanation: Both S and T become "".

Example 3:

Input: S = "a##c", T = "#a#c"

Output: true

Explanation: Both S and T become "c".

Example 4:

Input: S = "a#c", T = "b"

Output: false

Explanation: S becomes "c" while T becomes "b".

Note:

1 <= S.length <= 200

1 <= T.length <= 200

S and T only contain lowercase letters and '#' characters.

Follow up:

Can you solve it in O(N) time and O(1) space?

Method 1:

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public boolean backspaceCompare(String S, String T) {

String s = format(S);

String t = format(T);

return s.equals(t);

}

private String format(String S){

Stack<Character> stack = new Stack<>();

int i = 0;

while (i < S.length()){

if (S.charAt(i) != '#'){

stack.push(S.charAt(i));

}else{

if (!stack.isEmpty()){

stack.pop();

}

}

i++;

}

StringBuilder sb = new StringBuilder();

for (char c : stack){

sb.append(c);

}

return sb.toString();

}

}

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public boolean backspaceCompare(String S, String T) {

String a = format(S);

String b = format(T);

return a.equals(b);

}

private String format(String s){

StringBuilder sb = new StringBuilder();

for (char c : s.toCharArray()){

if (c != '#'){

sb.append(c);

}else{

if (sb.length() > 0){

sb.deleteCharAt(sb.length() - 1);

}

}

}

return sb.toString();

}

}

Method 2:

Time complexixty: O(n)

Space complexity: O(1)

class Solution {

public boolean backspaceCompare(String S, String T) {

int i = S.length()- 1;

int j = T.length() - 1;

int skipS = 0;

int skipT = 0;

while (i >= 0 || j >= 0){

while (i >= 0){

if (S.charAt(i) == '#'){

skipS++;

i--;

}else if (skipS > 0){

skipS--;

i--;

}else{

break;

}

}

while (j >= 0){

if (T.charAt(j) == '#'){

skipT++;

j--;

}else if (skipT > 0){

skipT--;

j--;

}else{

break;

}

}

if ((i >= 0) ^ (j >= 0)){

return false;

}

if (i >= 0 && j >= 0 && S.charAt(i) != T.charAt(j)){

return false;

}

i--;

j--;

}

return true;

}

}

## 845\_Longest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Let's call any (contiguous) subarray B (of A) a mountain if the following properties hold:

B.length >= 3

There exists some 0 < i < B.length - 1 such that B[0] < B[1] < ... B[i-1] < B[i] > B[i+1] > ... > B[B.length - 1]

(Note that B could be any subarray of A, including the entire array A.)

Given an array A of integers, return the length of the longest mountain.

Return 0 if there is no mountain.

Example 1:

Input: [2,1,4,7,3,2,5]

Output: 5

Explanation: The largest mountain is [1,4,7,3,2] which has length 5.

Example 2:

Input: [2,2,2]

Output: 0

Explanation: There is no mountain.

Note:

0 <= A.length <= 10000

0 <= A[i] <= 10000

Follow up:

Can you solve it using only one pass?

Can you solve it in O(1) space?

Method 1:

class Solution {

public int longestMountain(int[] A) {

if (A.length < 3){

return 0;

}

int longest = 0;

int i = 0;

while (i < A.length - 1){

while (i < A.length - 1 && A[i+1] <= A[i]){

i++;

}

int start = i;

while (i < A.length - 1 && A[i+1] > A[i]){

i++;

}

int peak = i;

while (i < A.length - 1 && A[i+1] < A[i]){

i++;

}

int end = i;

if (start < peak && peak < end){

longest = Math.max(longest, i - start + 1);

}

}

return longest;

}

}

class Solution {

public int longestMountain(int[] A) {

if (A.length < 3){

return 0;

}

int max = 0;

int i = 0;

int j = 0;

while (i < A.length){

j = i+1;

boolean up = false;

boolean down = false;

while (j < A.length && A[j] > A[j-1]){

up = true;

j++;

}

if (j == i+1){

i++;

continue;

}

while (j < A.length && A[j] < A[j-1]){

down = true;

j++;

}

if (up && down){

max = Math.max(max, j - i);

}

i = j - 1;

}

return max;

}

}

Method 2: Similar as

Maximize Distance to Closest Person

Shortest Distance to a Character

Time compleixty: O(n)

Space complexity: O(n)

class Solution {

public int longestMountain(int[] A) {

int n = A.length;

int[] increase = new int[n];

int[] decrease = new int[n];

for (int i = 1; i < n; i++){

if (A[i] > A[i-1]){

increase[i] = increase[i-1] + 1;

}

}

for (int i = n-2; i >= 0; i--){

if (A[i] > A[i+1]){

decrease[i] = decrease[i+1] + 1;

}

}

int max = 0;

for (int i = 0; i < n; i++){

if (increase[i] > 0 && decrease[i] > 0){

max = Math.max(max, increase[i] + decrease[i] + 1);

}

}

return max;

}

}

class Solution {

public int longestMountain(int[] A) {

int n = A.length;

int[] increase = new int[n];

int[] decrease = new int[n];

for (int i = 1; i < n; i++){

if (A[i] > A[i-1]){

increase[i] = increase[i-1] + 1;

}

}

int max = 0;

for (int i = n-2; i >= 0; i--){

if (A[i] > A[i+1]){

decrease[i] = decrease[i+1] + 1;

}

if (increase[i] > 0 && decrease[i] > 0){

max = Math.max(max, increase[i] + decrease[i] + 1);

}

}

return max;

}

}

Best solution

Time complexity: O(n)

Space complexiyt: O(1)

class Solution {

public int longestMountain(int[] A) {

int max = 0;

int up = 0;

int down = 0;

for (int i = 1; i < A.length; i++){

if (down > 0 && A[i] > A[i-1] || A[i] == A[i-1]){//refresh at the starting point of each mountain

down = 0;

up = 0;

}

if (A[i] > A[i-1]){

up++;

}

if (A[i] < A[i-1]){

down++;

}

if (up > 0 && down > 0){

max = Math.max(max, up + down + 1);

}

}

return max;

}

}

## 846\_Hand.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Alice has a hand of cards, given as an array of integers.

Now she wants to rearrange the cards into groups so that each group is size W, and consists of W consecutive cards.

Return true if and only if she can.

Example 1:

Input: hand = [1,2,3,6,2,3,4,7,8], W = 3

Output: true

Explanation: Alice's hand can be rearranged as [1,2,3],[2,3,4],[6,7,8].

Example 2:

Input: hand = [1,2,3,4,5], W = 4

Output: false

Explanation: Alice's hand can't be rearranged into groups of 4.

Note:

1 <= hand.length <= 10000

0 <= hand[i] <= 10^9

1 <= W <= hand.length

Method 1:

Time complexity: O(NlogN + NWlogW) N is the number of cards

Space complexity: O(N) number of cardsd

class Solution {

public boolean isNStraightHand(int[] hand, int W) {

int n = hand.length;

if (n % W != 0){

return false;

}

Map<Integer, Integer> map = new TreeMap<>();

for (int i : hand){

map.put(i, map.getOrDefault(i, 0) + 1);

}

for (int key : map.keySet()){

while (map.get(key) > 0){

for (int i = 0; i < W; i++){

int index = key + i;

if (!map.containsKey(index) || map.get(index) <= 0){

return false;

}

map.put(index, map.get(index) - 1);

}

}

}

return true;

}

}

Method 2:

Time complexity: O(NlogN + MWLogW) M is number of different cards

class Solution {

public boolean isNStraightHand(int[] hand, int W) {

int n = hand.length;

if (n % W != 0){

return false;

}

Map<Integer, Integer> map = new TreeMap<>();

for (int i : hand){

map.put(i, map.getOrDefault(i, 0) + 1);

}

for (int key : map.keySet()){

if (map.get(key) > 0){ // remove one loop

for (int i = W - 1; i >= 0; i--){

int index = key + i;

if (!map.containsKey(index) || map.get(index) < map.get(key)){

return false;

}

map.put(index, map.get(index) - map.get(key));

}

}

}

return true;

}

}

## 847\_Shortest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

An undirected, connected graph of N nodes (labeled 0, 1, 2, ..., N-1) is given as graph.

graph.length = N, and j != i is in the list graph[i] exactly once, if and only if nodes i and j are connected.

Return the length of the shortest path that visits every node. You may start and stop at any node, you may revisit nodes multiple

times, and you may reuse edges.

Example 1:

Input: [[1,2,3],[0],[0],[0]]

Output: 4

Explanation: One possible path is [1,0,2,0,3]

Example 2:

Input: [[1],[0,2,4],[1,3,4],[2],[1,2]]

Output: 4

Explanation: One possible path is [0,1,4,2,3]

Note:

1 <= graph.length <= 12

0 <= graph[i].length < graph.length

Method 1: BFS

https://leetcode.com/problems/shortest-path-visiting-all-nodes/discuss/152679/Short-Java-Solution-BFS-with-a-Set

https://leetcode.com/problems/shortest-path-visiting-all-nodes/solution/

https://leetcode.com/problems/shortest-path-visiting-all-nodes/discuss/135809/Fast-BFS-Solution-(46ms)-Clear-Detailed-Explanation-Included

Key points:

1. Use bit to record visited node to optimize code, otherwise it will TLE if use Set<Integer>

2. Use string to save path candidate

3. Use separate ":" to avoid repeating route since big mask alone can't do the job, it has to include vertex info

Note that to avoid repeating route: the following string format is used

e.g. 0 -> 1 ==> set add 11:1

1 -> 0 ==> set add 11:0

to avoid 0 -> 1 -> 0 -> 1

class Solution {

public int shortestPathLength(int[][] graph) {

int N = graph.length;

int count = 0;

int target = 0;

Set<String> set = new HashSet<>();

Queue<int[]> queue = new LinkedList<>();

for (int i = 0; i < N; i++){

target |= (1 << i);

int[] pair = new int[]{(1 << i), i};

queue.offer(pair);

set.add(pair[0] + ":" + pair[1]);

}

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

int[] pair = queue.poll();

if (target == pair[0]){

return count;

}else{

int currPath = pair[0];

int currNode = pair[1];

for (int nei : graph[currNode]){

int newPath = (1 << nei) | currPath;

String key = newPath + ":" + nei;

if (!set.contains(key)){

int[] newPair = new int[]{newPath, nei};

queue.offer(newPair);

set.add(key);

}

}

}

}

count++;

}

return -1;

}

}

## 848\_Shifting.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have a string S of lowercase letters, and an integer array shifts.

Call the shift of a letter, the next letter in the alphabet, (wrapping around so that 'z' becomes 'a').

For example, shift('a') = 'b', shift('t') = 'u', and shift('z') = 'a'.

Now for each shifts[i] = x, we want to shift the first i+1 letters of S, x times.

Return the final string after all such shifts to S are applied.

Example 1:

Input: S = "abc", shifts = [3,5,9]

Output: "rpl"

Explanation:

We start with "abc".

After shifting the first 1 letters of S by 3, we have "dbc".

After shifting the first 2 letters of S by 5, we have "igc".

After shifting the first 3 letters of S by 9, we have "rpl", the answer.

Note:

1 <= S.length = shifts.length <= 20000

0 <= shifts[i] <= 10 ^ 9

Method 1: Brute Force

Time complexity: O(n^2)

Space complexity: O(1)

class Solution {

public String shiftingLetters(String S, int[] shifts) {

StringBuilder res = new StringBuilder(S);

for (int i = 0; i < shifts.length; i++){

for (int j = 0; j <= i; j++){

char c = (char)((res.charAt(j) - 'a' + shifts[i])%26 + 'a');

res.setCharAt(j, c);

}

}

return res.toString();

}

}

Method 2: Prefix Sum Best solution

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public String shiftingLetters(String S, int[] shifts) {

StringBuilder res = new StringBuilder(S);

for (int i = shifts.length - 2; i >= 0; i--){

shifts[i] = (shifts[i+1] + shifts[i]) % 26;

}

for (int i = 0; i < shifts.length; i++){

char c = (char)((S.charAt(i) - 'a' + shifts[i]) % 26 + 'a');

res.setCharAt(i, c);

}

return res.toString();

}

}

class Solution {

public String shiftingLetters(String S, int[] shifts) {

int n = shifts.length;

StringBuilder sb = new StringBuilder();

int[] preSum = new int[n];

int sum = 0;

for (int i = n- 1; i >= 0; i--){

sum = (sum + shifts[i]) % 26;

preSum[i] = sum;

}

for (int i = 0; i < n; i++){

int delta = ((int)(S.charAt(i) - 'a') + preSum[i]) % 26;

char c = (char)('a' + delta);

sb.append(c);

}

return sb.toString();

}

}

## 849\_Maximize.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a row of seats, 1 represents a person sitting in that seat, and 0 represents that the seat is empty.

There is at least one empty seat, and at least one person sitting.

Alex wants to sit in the seat such that the distance between him and the closest person to him is maximized.

Return that maximum distance to closest person.

Example 1:

Input: [1,0,0,0,1,0,1]

Output: 2

Explanation:

If Alex sits in the second open seat (seats[2]), then the closest person has distance 2.

If Alex sits in any other open seat, the closest person has distance 1.

Thus, the maximum distance to the closest person is 2.

Example 2:

Input: [1,0,0,0]

Output: 3

Explanation:

If Alex sits in the last seat, the closest person is 3 seats away.

This is the maximum distance possible, so the answer is 3.

Note:

1 <= seats.length <= 20000

seats contains only 0s or 1s, at least one 0, and at least one 1

Method 1: Group by Zero

class Solution {

public int maxDistToClosest(int[] seats) {

int max = 1;

int i = 0;

int start = 0;

while (i < seats.length){

if (seats[i] == 1){

max = Math.max(max, (i - start) / 2 );

start = i;

}

i++;

}

//deal with corner case

int count = 0;

for (int j = 0; j < seats.length; j++){

if (seats[j] == 1){

max = Math.max(max, count);

break;

}

count++;

}

count = 0;

for (int j = seats.length - 1; j >= 0; j--){

if (seats[j] == 1){

max = Math.max(max, count);

break;

}

count++;

}

return max;

}

}

Group by Zero

class Solution {

public int maxDistToClosest(int[] seats) {

int N = seats.length;

int K = 0; //current longest group of empty seats

int ans = 0;

for (int i = 0; i < N; ++i) {

if (seats[i] == 1) {

K = 0;

} else {

K++;

ans = Math.max(ans, (K + 1) / 2);

}

}

for (int i = 0; i < N; ++i) if (seats[i] == 1) {

ans = Math.max(ans, i);

break;

}

for (int i = N-1; i >= 0; --i) if (seats[i] == 1) {

ans = Math.max(ans, N - 1 - i);

break;

}

return ans;

}

}

Method 2: Best solution

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public int maxDistToClosest(int[] seats) {

int count = 0;

int res = 0;

int firstInd = 0;

int lastInd = seats.length - 1;

for (int i = 0; i < seats.length; i++){

if (seats[i] == 0){

count++;

}else{

res = count;

firstInd = i;

break;

}

}

count = 0;

for (int i = seats.length - 1; i >= 0; i--){

if (seats[i] == 0){

count++;

}else{

res = Math.max(res, count);

lastInd = i;

break;

}

}

if (firstInd == lastInd){

return res;

}

count = 0;

for (int i = firstInd + 1; i <= lastInd; i++){

if (seats[i] == 0){

count++;

}else{

if (count % 2 == 0){

res = Math.max(res, count/2);

}else{

res = Math.max(res, count/2+1);

}

count = 0;

}

}

return res;

}

}

Method 3: Similar as https://github.com/optimisea/Leetcode/blob/master/Java/821\_Shortest.java

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int maxDistToClosest(int[] seats) {

int n = seats.length;

int pos = -2 \* n;

int[] res = new int[n];

for (int i = 0; i < seats.length; i++){

if (seats[i] == 1){

pos = i;

}

res[i] = i - pos;

}

pos = 2 \* n;

for (int i = n - 1; i >= 0; i--){

if (seats[i] == 1){

pos = i;

}

res[i] = Math.min(res[i], pos - i);

}

int max = 0;

for (int i = 0; i < n; i++){

max = Math.max(max, res[i]);

}

return max;

}

}

## 85\_MaximalRectangle.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

For example, given the following matrix:

1 0 1 0 0

1 0 1 1 1

1 1 1 1 1

1 0 0 1 0

Return 6.

Method: Monotonic Stack O(m \* n)

class Solution {

public int maximalRectangle(char[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0){

return 0;

}

int m = matrix.length;

int n = matrix[0].length;

int[][] heights = new int[m][n];

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (matrix[i][j] == '0'){

heights[i][j] = 0;

}else{

heights[i][j] = (i == 0) ? 1 : heights[i-1][j] + 1;

}

}

}

int max = 0;

for (int i = 0; i < m; i++){

int area = largestRectangleArea(heights[i]);

if (area > max){

max = area;

}

}

return max;

}

private int largestRectangleArea(int[] heights) {

if (heights == null || heights.length == 0){

return 0;

}

Stack<Integer> stack = new Stack<>();

int max = 0;

for (int i = 0; i <= heights.length; i++){

int curr = (i == heights.length) ? -1 : heights[i];

while (!stack.isEmpty() && curr < heights[stack.peek()]){

int h = heights[stack.pop()];

int w = stack.isEmpty() ? i : i - stack.peek() - 1;

max = Math.max(max, h \* w);

}

stack.push(i);

}

return max;

}

}

## 851\_Loud.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a group of N people (labelled 0, 1, 2, ..., N-1), each person has different amounts of money, and different levels of quietness.

For convenience, we'll call the person with label x, simply "person x".

We'll say that richer[i] = [x, y] if person x definitely has more money than person y. Note that richer may only be a subset of valid observations.

Also, we'll say quiet[x] = q if person x has quietness q.

Now, return answer, where answer[x] = y if y is the least quiet person (that is, the person y with the smallest value of quiet[y]), among all people who definitely have equal to or more money than person x.

Example 1:

Input: richer = [[1,0],[2,1],[3,1],[3,7],[4,3],[5,3],[6,3]], quiet = [3,2,5,4,6,1,7,0]

Output: [5,5,2,5,4,5,6,7]

Explanation:

answer[0] = 5.

Person 5 has more money than 3, which has more money than 1, which has more money than 0.

The only person who is quieter (has lower quiet[x]) is person 7, but

it isn't clear if they have more money than person 0.

answer[7] = 7.

Among all people that definitely have equal to or more money than person 7

(which could be persons 3, 4, 5, 6, or 7), the person who is the quietest (has lower quiet[x])

is person 7.

The other answers can be filled out with similar reasoning.

Note:

1 <= quiet.length = N <= 500

0 <= quiet[i] < N, all quiet[i] are different.

0 <= richer.length <= N \* (N-1) / 2

0 <= richer[i][j] < N

richer[i][0] != richer[i][1]

richer[i]'s are all different.

The observations in richer are all logically consistent.

Method 1: DFS + memo

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int[] loudAndRich(int[][] richer, int[] quiet) {

int n = quiet.length;

int[] res = new int[n];

Map<Integer, Set<Integer>> map = new HashMap<>();

for (int[] row : richer){

if (!map.containsKey(row[1])){

Set<Integer> set = new HashSet<Integer>();

map.put(row[1], set);

}

map.get(row[1]).add(row[0]);

}

Map<Integer, Integer> memo = new HashMap<>();

for (int i = 0; i < n; i++){

res[i] = i;

if (map.containsKey(i)){

res[i] = dfs(map, quiet, i, memo);

}

}

return res;

}

private int dfs(Map<Integer, Set<Integer>> map, int[] quiet, int i, Map<Integer, Integer> memo){

if (memo.containsKey(i)){

return memo.get(i);

}

if (!map.containsKey(i)){

return i;

}

Set<Integer> set = map.get(i);

int minIndex = i;

for (int key : set){

int index = dfs(map, quiet, key, memo);

if (quiet[index] < quiet[minIndex]){

minIndex = index;

}

}

memo.put(i, minIndex);

return minIndex;

}

}

Method 2: use res as memo + DFS

class Solution {

public int[] loudAndRich(int[][] richer, int[] quiet) {

int n = quiet.length;

int[] res = new int[n];

Map<Integer, Set<Integer>> map = new HashMap<>();

for (int[] row : richer){

if (!map.containsKey(row[1])){

Set<Integer> set = new HashSet<Integer>();

map.put(row[1], set);

}

map.get(row[1]).add(row[0]);

}

Arrays.fill(res, -1);

for (int i = 0; i < n; i++){

if (map.containsKey(i)){

res[i] = dfs(map, quiet, i, res);

}else{

res[i] = i;

}

}

return res;

}

private int dfs(Map<Integer, Set<Integer>> map, int[] quiet, int i, int[] res){

if (res[i] != -1){

return res[i];

}

if (!map.containsKey(i)){

return i;

}

Set<Integer> set = map.get(i);

res[i] = i;

for (int key : set){

int index = dfs(map, quiet, key, res);

if (quiet[index] < quiet[res[i]]){

res[i] = index;

}

}

return res[i];

}

}

Intuition

Consider the directed graph with edge x -> y if y is richer than x.

For each person x, we want the quietest person in the subtree at x.

Algorithm

Construct the graph described above, and say dfs(person) is the quietest person in the subtree at person. Notice because the statements are logically consistent, the graph must be a DAG - a directed graph with no cycles.

Now dfs(person) is either person, or min(dfs(child) for child in person). That is to say, the quietest person in the subtree is either the person itself, or the quietest person in some subtree of a child of person.

We can cache values of dfs(person) as answer[person], when performing our post-order traversal of the graph. That way, we don't repeat work. This technique reduces a quadratic time algorithm down to linear time.

class Solution {

ArrayList<Integer>[] graph;

int[] answer;

int[] quiet;

public int[] loudAndRich(int[][] richer, int[] quiet) {

int N = quiet.length;

graph = new ArrayList[N];

answer = new int[N];

this.quiet = quiet;

for (int node = 0; node < N; ++node)

graph[node] = new ArrayList<Integer>();

for (int[] edge: richer)

graph[edge[1]].add(edge[0]);

Arrays.fill(answer, -1);

for (int node = 0; node < N; ++node)

dfs(node);

return answer;

}

public int dfs(int node) {

if (answer[node] == -1) {

answer[node] = node;

for (int child: graph[node]) {

int cand = dfs(child);

if (quiet[cand] < quiet[answer[node]])

answer[node] = cand;

}

}

return answer[node];

}

}

## 852\_Peak.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Let's call an array A a mountain if the following properties hold:

A.length >= 3

There exists some 0 < i < A.length - 1 such that A[0] < A[1] < ... A[i-1] < A[i] > A[i+1] > ... > A[A.length - 1]

Given an array that is definitely a mountain, return any i such that A[0] < A[1] < ... A[i-1] < A[i] > A[i+1] > ... > A[A.length - 1].

Example 1:

Input: [0,1,0]

Output: 1

Example 2:

Input: [0,2,1,0]

Output: 1

Method 1:

class Solution {

public int peakIndexInMountainArray(int[] A) {

int start = 0;

int end = A.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (A[mid] < A[mid+1]){

start = mid;

}else{

end = mid;

}

}

if (A[start] < A[end]){

return end;

}

return start;

}

}

class Solution {

public int peakIndexInMountainArray(int[] A) {

int start = 0;

int end = A.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (A[mid] > A[mid-1] && A[mid] > A[mid+1]){

return mid;

}else if (A[mid] > A[mid-1] && A[mid] < A[mid+1]){

start = mid;

}else{

end = mid;

}

}

if (A[start] > A[end]){

return start;

}

return end;

}

}

Method 2:

class Solution {

public int peakIndexInMountainArray(int[] A) {

for (int i = 0; i < A.length - 1; i++){

if (A[i] > A[i+1]){

return i;

}

}

return 0;

}

}

## 853\_Car.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

N cars are going to the same destination along a one lane road. The destination is target miles away.

Each car i has a constant speed speed[i] (in miles per hour), and initial position position[i] miles towards the target along the road.

A car can never pass another car ahead of it, but it can catch up to it, and drive bumper to bumper at the same speed.

The distance between these two cars is ignored - they are assumed to have the same position.

A car fleet is some non-empty set of cars driving at the same position and same speed. Note that a single car is also a car fleet.

If a car catches up to a car fleet right at the destination point, it will still be considered as one car fleet.

How many car fleets will arrive at the destination?

Example 1:

Input: target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]

Output: 3

Explanation:

The cars starting at 10 and 8 become a fleet, meeting each other at 12.

The car starting at 0 doesn't catch up to any other car, so it is a fleet by itself.

The cars starting at 5 and 3 become a fleet, meeting each other at 6.

Note that no other cars meet these fleets before the destination, so the answer is 3.

Note:

0 <= N <= 10 ^ 4

0 < target <= 10 ^ 6

0 < speed[i] <= 10 ^ 6

0 <= position[i] < target

All initial positions are different.

class Solution {

public int carFleet(int target, int[] position, int[] speed) {

Map<Integer, Double> map = new TreeMap<>();

for (int i = 0; i < position.length; i++){

map.put(-position[i], (double)(target - position[i]) / speed[i]);

}

int res = 0;

double curr = 0.0;

for (double val : map.values()){

if (val > curr){

curr = val;

res++;

}

}

return res;

}

}

class Solution {

class Car {

int pos;

double time;

public Car (int pos, double time){

this.pos = pos;

this.time = time;

}

}

public int carFleet(int target, int[] position, int[] speed) {

int n = position.length;

if (n == 0){

return 0;

}

Car[] cars = new Car[n];

for (int i = 0; i < n; i++){

cars[i] = new Car(position[i], (double)(target - position[i]) / speed[i]);

}

Arrays.sort(cars, new Comparator<Car>(){

public int compare (Car car1, Car car2){

return car2.pos - car1.pos;

}

});

int res = 0;

double curr = 0.0;

for (int i = 0; i < n; i++){

if (cars[i].time > curr){

res++;

curr = cars[i].time;

}

}

return res;

}

}

## 854\_Similar.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Strings A and B are K-similar (for some non-negative integer K) if we can swap the positions of two letters in A exactly K times so

that the resulting string equals B.

Given two anagrams A and B, return the smallest K for which A and B are K-similar.

Example 1:

Input: A = "ab", B = "ba"

Output: 1

Example 2:

Input: A = "abc", B = "bca"

Output: 2

Example 3:

Input: A = "abac", B = "baca"

Output: 2

Example 4:

Input: A = "aabc", B = "abca"

Output: 2

Note:

1 <= A.length == B.length <= 20

A and B contain only lowercase letters from the set {'a', 'b', 'c', 'd', 'e', 'f'}

Classic BFS: swap only pair at every step and use bfs to guaranteee shortest path

find the first mismatch character at j , then find the second one at k, only swap when

curr.charAt(k) == B.charAt(j) && curr.charAt(k) != B.charAt(k)

Method 1: BFS

class Solution {

public int kSimilarity(String A, String B) {

Queue<String> queue = new LinkedList<>();

Set<String> seen = new HashSet<>();

queue.offer(A);

seen.add(A);

int res = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

String curr = queue.poll();

if (curr.equals(B)){

return res;

}

int j = 0;

while (j < curr.length() && curr.charAt(j) == B.charAt(j)){

j++;

}

for (int k = j + 1; k < curr.length(); k++){

if (curr.charAt(k) == B.charAt(j) && curr.charAt(k) != B.charAt(k)){//this is the key

String next = swap(curr, j, k);//return string to ensure that curr won't change

if (!seen.contains(next)){

queue.offer(next);

seen.add(next);

}

}

}

}

res++;

}

return res;

}

private String swap(String curr, int j, int k){

char[] arr = curr.toCharArray();

char temp = arr[j];

arr[j] = arr[k];

arr[k] = temp;

return new String(arr);

}

}

Method 2: DFS + memo

class Solution {

public int kSimilarity(String A, String B) {

Map<String, Integer> memo = new HashMap<>();

return minStep(A.toCharArray(), B, memo, 0);

}

private int minStep(char[] A, String B, Map<String, Integer> memo, int i){

String sa = new String(A);

if (sa.equals(B)){

return 0;

}

if (memo.containsKey(sa)){

return memo.get(sa);

}

while (i < sa.length() && A[i] == B.charAt(i)){

i++;

}

int min = Integer.MAX\_VALUE;

for (int j = i + 1; j < sa.length(); j++){

if (sa.charAt(j) == B.charAt(i)&& sa.charAt(j) != B.charAt(j)){

swap(A, i, j);

int next = minStep(A, B, memo, i+1);

if (next != Integer.MAX\_VALUE){

min = Math.min(min, next + 1);

}

swap(A, i, j);

}

}

memo.put(sa, min);

return min;

}

private void swap(char[] cs, int i, int j) {

char temp = cs[i];

cs[i] = cs[j];

cs[j] = temp;

}

}

## 855\_Exam.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In an exam room, there are N seats in a single row, numbered 0, 1, 2, ..., N-1.

When a student enters the room, they must sit in the seat that maximizes the distance to the closest person. If there are multiple such seats, they sit in the seat with the lowest number. (Also, if no one is in the room, then the student sits at seat number 0.)

Return a class ExamRoom(int N) that exposes two functions: ExamRoom.seat() returning an int representing what seat the student sat in, and ExamRoom.leave(int p) representing that the student in seat number p now leaves the room. It is guaranteed that any calls to ExamRoom.leave(p) have a student sitting in seat p.

Example 1:

Input: ["ExamRoom","seat","seat","seat","seat","leave","seat"], [[10],[],[],[],[],[4],[]]

Output: [null,0,9,4,2,null,5]

Explanation:

ExamRoom(10) -> null

seat() -> 0, no one is in the room, then the student sits at seat number 0.

seat() -> 9, the student sits at the last seat number 9.

seat() -> 4, the student sits at the last seat number 4.

seat() -> 2, the student sits at the last seat number 2.

leave(4) -> null

seat() -> 5, the student​​​​​​​ sits at the last seat number 5.

​​​​​​​

Note:

1 <= N <= 10^9

ExamRoom.seat() and ExamRoom.leave() will be called at most 10^4 times across all test cases.

Calls to ExamRoom.leave(p) are guaranteed to have a student currently sitting in seat number p.

Time complexity: O(n) for both

class ExamRoom {

private List<Integer> list;

private int n;

public ExamRoom(int N) {

list = new ArrayList<>();

n = N;

}

public int seat() {

if (list.isEmpty()){

list.add(0);

return 0;

}

int max = Math.max(list.get(0), n - 1 - list.get(list.size() - 1)); // to handle the corner cases: before the first and after the last

for (int i = 0; i < list.size() - 1 ; i++){

max = Math.max(max, (list.get(i+1) - list.get(i)) / 2);

}

//first case: in the first range

if (list.get(0) == max){

list.add(0, 0);

return 0;

}

//second case: in the middle

for (int i = 0; i < list.size() - 1; i++){

if ((list.get(i+1) - list.get(i)) / 2 == max){

list.add(i+1, list.get(i) + max);

return list.get(i+1);

}

}

//third case: in the last range

list.add(n-1);

return n-1;

}

public void leave(int p) {

list.remove(new Integer(p));//remove Object, not index if we want to remove index, we need use loop

//list.remove(Integer.valueOf(p));

}

/\*

for (int i = 0; i < list.size(); i++){

if (list.get(i) == p){

list.remove(i);

}

}

\*/

}

/\*\*

\* Your ExamRoom object will be instantiated and called as such:

\* ExamRoom obj = new ExamRoom(N);

\* int param\_1 = obj.seat();

\* obj.leave(p);

\*/

## 856\_Score.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a balanced parentheses string S, compute the score of the string based on the following rule:

() has score 1

AB has score A + B, where A and B are balanced parentheses strings.

(A) has score 2 \* A, where A is a balanced parentheses string.

Example 1:

Input: "()"

Output: 1

Example 2:

Input: "(())"

Output: 2

Example 3:

Input: "()()"

Output: 2

Example 4:

Input: "(()(()))"

Output: 6

Note:

S is a balanced parentheses string, containing only ( and ).

2 <= S.length <= 50

Method 1: Better

class Solution {

public int scoreOfParentheses(String S) {

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < S.length(); i++){

char c = S.charAt(i);

if (c == '('){

stack.push(-1);

}else{

int cur = 0; //store the sum between ( and ), if there is no value, push 1 otherwise push value \* 2

while (stack.peek() != -1){

cur += stack.pop();

}

stack.pop();

stack.push(cur == 0 ? 1 : cur \* 2);

}

}

int ans = 0;

while (!stack.isEmpty()){

ans += stack.pop();

}

return ans;

}

}

Method 2:

class Solution {

public int scoreOfParentheses(String S) {

Stack<Integer> stack = new Stack<>();

int res = 0;

int cur = 0;//store

for (int i = 0; i < S.length(); i++){

char c = S.charAt(i);

if (c == '('){

stack.push(cur);

cur = 0;

}else{

cur = stack.pop();

if (S.charAt(i-1) == '('){

cur += 1;

}else{

cur += res \* 2;

}

res = cur;

}

}

return res;

}

}

Method 3:

Our goal is to maintain the score at the current depth we are on. When we see an opening bracket, we increase our depth,

and our score at the new depth is 0. When we see a closing bracket,

we add twice the score of the previous deeper part - except when counting (), which has a score of 1.

class Solution {

public int scoreOfParentheses(String S) {

Stack<Integer> stack = new Stack<>();

stack.push(0);

for (char c : S.toCharArray()){

if (c == '('){

stack.push(0);

}else{

int deeperLevel = stack.pop();

int currLevel = stack.pop();

stack.push(currLevel + Math.max(deeperLevel \* 2, 1));

}

}

return stack.pop();

}

}

class Solution {

public int scoreOfParentheses(String S) {

int res[] = new int[30], i = 0;

for (char c : S.toCharArray())

if (c == '(')

res[++i] = 0;

else

res[i - 1] += Math.max(res[i--] \* 2, 1);

return res[0];

}

}

Best solution:

O(1)

class Solution {

public int scoreOfParentheses(String S) {

int res = 0;

int layer = 0;

for (int i = 0; i < S.length(); i++){

if (S.charAt(i) == '('){

layer++;

}else{

layer--;

}

if (S.charAt(i) == '(' && S.charAt(i+1) == ')'){

res += 1 << (layer - 1);

}

}

return res;

}

}

## 857\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are N workers. The i-th worker has a quality[i] and a minimum wage expectation wage[i].

Now we want to hire exactly K workers to form a paid group. When hiring a group of K workers, we must pay them according

to the following rules:

Every worker in the paid group should be paid in the ratio of their quality compared to other workers in the paid group.

Every worker in the paid group must be paid at least their minimum wage expectation.

Return the least amount of money needed to form a paid group satisfying the above conditions.

Example 1:

Input: quality = [10,20,5], wage = [70,50,30], K = 2

Output: 105.00000

Explanation: We pay 70 to 0-th worker and 35 to 2-th worker.

Example 2:

Input: quality = [3,1,10,10,1], wage = [4,8,2,2,7], K = 3

Output: 30.66667

Explanation: We pay 4 to 0-th worker, 13.33333 to 2-th and 3-th workers seperately.

Note:

1 <= K <= N <= 10000, where N = quality.length = wage.length

1 <= quality[i] <= 10000

1 <= wage[i] <= 10000

Answers within 10^-5 of the correct answer will be considered correct.

class Solution {

class Worker{

int q;

double ratio;

public Worker(int q, double ratio){

this.q = q;

this.ratio = ratio;

}

}

public double mincostToHireWorkers(int[] quality, int[] wage, int K) {

//Step 1: sort by ratio

int N = quality.length;

Worker[] workers = new Worker[N];

for (int i = 0; i < N; ++i){

workers[i] = new Worker(quality[i], (double) (wage[i]) / quality[i]);

}

Arrays.sort(workers, new Comparator<Worker>(){

public int compare (Worker w1, Worker w2){//must be int, can't be double

//return (int)(w1.ratio - w2.ratio); //not working

return Double.compare(w1.ratio, w2.ratio);

}

});

// Arrays.sort(workers, (a, b) -> Double.compare(a.ratio, b.ratio)); // Double.compare static function returns int

//Step 2: maintain max priority queue for quality, use negative value and minPQ to implement max PQ

Queue<Integer> minPQ = new PriorityQueue<>();

double res = Double.MAX\_VALUE;

int sumQ = 0;

for (Worker worker : workers){

sumQ += worker.q;

minPQ.offer(-worker.q);

if (minPQ.size() > K){

sumQ += minPQ.poll();

}

if (minPQ.size() == K){

res = Math.min(res, sumQ \* worker.ratio);

}

}

return res;

}

}

class Solution {

class Pair{

int quality;

double ratio;

public Pair(int quality, double ratio){

this.quality = quality;

this.ratio = ratio;

}

}

public double mincostToHireWorkers(int[] quality, int[] wage, int K) {

int N = quality.length;

Pair[] pairs = new Pair[N];

for (int i = 0; i < N; i++){

pairs[i] = new Pair(quality[i], (double)wage[i]/quality[i]);

}

Arrays.sort(pairs, new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return Double.compare(p1.ratio, p2.ratio);

}

});

Queue<Integer> pq = new PriorityQueue<>(new Comparator<Integer>(){

public int compare (Integer i1, Integer i2){

return (int)(i2 - i1);

}

});

double res = Double.MAX\_VALUE;

int sumQ = 0;

for (Pair p : pairs){

sumQ += p.quality;

pq.offer(p.quality);

if (pq.size() > K){

sumQ -= pq.poll();

}

if (pq.size() == K){

res = Math.min(res, sumQ \* p.ratio);

}

}

return res;

}

}

## 858\_Mirror.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There is a special square room with mirrors on each of the four walls. Except for the southwest corner, there are receptors on each of the remaining corners, numbered 0, 1, and 2.

The square room has walls of length p, and a laser ray from the southwest corner first meets the east wall at a distance q from the 0th receptor.

Return the number of the receptor that the ray meets first. (It is guaranteed that the ray will meet a receptor eventually.)

Example 1:

class Solution {

public int mirrorReflection(int p, int q) {

while (p % 2 == 0 && q % 2 == 0){

p /= 2;

q /= 2;

}

if (q % 2 == 0){

return 0;

}else if (p % 2 == 0){

return 2;

}

return 1;

}

}

## 859\_Buddy.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two strings A and B of lowercase letters, return true if and only if we can swap two letters in A so that the result equals B.

Example 1:

Input: A = "ab", B = "ba"

Output: true

Example 2:

Input: A = "ab", B = "ab"

Output: false

Example 3:

Input: A = "aa", B = "aa"

Output: true

Example 4:

Input: A = "aaaaaaabc", B = "aaaaaaacb"

Output: true

Example 5:

Input: A = "", B = "aa"

Output: false

Note:

0 <= A.length <= 20000

0 <= B.length <= 20000

A and B consist only of lowercase letters.

Method:

class Solution {

public boolean buddyStrings(String A, String B) {

if (A.length() != B.length()){

return false;

}

int index1 = -1;

int index2 = -1;

for (int i = 0; i < A.length(); i++){

if (A.charAt(i) != B.charAt(i)){

if (index1 == -1){

index1 = i;

}else if (index2 == -1){

index2 = i;

if (A.charAt(index1) != B.charAt(index2) || A.charAt(index2) != B.charAt(index1)){

return false;

}

}else{

return false;

}

}

}

if (index1 == -1 && index2 == -1){ //deal corner case when A == B

Set<Character> set = new HashSet<>();

for (char c : A.toCharArray()){

if (!set.add(c)){

return true;

}

}

return false;

}else if (index2 == -1){

return false;

}

return true;

}

}

class Solution {

public boolean buddyStrings(String A, String B) {

int m = A.length();

int n = B.length();

if (m != n){

return false;

}

int index1 = -1;

int index2 = -1;

for (int i = 0; i < n; i++){

if (A.charAt(i) != B.charAt(i)){

if (index1 != -1 && index2 != -1){

return false;

}

if (index1 == -1){

index1 = i;

}else if (index2 == -1){

index2 = i;

if (A.charAt(index1) != B.charAt(index2) || A.charAt(index2) != B.charAt(index1)){

return false;

}

}

}

}

if (index1 == -1 && index2 == -1){

Set<Character> set = new HashSet<>();

for (char c : A.toCharArray()){

if (!set.add(c)){

return true;

}

}

return false;

}

return true;

}

}

## 86\_BSTIterator.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Design an iterator over a binary search tree with the following rules:

Elements are visited in ascending order (i.e. an in-order traversal)

next() and hasNext() queries run in O(1) time in average.

Have you met this question in a real interview?

Example

For the following binary search tree, in-order traversal by using iterator is [1, 6, 10, 11, 12]

10

/ \

1 11

\ \

6 12

Challenge

Extra memory usage O(h), h is the height of the tree.

Super Star: Extra memory usage O(1)

Method: Use stack to store infomation

/\*\*

\* Definition of TreeNode:

\* public class TreeNode {

\* public int val;

\* public TreeNode left, right;

\* public TreeNode(int val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\* }

\* Example of iterate a tree:

\* BSTIterator iterator = new BSTIterator(root);

\* while (iterator.hasNext()) {

\* TreeNode node = iterator.next();

\* do something for node

\* }

\*/

public class BSTIterator {

Stack<TreeNode> stack;

TreeNode next; //next treeNode to visit

/\*

\* @param root: The root of binary tree.

\*/public BSTIterator(TreeNode root) {

// do intialization if necessary

stack = new Stack<TreeNode>();

next = root;

addToStack(root);

}

/\*

\* @return: True if there has next node, or false

\*/

public boolean hasNext() {

return !stack.isEmpty();

}

/\*

\* @return: return next node

\*/

public TreeNode next() {

if (!hasNext()){

return null;

}

TreeNode cur = stack.pop();

next = cur.right;

if (next != null){

addToStack(next);

}

return cur;

}

private void addToStack(TreeNode node){

while (node != null){

stack.push(node);

node = node.left;

}

next = null;

}

}

Better version:

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class BSTIterator {

Stack<TreeNode> stack;

public BSTIterator(TreeNode root) {

stack = new Stack<>();

TreeNode node = root;

while (node != null){

stack.push(node);

node = node.left;

}

}

/\*\* @return whether we have a next smallest number \*/

public boolean hasNext() {

return !stack.isEmpty();

}

/\*\* @return the next smallest number \*/

public int next() {

TreeNode curr = stack.pop();

TreeNode node = curr.right;

while (node != null){

stack.push(node);

node = node.left;

}

return curr.val;

}

}

/\*\*

\* Your BSTIterator will be called like this:

\* BSTIterator i = new BSTIterator(root);

\* while (i.hasNext()) v[f()] = i.next();

\*/

Inorder traversal template

https://github.com/optimisea/Leetcode/blob/master/Java/94\_BinaryTreeInorderTraversal.java

class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<>();

Stack<TreeNode> stack = new Stack<>();

TreeNode node = root;

while (node != null || !stack.isEmpty()){

while (node != null){

stack.push(node);

node = node.left;

}

TreeNode curr = stack.pop();

res.add(curr.val);

node = curr.right;

}

return res;

}

}

## 860\_Lemonade.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

At a lemonade stand, each lemonade costs $5.

Customers are standing in a queue to buy from you, and order one at a time (in the order specified by bills).

Each customer will only buy one lemonade and pay with either a $5, $10, or $20 bill. You must provide the correct change to each customer, so that the net transaction is that the customer pays $5.

Note that you don't have any change in hand at first.

Return true if and only if you can provide every customer with correct change.

Example 1:

Input: [5,5,5,10,20]

Output: true

Explanation:

From the first 3 customers, we collect three $5 bills in order.

From the fourth customer, we collect a $10 bill and give back a $5.

From the fifth customer, we give a $10 bill and a $5 bill.

Since all customers got correct change, we output true.

Example 2:

Input: [5,5,10]

Output: true

Example 3:

Input: [10,10]

Output: false

Example 4:

Input: [5,5,10,10,20]

Output: false

Explanation:

From the first two customers in order, we collect two $5 bills.

For the next two customers in order, we collect a $10 bill and give back a $5 bill.

For the last customer, we can't give change of $15 back because we only have two $10 bills.

Since not every customer received correct change, the answer is false.

class Solution {

public boolean lemonadeChange(int[] bills) {

int five = 0;

int ten = 0;

for (int bill : bills){

if (bill == 5){

five++;

}else if (bill == 10){

if (five < 1){

return false;

}else{

five--;

ten++;

}

}else if (bill == 20){

if (five > 0 && ten > 0){

five--;

ten--;

}else if (five >= 3){

five -= 3;

}else{

return false;

}

}

}

return true;

}

}

class Solution {

public boolean lemonadeChange(int[] bills) {

Map<Integer, Integer> map = new HashMap<>();

for (int bill : bills){

if (bill == 5){

map.put(5, map.getOrDefault(5, 0) + 1);

}else if (bill == 10){

if (map.getOrDefault(5, 0) == 0){

return false;

}

map.put(5, map.get(5) - 1);

map.put(10, map.getOrDefault(10, 0) + 1);

}else{

if (map.getOrDefault(5, 0) == 0){

return false;

}

if (map.getOrDefault(10, 0) == 0){

if (map.get(5) < 3){

return false;

}

map.put(5, map.get(5) - 3);

}else{

map.put(10, map.get(10) - 1);

map.put(5, map.get(5) - 1);

}

}

}

return true;

}

}

## 861\_Score.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have a two dimensional matrix A where each value is 0 or 1.

A move consists of choosing any row or column, and toggling each value in that row or column: changing all 0s to 1s, and all 1s to 0s.

After making any number of moves, every row of this matrix is interpreted as a binary number, and the score of the matrix is the sum of these numbers.

Return the highest possible score.

Example 1:

Input: [[0,0,1,1],[1,0,1,0],[1,1,0,0]]

Output: 39

Explanation:

Toggled to [[1,1,1,1],[1,0,0,1],[1,1,1,1]].

0b1111 + 0b1001 + 0b1111 = 15 + 9 + 15 = 39

Note:

1 <= A.length <= 20

1 <= A[0].length <= 20

A[i][j] is 0 or 1.

Greedy: Notice that a 1 in the iith column from the right, contributes 2^i2

​i

​​ to the score.

Since 2^n > 2^{n-1} + 2^{n-2} + \cdots + 2^02

​n

​​ >2

​n−1

​​ +2

​n−2

​​ +⋯+2

​0

​​ , maximizing the left-most digit is more important than any other digit.

Thus, the rows should be toggled such that the left-most column is either all 0 or all 1 (so that after toggling the left-most column [if necessary], the left column is all 1.)

Algorithm

If we toggle rows by the first column (A[r][c] ^= A[r][0]), then the first column will be all 0.

Afterwards, the base score is max(col, R - col) where col is the column sum; and (1 << (C-1-c)) is the power of 2 that each 1 in that column contributes to the score.

class Solution {

public int matrixScore(int[][] A) {

int m = A.length;

int n = A[0].length;

int res = 0;

for (int j = 0; j < n; j++){

int zeros = 0;

for (int i = 0; i < m; i++){

zeros += A[i][j] ^ A[i][0];

}

res += Math.max(zeros, m - zeros) \* (1 << (n-1-j));

}

return res;

}

}

O(m\*n)

class Solution {

public int matrixScore(int[][] A) {

int m = A.length;

int n = A[0].length;

for (int i = 0; i < m; i++){

if (A[i][0] == 0){

for (int j = 0; j < n; j++){

A[i][j] ^= 1;

}

}

}

for (int j = 1; j < n; j++){

int count = 0;

for (int i = 0; i < m; i++){

if (A[i][j] == 0){

count++;

}

}

if (count > m - count){

for (int i = 0; i < m; i++){

A[i][j] ^= 1;

}

}

}

int res = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (A[i][j] == 1){

res += (1 << n-1-j);

}

}

}

return res;

}

}

## 862\_Shortest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Return the length of the shortest, non-empty, contiguous subarray of A with sum at least K.

If there is no non-empty subarray with sum at least K, return -1.

Example 1:

Input: A = [1], K = 1

Output: 1

Example 2:

Input: A = [1,2], K = 4

Output: -1

Example 3:

Input: A = [2,-1,2], K = 3

Output: 3

Note:

1 <= A.length <= 50000

-10 ^ 5 <= A[i] <= 10 ^ 5

1 <= K <= 10 ^ 9

Different from Leetcode 406 which is only positive integer

https://github.com/optimisea/Leetcode/blob/master/Java/406\_MinimumSizeSubarraySum.java

Method: Monotonic Stack/Queue

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public int shortestSubarray(int[] A, int K) {

int n = A.length;

//build prefix Sum array

int[] preSum = new int[n+1];

for (int i = 0; i < n; i++){

preSum[i+1] = preSum[i] + A[i];

}

//monotonic increasing stack/queue

Deque<Integer> deque = new LinkedList<>(); //store index, not prefix sum value

int min = n + 1;

int i = 0;

while (i < n+1){

//build monotonic stack

while (!deque.isEmpty() && preSum[deque.peekLast()] >= preSum[i]){

deque.pollLast();

}

//

while (!deque.isEmpty() && preSum[i] - preSum[deque.peekFirst()] >= K){

min = Math.min(min, i - deque.peekFirst());

deque.pollFirst();

}

deque.offerLast(i);

i++;

}

return min > n ? -1 : min;

}

}

https://leetcode.com/problems/shortest-subarray-with-sum-at-least-k/solution/

We can rephrase this as a problem about the prefix sums of A. Let P[i] = A[0] + A[1] + ... + A[i-1].

We want the smallest y-x such that y > x and P[y] - P[x] >= K.

Motivated by that equation, let opt(y) be the largest x such that P[x] <= P[y] - K. We need two key observations:

Key observation 1:

If x1 < x2 and P[x2] <= P[x1], then opt(y) can never be x1, as if P[x1] <= P[y] - K, then P[x2] <= P[x1] <= P[y] - K but y - x2

is smaller. This implies that our candidates x for opt(y) will have increasing values of P[x].

Key observation 2:

If opt(y1) = x, then we do not need to consider this x again. For if we find some y2 > y1 with opt(y2) = x,

then it represents an answer of y2 - x which is worse (larger) than y1 - x.

Algorithm

Maintain a "monoqueue" of indices of P: a deque of indices x\_0, x\_1, ... such that P[x\_0], P[x\_1], ... is increasing.

When adding a new index y, we'll pop x\_i from the end of the deque so that P[x\_0], P[x\_1], ..., P[y] will be increasing.

If P[y] >= P[x\_0] + K, then (as previously described), we don't need to consider this x\_0 again, and we can pop it from the

front of the deque.

## 863\_All.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given a binary tree (with root node root), a target node, and an integer value K.

Return a list of the values of all nodes that have a distance K from the target node. The answer can be returned in any order.

Example 1:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

Output: [7,4,1]

Explanation:

The nodes that are a distance 2 from the target node (with value 5)

have values 7, 4, and 1.

Note:

The given tree is non-empty.

Each node in the tree has unique values 0 <= node.val <= 500.

The target node is a node in the tree.

0 <= K <= 1000.

Method 1: convert to Graph

Time complexity: O(N)

Space complexity: O(N)

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> distanceK(TreeNode root, TreeNode target, int K) {

Map<Integer, List<Integer>> graph = new HashMap<>();

buildGraph(graph, root, null);

List<Integer> res = new ArrayList<>();

bfs(res, target, K, graph);

return res;

}

private void buildGraph(Map<Integer, List<Integer>> graph, TreeNode child, TreeNode parent){

if (!graph.containsKey(child.val)){

graph.put(child.val, new ArrayList<>());

}

List<Integer> list = graph.get(child.val);

if (parent != null){

list.add(parent.val);

}

if (child.left != null){

list.add(child.left.val);

buildGraph(graph, child.left, child);

}

if (child.right != null){

list.add(child.right.val);

buildGraph(graph, child.right, child);

}

}

private void bfs(List<Integer> res, TreeNode target, int K, Map<Integer, List<Integer>> graph){

Queue<Integer> queue = new LinkedList<>();

Set<Integer> seen = new HashSet<>();

queue.offer(target.val);

seen.add(target.val);

int level = 0;

while (!queue.isEmpty() && level <= K){

int size = queue.size();

for (int i = 0; i < size; i++){

int value = queue.poll();

if (level == K){

res.add(value);

}

for (int val : graph.get(value)){

if (!seen.contains(val)){

seen.add(val);

queue.add(val);

}

}

}

level++;

}

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> distanceK(TreeNode root, TreeNode target, int K) {

Map<TreeNode, List<TreeNode>> graph = new HashMap<>();

buildGraph(graph, root, null);

List<Integer> res = new ArrayList<>();

bfs(res, target, K, graph);

return res;

}

private void buildGraph(Map<TreeNode, List<TreeNode>> graph, TreeNode child, TreeNode parent){

if (child == null){

return;

}

graph.put(child, new ArrayList<>());

if (parent != null){

graph.get(child).add(parent);

graph.get(parent).add(child);

}

buildGraph(graph, child.left, child);

buildGraph(graph, child.right, child);

}

private void bfs(List<Integer> res, TreeNode target, int K, Map<TreeNode, List<TreeNode>> graph){

Queue<TreeNode> queue = new LinkedList<>();

Set<TreeNode> seen = new HashSet<>();

queue.offer(target);

seen.add(target);

int level = 0;

while (!queue.isEmpty() && level <= K){

int size = queue.size();

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

if (level == K){

res.add(node.val);

}

for (TreeNode n : graph.get(node)){

if (!seen.contains(n)){

seen.add(n);

queue.add(n);

}

}

}

level++;

}

}

}

Best solution:

class Solution {

public List<Integer> distanceK(TreeNode root, TreeNode target, int K) {

List<Integer> res = new ArrayList<>();

Map<TreeNode, Set<TreeNode>> graph = new HashMap<>();

buildGraph(graph, root, null);

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(target);

Set<TreeNode> visited = new HashSet<>();

visited.add(target);

int k = 0;

while (!queue.isEmpty() && k <= K){

int size = queue.size();

for (int i = 0; i < size; i++){

TreeNode curr = queue.poll();

if (k == K){

res.add(curr.val);

}

for (TreeNode next : graph.get(curr)){

if (!visited.contains(next)){

queue.offer(next);

visited.add(next);

}

}

}

k++;

}

return res;

}

private void buildGraph(Map<TreeNode, Set<TreeNode>> graph, TreeNode root, TreeNode parent){

if (root == null){

return;

}

if (!graph.containsKey(root)){

graph.put(root, new HashSet<>());

}

if (parent != null && !graph.containsKey(parent)){

graph.put(parent, new HashSet<>());

}

if (parent != null){

graph.get(root).add(parent);

graph.get(parent).add(root);

}

buildGraph(graph, root.left, root);

buildGraph(graph, root.right, root);

}

}

https://leetcode.com/problems/all-nodes-distance-k-in-binary-tree/discuss/143752/JAVA-Graph-+-BFS

## 864\_Shortest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given a 2-dimensional grid. "." is an empty cell, "#" is a wall, "@" is the starting point, ("a", "b", ...) are keys,

and ("A", "B", ...) are locks.

We start at the starting point, and one move consists of walking one space in one of the 4 cardinal directions. We cannot walk

outside the grid, or walk into a wall. If we walk over a key, we pick it up. We can't walk over a lock unless we have the

corresponding key.

For some 1 <= K <= 6, there is exactly one lowercase and one uppercase letter of the first K letters of the English alphabet

in the grid. This means that there is exactly one key for each lock, and one lock for each key; and also that the letters

used to represent the keys and locks were chosen in the same order as the English alphabet.

Return the lowest number of moves to acquire all keys. If it's impossible, return -1.

Example 1:

Input: ["@.a.#","###.#","b.A.B"]

Output: 8

Example 2:

Input: ["@..aA","..B#.","....b"]

Output: 6

Note:

1 <= grid.length <= 30

1 <= grid[0].length <= 30

grid[i][j] contains only '.', '#', '@', 'a'-'f' and 'A'-'F'

The number of keys is in [1, 6]. Each key has a different letter and opens exactly one lock.

Method: BFS

Similar as 847. Shortest Path Visiting All Nodes

Use bit to represent the key

Use key + ":" + x + ":" + y to represent the visited point to avoid repeating route

class Solution {

public int shortestPathAllKeys(String[] grid) {

int m = grid.length;

int n = grid[0].length();

int count = 0;

Queue<int[]> queue = new LinkedList<>();

Set<String> set = new HashSet<>();

int lowcaseNum = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

char c = grid[i].charAt(j);

if (c == '@'){

int[] state = new int[]{0, i, j};

queue.offer(state);

set.add(state[0] + ":" + state[1] + ":" + state[2]);

}else if (Character.isLowerCase(c)){

lowcaseNum++;

}

}

}

int target = 0;

for (int i = 0; i < lowcaseNum; i++){

target |= (1 << i);

}

int[][] dirs = new int[][]{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

int[] state = queue.poll();

if (state[0] == target){

return count;

}

int key = state[0];

int cx = state[1];

int cy = state[2];

for (int[] dir : dirs){

int nx = cx + dir[0];

int ny = cy + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n){

char c = grid[nx].charAt(ny);

if (Character.isLowerCase(c)){//get the key

int newKey = key | (1 << ((int)(c -'a')));

String newStr = newKey + ":" + nx + ":" + ny;

if (!set.contains(newStr)){

queue.offer(new int[]{newKey, nx, ny});

set.add(newStr);

}

}else if (Character.isUpperCase(c)){

if ((key & (1 << (int)(c - 'A'))) != 0){//already has the key for this lock

String newStr = key + ":" + nx + ":" + ny;

if (!set.contains(newStr)){

queue.offer(new int[]{key, nx, ny});

set.add(newStr);

}

}

}else if (c == '.' || c == '@'){

String newStr = key + ":" + nx + ":" + ny;

if (!set.contains(newStr)){

queue.offer(new int[]{key, nx, ny});

set.add(newStr);

}

}

}

}

}

count++;

}

return -1;

}

}

class Solution {

public int shortestPathAllKeys(String[] grid) {

int m = grid.length;

int n = grid[0].length();

int count = 0;

Queue<int[]> queue = new LinkedList<>();

Set<String> set = new HashSet<>();

int target = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

char c = grid[i].charAt(j);

if (c == '@'){

int[] state = new int[]{0, i, j};

queue.offer(state);

set.add(state[0] + ":" + state[1] + ":" + state[2]);

}else if (Character.isLowerCase(c)){

target |= 1 << (int)(c - 'a');

}

}

}

int[][] dirs = new int[][]{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

int[] state = queue.poll();

if (state[0] == target){

return count;

}

int key = state[0];

int cx = state[1];

int cy = state[2];

for (int[] dir : dirs){

int nx = cx + dir[0];

int ny = cy + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n){

char c = grid[nx].charAt(ny);

if (Character.isLowerCase(c)){//get the key

int newKey = key | (1 << ((int)(c -'a')));

String newStr = newKey + ":" + nx + ":" + ny;

if (!set.contains(newStr)){

queue.offer(new int[]{newKey, nx, ny});

set.add(newStr);

}

}else if (Character.isUpperCase(c) && (key & (1 << (int)(c - 'A'))) != 0 || c == '.' || c == '@'){

String newStr = key + ":" + nx + ":" + ny;

if (!set.contains(newStr)){

queue.offer(new int[]{key, nx, ny});

set.add(newStr);

}

}

}

}

}

count++;

}

return -1;

}

}

## 865\_Smallest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree rooted at root, the depth of each node is the shortest distance to the root.

A node is deepest if it has the largest depth possible among any node in the entire tree.

The subtree of a node is that node, plus the set of all descendants of that node.

Return the node with the largest depth such that it contains all the deepest nodes in its subtree.

Example 1:

Input: [3,5,1,6,2,0,8,null,null,7,4]

Output: [2,7,4]

Explanation:

We return the node with value 2, colored in yellow in the diagram.

The nodes colored in blue are the deepest nodes of the tree.

The input "[3, 5, 1, 6, 2, 0, 8, null, null, 7, 4]" is a serialization of the given tree.

The output "[2, 7, 4]" is a serialization of the subtree rooted at the node with value 2.

Both the input and output have TreeNode type.

Note:

The number of nodes in the tree will be between 1 and 500.

The values of each node are unique.

Method 1: preOrder + postOrder

Two pass:

Time complexity: O(N)

Space complexity: O(N)

preOrder == top down

postOrder == bottom up

class Solution {

public TreeNode subtreeWithAllDeepest(TreeNode root) {

Map<TreeNode, Integer> map = new HashMap<>();

map.put(null, -1);

preOrder(root, null, map);

int max = 0;

for (int depth : map.values()){

max = Math.max(max, depth);

}

return postOrder(root, max, map);

}

//label node with depth, increase order so use preOrder

private void preOrder(TreeNode root, TreeNode parent, Map<TreeNode, Integer> map){

if (root == null){

return;

}

map.put(root, map.get(parent) + 1);

preOrder(root.left, root, map);

preOrder(root.right, root, map);

}

//collect the deepest node

private TreeNode postOrder(TreeNode root, int max, Map<TreeNode, Integer> map){

if (root == null || map.get(root) == max){

return root;

}

TreeNode left = postOrder(root.left, max, map);

TreeNode right = postOrder(root.right, max, map);

if (left != null && right != null){

return root;

}

if (left != null){

return left;

}

if (right != null){

return right;

}

return null;

}

}

O(N)

class Solution {

public TreeNode subtreeWithAllDeepest(TreeNode root) {

Map<TreeNode, Integer> map = new HashMap<>();

map.put(null, 0);

dfs(root, null, map);

int max = 0;

for (int d : map.values()){

max = Math.max(d, max);

}

return lowestCommonDepth(root, max, map);

}

private void dfs(TreeNode root, TreeNode parent, Map<TreeNode, Integer> map){

if (root == null){

return;

}

map.put(root, map.get(parent) + 1);

dfs(root.left, root, map);

dfs(root.right, root, map);

}

private TreeNode lowestCommonDepth(TreeNode root, int max, Map<TreeNode, Integer> map){

if (root == null || map.get(root) == max){

return root;

}

TreeNode left = lowestCommonDepth(root.left, max, map);

TreeNode right = lowestCommonDepth(root.right, max, map);

if (left != null && right != null){

return root;

}

if (left != null && right == null){

return left;

}

if (left == null && right != null){

return right;

}

return null;

}

}

Method 2:

Time complexity: O(N^2)

class Solution {

public TreeNode subtreeWithAllDeepest(TreeNode root) {

if (root == null){

return root;

}

int leftDepth = getDepth(root.left);

int rightDepth = getDepth(root.right);

if (leftDepth == rightDepth){

return root;

}

if (leftDepth > rightDepth){

return subtreeWithAllDeepest(root.left);

}

return subtreeWithAllDeepest(root.right);

}

private int getDepth(TreeNode root){

if (root == null){

return 0;

}

return 1 + Math.max(getDepth(root.left), getDepth(root.right));

}

}

## 867\_Transpose.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a matrix A, return the transpose of A.

The transpose of a matrix is the matrix flipped over it's main diagonal, switching the row and column indices of the matrix.

Example 1:

Input: [[1,2,3],[4,5,6],[7,8,9]]

Output: [[1,4,7],[2,5,8],[3,6,9]]

Example 2:

Input: [[1,2,3],[4,5,6]]

Output: [[1,4],[2,5],[3,6]]

Note:

1 <= A.length <= 1000

1 <= A[0].length <= 1000

class Solution {

public int[][] transpose(int[][] A) {

int m = A.length;

int n = A[0].length;

int[][] res = new int[n][m];

for (int i = 0; i < n; i++){

for (int j = 0; j < m; j++){

res[i][j] = A[j][i];

}

}

return res;

}

}

868\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a positive integer N, find and return the longest distance between two consecutive 1's in the binary representation of N.

If there aren't two consecutive 1's, return 0.

Example 1:

Input: 22

Output: 2

Explanation:

22 in binary is 0b10110.

In the binary representation of 22, there are three ones, and two consecutive pairs of 1's.

The first consecutive pair of 1's have distance 2.

The second consecutive pair of 1's have distance 1.

The answer is the largest of these two distances, which is 2.

Example 2:

Input: 5

Output: 2

Explanation:

5 in binary is 0b101.

Example 3:

Input: 6

Output: 1

Explanation:

6 in binary is 0b110.

Example 4:

Input: 8

Output: 0

Explanation:

8 in binary is 0b1000.

There aren't any consecutive pairs of 1's in the binary representation of 8, so we return 0.

Note:

1 <= N <= 10^9

Method 1:

Time Complexity: O(logN). Note that logN is the number of digits in the binary representation of N.

Space Complexity: O(logN), the space used by A.

class Solution {

public int binaryGap(int N) {

int max = 0;

List<Integer> list = new ArrayList<>();

int index = 0;

while (N > 0){

if (N % 2 == 1){

list.add(index);

}

N /= 2;

index++;

}

for (int i = 1; i < list.size(); i++){

max = Math.max(max, list.get(i) - list.get(i-1));

}

return max;

}

}

Method 2:

Since we only care about consecutive values of this array A, we don't need to store the whole array.

We only need to remember the last value seen.

Time complexity: O(logN)

Space complexity: O(1)

class Solution {

public int binaryGap(int N) {

int max = 0;

int index = 0;

int lastIndex = -1;

while (N > 0){

if (N % 2 == 1){

if (lastIndex >= 0){

max = Math.max(max, index - lastIndex);

}

lastIndex = index;

}

N /= 2;

index++;

}

return max;

}

}

Best solution:

Time complexity: O(1)

Space complexity: O(1)

class Solution {

public int binaryGap(int N) {

int prev = -1;

int curr = -1;

int max = 0;

for (int i = 0; i < 32; i++){

if (((N >> i) & 1) == 1){

curr = i;

if (curr != - 1 && prev != -1){

max = Math.max(max, curr - prev);

}

prev = curr;

}

}

return max;

}

}

## 869\_Reordered.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Starting with a positive integer N, we reorder the digits in any order (including the original order) such that the leading digit is not zero.

Return true if and only if we can do this in a way such that the resulting number is a power of 2.

Example 1:

Input: 1

Output: true

Example 2:

Input: 10

Output: false

Example 3:

Input: 16

Output: true

Example 4:

Input: 24

Output: false

Example 5:

Input: 46

Output: true

Note:

1 <= N <= 10^9

Method 1: Brute Force

Time Complexity: O((logN)!∗logN). Note that logN is the number of digits in the binary representation of N.

For each of (logN)! permutations of the digits of N, we need to check that it is a power of 2 in O(logN) time.

Space Complexity: O(logN), the space used by A

class Solution {

public boolean reorderedPowerOf2(int N) {

String s = Integer.toString(N);

int[] A = new int[s.length()];

for (int i = 0; i < A.length; i++){

A[i] = s.charAt(i) - '0';

}

return permutation(A, 0);

}

private boolean permutation(int[] A, int start){

if (start == A.length){

return isPowerOfTwo(A);

}

for (int i = start; i < A.length; i++){

swap(A, start, i);

if (permutation(A, start+1)){

return true;

}

swap(A, start, i);

}

return false;

}

private void swap(int[] A, int i, int j){

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

private boolean isPowerOfTwo(int[] A){

if (A[0] == 0){

return false;

}

int num = 0;

for (int i = 0; i < A.length; i++){

num = 10 \* num + A[i];

}

int count = 0;

while (num > 0){

if ((num & 1) == 1){

count++;

}

num >>= 1;

}

return count == 1;

}

}

Method 2:

Note that the range of integer is between -2^31 and 2^31, which is

Time Complexity: O(logn \* logn) There are logN different candidate powers of 2, and each comparison hasO(logN) time complexity.

Space Complexity: O(logN).

class Solution {

public boolean reorderedPowerOf2(int N) {

int[] hash = count(N);

for (int i = 0; i < 31; i++){

int[] arr = count((1 << i));

if (Arrays.equals(arr, hash)){

return true;

}

}

return false;

}

private int[] count(int N){

int[] res = new int[10];

while (N > 0){

res[N % 10]++;

N /= 10;

}

return res;

}

}

class Solution {

public boolean reorderedPowerOf2(int N) {

int[] counter = count(N);

for (int i = 0; i < 31; i++){

int[] cand = count(1<< i);

boolean found = true;

for (int j = 0; j < 10; j++){

if (cand[j] != counter[j]){

found = false;

break;

}

}

if (found){

return true;

}

}

return false;

}

private int[] count(int N){

int[] res = new int[10];

while (N > 0){

res[N%10]++;

N /= 10;

}

return res;

}

}

## 87\_ScrambleString.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string s1, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of s1 = "great":

great

/ \

gr eat

/ \ / \

g r e at

/ \

a t

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

rgeat

/ \

rg eat

/ \ / \

r g e at

/ \

a t

We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

rgtae

/ \

rg tae

/ \ / \

r g ta e

/ \

t a

We say that "rgtae" is a scrambled string of "great".

Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.

Method 1: Recursion Time complexity: O(2^n)

class Solution {

public boolean isScramble(String s1, String s2) {

if (s1.equals(s2)){

return true;

}

int[] hash = new int[26];

for (int i = 0; i < s1.length(); i++){

hash[s1.charAt(i) - 'a']++;

hash[s2.charAt(i) - 'a']--;

}

for (int i = 0; i < 26; i++){

if (hash[i] != 0){

return false;

}

}

for (int i = 1; i < s1.length(); i++){

if (isScramble(s1.substring(0, i), s2.substring(0, i)) //non-swap case

&& isScramble(s1.substring(i), s2.substring(i))){

return true;

}

if (isScramble(s1.substring(0, i), s2.substring(s2.length() - i)) // swap case

&& isScramble(s1.substring(i), s2.substring(0, s2.length() - i))){

return true;

}

}

return false;

}

}

Method 2: dp time complexity: O(n^4)

https://leetcode.com/problems/scramble-string/discuss/29396/Simple-iterative-DP-Java-solution-with-explanation

## 870\_Advantage.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two arrays A and B of equal size, the advantage of A with respect to B is the number of indices i for which A[i] > B[i].

Return any permutation of A that maximizes its advantage with respect to B.

Example 1:

Input: A = [2,7,11,15], B = [1,10,4,11]

Output: [2,11,7,15]

Example 2:

Input: A = [12,24,8,32], B = [13,25,32,11]

Output: [24,32,8,12]

Note:

1 <= A.length = B.length <= 10000

0 <= A[i] <= 10^9

0 <= B[i] <= 10^9

Note that Time complexity: treemap remove operation: O(logn)

priority queue remove operation: O(n)

Method 1: TreeMap Best solution

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public int[] advantageCount(int[] A, int[] B) {

int[] res = new int[A.length];

TreeMap<Integer, Integer> map = new TreeMap<>();

for (int i : A){

map.put(i, map.getOrDefault(i, 0) + 1);

}

for (int i = 0; i < B.length; i++){

Integer key = map.higherKey(B[i]);

if (key == null){

key = map.firstKey();

}

res[i] = key;

map.put(key, map.get(key) - 1);

if (map.get(key) == 0){

map.remove(key);

}

}

return res;

}

}

Method 2: PQ

Time complexity: O(nlogn)

Space complexity: O(n)

class Solution {

public int[] advantageCount(int[] A, int[] B) {

int n = A.length;

int[] res = new int[n];

Arrays.sort(A);

Queue<int[]> maxQ = new PriorityQueue<int[]>(new Comparator<int[]>(){

public int compare (int[] p1, int[] p2){

return p2[0] - p1[0];

}

});

for (int i = 0; i < n; i++){

maxQ.offer(new int[]{B[i], i});

}

int low = 0;

int high = n - 1;

while (!maxQ.isEmpty()){

int[] p = maxQ.poll();

int val = p[0];

int index = p[1];

if (A[high] > val){

res[index] = A[high];

high--;

}else{

res[index] = A[low];

low++;

}

}

return res;

}

}

## 871\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A car travels from a starting position to a destination which is target miles east of the starting position.

Along the way, there are gas stations. Each station[i] represents a gas station that is station[i][0] miles east of the starting

position, and has station[i][1] liters of gas.

The car starts with an infinite tank of gas, which initially has startFuel liters of fuel in it. It uses 1 liter of gas per 1 mile

that it drives.

When the car reaches a gas station, it may stop and refuel, transferring all the gas from the station into the car.

What is the least number of refueling stops the car must make in order to reach its destination? If it cannot reach the destination,

return -1.

Note that if the car reaches a gas station with 0 fuel left, the car can still refuel there. If the car reaches the destination with

0 fuel left, it is still considered to have arrived.

Example 1:

Input: target = 1, startFuel = 1, stations = []

Output: 0

Explanation: We can reach the target without refueling.

Example 2:

Input: target = 100, startFuel = 1, stations = [[10,100]]

Output: -1

Explanation: We can't reach the target (or even the first gas station).

Example 3:

Input: target = 100, startFuel = 10, stations = [[10,60],[20,30],[30,30],[60,40]]

Output: 2

Explanation:

We start with 10 liters of fuel.

We drive to position 10, expending 10 liters of fuel. We refuel from 0 liters to 60 liters of gas.

Then, we drive from position 10 to position 60 (expending 50 liters of fuel),

and refuel from 10 liters to 50 liters of gas. We then drive to and reach the target.

We made 2 refueling stops along the way, so we return 2.

Note:

1 <= target, startFuel, stations[i][1] <= 10^9

0 <= stations.length <= 500

0 < stations[0][0] < stations[1][0] < ... < stations[stations.length-1][0] < target

Method 1: DP

Time complexity:O(N^2)

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

int N = stations.length;

int[] dp = new int[N+1]; //dp[t] means the furthest distance that we can get with t times of refueling.

dp[0] = startFuel;

for (int i = 0; i < N; i++){

for (int j = i; j >= 0; j--){

if (dp[j] >= stations[i][0]){

dp[j+1] = Math.max(dp[j+1], dp[j] + stations[i][1]);

}

}

}

for (int i = 0; i <= N; i++){

if (dp[i] >= target){

return i;

}

}

return -1;

}

}

Method 2: Better solution

PriorityQueue, always try to add gas at the largest gas station first

https://leetcode.com/problems/minimum-number-of-refueling-stops/solution/

We initial res = 0 and in every loop:

We add all reachable stop to priority queue.

We pop out the largest gas from pq and refeul once.

If we can't refuel, means that we can not go forward and return -1

Time complexity: O(NlogN)

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

Queue<Integer> maxPQ = new PriorityQueue<>(new Comparator<Integer>(){

public int compare (Integer i1, Integer i2){

return i2 - i1;

}

});

int res = 0;

int N = stations.length;

int curr = startFuel;

int i = 0;

while (curr < target){

while (i < N && curr >= stations[i][0]){

maxPQ.offer(stations[i][1]);

i++;

}

if (maxPQ.isEmpty()){

return -1;

}

curr += maxPQ.poll();

res++;

}

return res;

}

}

## 872\_Leaf-Similar.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Consider all the leaves of a binary tree. From left to right order, the values of those leaves form a leaf value sequence.

For example, in the given tree above, the leaf value sequence is (6, 7, 4, 9, 8).

Two binary trees are considered leaf-similar if their leaf value sequence is the same.

Return true if and only if the two given trees with head nodes root1 and root2 are leaf-similar.

Note:

Both of the given trees will have between 1 and 100 nodes.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean leafSimilar(TreeNode root1, TreeNode root2) {

List<Integer> list1 = getLeaf(root1);

List<Integer> list2 = getLeaf(root2);

if (list1.size() != list2.size()){

return false;

}

for (int i = 0; i < list1.size(); i++){

if (list1.get(i) != list2.get(i)){

return false;

}

}

return true;

}

private List<Integer> getLeaf(TreeNode root){

List<Integer> res = new ArrayList<>();

preOrder(root, res);

return res;

}

private void preOrder(TreeNode root, List<Integer> res){

if (root == null){

return;

}

if (root.left == null && root.right == null){

res.add(root.val);

}

preOrder(root.left, res);

preOrder(root.right, res);

}

}

## 873\_Length.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A sequence X\_1, X\_2, ..., X\_n is fibonacci-like if:

n >= 3

X\_i + X\_{i+1} = X\_{i+2} for all i + 2 <= n

Given a strictly increasing array A of positive integers forming a sequence, find the length of the longest fibonacci-like subsequence of A. If one does not exist, return 0.

(Recall that a subsequence is derived from another sequence A by deleting any number of elements (including none) from A, without changing the order of the remaining elements. For example, [3, 5, 8] is a subsequence of [3, 4, 5, 6, 7, 8].)

Example 1:

Input: [1,2,3,4,5,6,7,8]

Output: 5

Explanation:

The longest subsequence that is fibonacci-like: [1,2,3,5,8].

Example 2:

Input: [1,3,7,11,12,14,18]

Output: 3

Explanation:

The longest subsequence that is fibonacci-like:

[1,11,12], [3,11,14] or [7,11,18].

Note:

3 <= A.length <= 1000

1 <= A[0] < A[1] < ... < A[A.length - 1] <= 10^9

(The time limit has been reduced by 50% for submissions in Java, C, and C++.)

very similar as Longest Increase subsequence, the only difference is that use pair (i, j) and (j, k) instead of single number to show

(i,j) and (j, k) are connected, also hash (i,j) to one number in hashmap

Method: dp

Time complexity: O(N^2)

class Solution {

public int lenLongestFibSubseq(int[] A) {

int N = A.length;

Map<Integer, Integer> index = new HashMap<>();

for (int i = 0; i < N; i++){

index.put(A[i], i);

}

Map<Integer, Integer> dp = new HashMap<>();

int res = 0;

for (int k = 2; k < N; k++){

for (int j = 0; j < k; j++){

int i = index.getOrDefault(A[k] - A[j], -1); // show that (i, j) and (j, k) are connected

if (i >= 0 && i < j){

int keyJ = i \* N + j; //hash pair

int keyK = j \* N + k;

int valJ = dp.getOrDefault(keyJ, 2);

int valK = dp.getOrDefault(keyK, 3);

if (valJ + 1 >= valK){

dp.put(keyK, valJ + 1);

res = Math.max(res, valJ + 1);

}

}

}

}

return res >= 3 ? res : 0;

}

}

class Solution {

public int lenLongestFibSubseq(int[] A) {

int N = A.length;

Map<Integer, Integer> index = new HashMap<>();

int[][] dp = new int[N][N];

int res = 0;

for (int k = 0; k < N; k++){

index.put(A[k], k);

for (int j = 0; j < k; j++){

int i = index.getOrDefault(A[k] - A[j], -1);

if (i >= 0 && i < j){

dp[j][k] = Math.max(dp[i][j] + 1, 3);

res = Math.max(res, dp[j][k]);

}

}

}

return res >= 3 ? res : 0;

}

}

Best:

dp[i][j] represents the length of longest sequence which ends with A[i] and A[j].

class Solution {

public int lenLongestFibSubseq(int[] A) {

int N = A.length;

int[][] dp = new int[N][N];

Map<Integer, Integer> pos = new HashMap<>(); // to reduce one dimension loop

for (int i = 0; i < N; i++){

Arrays.fill(dp[i], 2);

pos.put(A[i], i);

}

int max = 2;

for (int j = 0; j < N; j++){

for (int i = 0; i < j; i++){

int first = A[j] - A[i];

if (first >= A[i]){

continue;

}

if (!pos.containsKey(first)){

continue;

}

int index = pos.get(first);

dp[i][j] = dp[index][i] + 1;

max = Math.max(max, dp[i][j]);

}

}

return max == 2 ? 0 : max;

}

}

## 874\_WalkingRobot.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A robot on an infinite grid starts at point (0, 0) and faces north. The robot can receive one of three possible types of commands:

-2: turn left 90 degrees

-1: turn right 90 degrees

1 <= x <= 9: move forward x units

Some of the grid squares are obstacles.

The i-th obstacle is at grid point (obstacles[i][0], obstacles[i][1])

If the robot would try to move onto them, the robot stays on the previous grid square instead (but still continues following the rest of the route.)

Return the square of the maximum Euclidean distance that the robot will be from the origin.

Example 1:

Input: commands = [4,-1,3], obstacles = []

Output: 25

Explanation: robot will go to (3, 4)

Example 2:

Input: commands = [4,-1,4,-2,4], obstacles = [[2,4]]

Output: 65

Explanation: robot will be stuck at (1, 4) before turning left and going to (1, 8)

Note:

0 <= commands.length <= 10000

0 <= obstacles.length <= 10000

-30000 <= obstacle[i][0] <= 30000

-30000 <= obstacle[i][1] <= 30000

The answer is guaranteed to be less than 2 ^ 31.

Method 1:

class Solution {

public int robotSim(int[] commands, int[][] obstacles) {

int x = 0;

int y = 0;

Set<String> set = new HashSet<>();

for (int[] obstacle : obstacles){

set.add(obstacle[0] + ":" + obstacle[1]);

}

boolean yMove = true;

boolean movePositive = true; //move towards to positive direction

int res = 0;

for (int command : commands){

if (command < 0){

if (yMove && movePositive){

if (command == -2){

movePositive = !movePositive;

}

}else if (yMove && !movePositive){

if (command == -2){

movePositive = !movePositive;

}

}else if (!yMove && movePositive){

if (command == -1){

movePositive = !movePositive;

}

}else{

if (command == -1){

movePositive = !movePositive;

}

}

yMove = !yMove;

}else{

int step = 1;

if (yMove && movePositive){

while (step <= command && !set.contains(x + ":" + (y+1))){

y++;

step++;

}

}else if (yMove && !movePositive){

while (step <= command && !set.contains(x + ":" + (y-1))){

y--;

step++;

}

}else if (!yMove && movePositive){

while (step <= command && !set.contains((x+1) + ":" + y)){

x++;

step++;

}

}else{

while (step <= command && !set.contains((x-1) + ":" + y)){

x--;

step++;

}

}

}

res = Math.max(res, x \* x + y \* y);

}

return res;

}

}

Method 2: Better

class Solution {

public int robotSim(int[] commands, int[][] obstacles) {

int res = 0;

int x = 0;

int y = 0;

int[][] dirs = new int[][]{{0,1}, {1,0}, {0,-1}, {-1,0}};

Set<String> set = new HashSet<>();

for (int[] ob : obstacles){

set.add(ob[0] + ":" + ob[1]);

}

int d = 0;

int n = dirs.length;

for (int c : commands){

if (c == -1){

d = (d + 1) % n;

}else if (c == -2){

d = (d + 3) % n;//note that in Java (-1) % 4 = -1

}else{

while (c > 0 && !set.contains(x + dirs[d][0] + ":" + (y + dirs[d][1]))){

x += dirs[d][0];

y += dirs[d][1];

c--;

}

}

res = Math.max(res, x \* x + y \* y);

}

return res;

}

}

class Solution {

public int robotSim(int[] commands, int[][] obstacles) {

Set<String> set = new HashSet<>();

for (int[] o : obstacles){

set.add(o[0] + ":" + o[1]);

}

int x = 0;

int y = 0;

int[][] dirs = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

int max = 0;

int d = 0;

for (int c : commands){

if (c == -1){

d = (d + 1) % 4;

}else if (c == -2){

d = (d + 3) %4;

}else{

int[] dir = dirs[d];

int nx = x + dir[0];

int ny = y + dir[1];

String key = nx + ":" + ny;

while (c > 0 && !set.contains(key)){

x += dir[0];

y += dir[1];

c--;

nx = x + dir[0];

ny = y + dir[1];

key = nx + ":" + ny;

}

}

max = Math.max(max, x \* x + y \* y);

}

return max;

}

}

## 875\_Koko.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Koko loves to eat bananas. There are N piles of bananas, the i-th pile has piles[i] bananas. The guards have gone and will come

back in H hours.

Koko can decide her bananas-per-hour eating speed of K. Each hour, she chooses some pile of bananas, and eats K bananas

from that pile. If the pile has less than K bananas, she eats all of them instead, and won't eat any more bananas during this hour.

Koko likes to eat slowly, but still wants to finish eating all the bananas before the guards come back.

Return the minimum integer K such that she can eat all the bananas within H hours.

Example 1:

Input: piles = [3,6,7,11], H = 8

Output: 4

Example 2:

Input: piles = [30,11,23,4,20], H = 5

Output: 30

Example 3:

Input: piles = [30,11,23,4,20], H = 6

Output: 23

Note:

1 <= piles.length <= 10^4

piles.length <= H <= 10^9

1 <= piles[i] <= 10^9

Similar as Leetcode 1014. Capacity To Ship Packages Within D Days

https://github.com/optimisea/Leetcode/blob/master/Java/1014\_Capacity.java

Simiar as 410 Split Array Largest Sum

https://github.com/optimisea/Leetcode/blob/master/Java/410\_Split.java

Time complexity: O(NlogM)

class Solution {

public int minEatingSpeed(int[] piles, int H) {

int low = 1;

int high = (int)Math.pow(10, 9);

while (low <= high){

int mid = low + (high - low) / 2;

int hrs = hrsToFinish(piles, mid);

if (hrs <= H){

high = mid - 1;

}else{

low = mid + 1;

}

}

return low;

}

private int hrsToFinish(int[] piles, int speed){

int count = 0;

for (int pile : piles){

count += (int)Math.ceil((double) pile / speed);

}

return count;

}

}

class Solution {

public int minEatingSpeed(int[] piles, int H) {

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;

for (int i : piles){

min = Math.min(min, i);

max = Math.max(max, i);

}

int low = 0;

int high = max;

while (low <= high){

int mid = low + (high - low) / 2;

if (possible(piles, H, mid)){

high = mid - 1;

}else{

low = mid + 1;

}

}

return low;

}

private boolean possible(int[] piles, int H, int mid){

int sum = 0;

for (int i : piles){

sum += (int)Math.ceil( (double)i / mid);

}

return sum <= H;

}

}

## 876\_Middle.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty, singly linked list with head node head, return a middle node of linked list.

If there are two middle nodes, return the second middle node.

Example 1:

Input: [1,2,3,4,5]

Output: Node 3 from this list (Serialization: [3,4,5])

The returned node has value 3. (The judge's serialization of this node is [3,4,5]).

Note that we returned a ListNode object ans, such that:

ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, and ans.next.next.next = NULL.

Example 2:

Input: [1,2,3,4,5,6]

Output: Node 4 from this list (Serialization: [4,5,6])

Since the list has two middle nodes with values 3 and 4, we return the second one.

Note:

The number of nodes in the given list will be between 1 and 100.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode middleNode(ListNode head) {

if (head == null){

return head;

}

ListNode slow = head;

ListNode fast = head;

while (fast != null && fast.next != null){

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

}

## 877\_Stone.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Alex and Lee play a game with piles of stones. There are an even number of piles arranged in a row, and each pile has a positive

integer number of stones piles[i].

The objective of the game is to end with the most stones. The total number of stones is odd, so there are no ties.

Alex and Lee take turns, with Alex starting first. Each turn, a player takes the entire pile of stones from either the beginning

or the end of the row. This continues until there are no more piles left, at which point the person with the most stones wins.

Assuming Alex and Lee play optimally, return True if and only if Alex wins the game.

Example 1:

Input: [5,3,4,5]

Output: true

Explanation:

Alex starts first, and can only take the first 5 or the last 5.

Say he takes the first 5, so that the row becomes [3, 4, 5].

If Lee takes 3, then the board is [4, 5], and Alex takes 5 to win with 10 points.

If Lee takes the last 5, then the board is [3, 4], and Alex takes 4 to win with 9 points.

This demonstrated that taking the first 5 was a winning move for Alex, so we return true.

Note:

2 <= piles.length <= 500

piles.length is even.

1 <= piles[i] <= 500

sum(piles) is odd.

https://leetcode.com/problems/stone-game/discuss/154610/C++JavaPython-DP-or-Just-return-true?page=1

class Solution {

public boolean stoneGame(int[] piles) {

int n = piles.length;

int[][] dp = new int[n][n];

for (int i = 0; i < n; i++){

dp[i][i] = piles[i];

}

for (int i = 0; i < n - 1; i++){

for (int j = n -1; j > i; j--){

dp[i][j] = Math.max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1]);

}

}

return dp[0][n-1] > 0;

}

}

The same as Leetcode 486 Predict the winner

https://github.com/optimisea/Leetcode/blob/master/Java/486\_Predict.java

class Solution {

public boolean stoneGame(int[] piles) {

int N = piles.length;

int[][] cache = new int[N][N];

for (int i = 0; i < N; i++){

Arrays.fill(cache[i], -1);

}

return maxRelativeStone(piles, 0, N-1, cache) >= 0;

}

private int maxRelativeStone(int[] piles, int start, int end, int[][] cache){

if (cache[start][end] != -1){

return cache[start][end];

}

if (start > end){

return 0;

}

if (start == end){

return piles[start];

}

int chooseFront = piles[start] - maxRelativeStone(piles, start+1, end, cache);

int chooseEnd = piles[end] - maxRelativeStone(piles, start, end-1, cache);

cache[start][end] = Math.max(chooseFront, chooseEnd);

return cache[start][end];

}

}

## 878\_Nth.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A positive integer is magical if it is divisible by either A or B.

Return the N-th magical number. Since the answer may be very large, return it modulo 10^9 + 7.

Example 1:

Input: N = 1, A = 2, B = 3

Output: 2

Example 2:

Input: N = 4, A = 2, B = 3

Output: 6

Example 3:

Input: N = 5, A = 2, B = 4

Output: 10

Example 4:

Input: N = 3, A = 6, B = 4

Output: 8

Note:

1 <= N <= 10^9

2 <= A <= 40000

2 <= B <= 40000

Method 1: TLE

Time complexity: NlogN

class Solution {

public int nthMagicalNumber(int N, int A, int B) {

int modulo = (int) Math.pow(10, 9) + 7;

Queue<Long> pq = new PriorityQueue<>();

Set<Long> set = new HashSet<>();

for (int i = 1; i <= N; i++){

long a = i \* (long) A;

long b = i \* (long) B;

if (set.add(a)){

pq.offer(a);

}

if (set.add(b)){

pq.offer(b);

}

}

int count = 0;

while (count < N - 1){

pq.poll();

count++;

}

return (int) (pq.peek() % modulo);

}

}

Method 2: Binary Search

https://leetcode.com/problems/nth-magical-number/discuss/154613/C%2B%2BJavaPython-Binary-Search

Time complexity: O(logN)

class Solution {

public int nthMagicalNumber(int N, int A, int B) {

//first calculate gcd (greatest common divisor) and lcm (least common multiple)

long gcd = gcdLong(A, B);

long lcm = (A \* B) / gcd;

long low = Math.min(A, B);

long high = N \* low;

while (low <= high){

long mid = low + (high - low) / 2;

long numOfMagicNum = mid / A + mid / B - mid / lcm; // method to calculate number of magic number: Inclusion exclusion principle.

if (numOfMagicNum >= N){

high = mid - 1;

}else{

low = mid + 1;

}

}

long mod = (long)1e9 + 7;

return (int)(low % mod);

}

private long gcdLong (long a, long b){

if (b == 0){

return a;

}

return gcd(b, a%b);

}

}

## 879\_Profitable.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are G people in a gang, and a list of various crimes they could commit.

The i-th crime generates a profit[i] and requires group[i] gang members to participate.

If a gang member participates in one crime, that member can't participate in another crime.

Let's call a profitable scheme any subset of these crimes that generates at least P profit, and the total number of gang members participating in that subset of crimes is at most G.

How many schemes can be chosen? Since the answer may be very large, return it modulo 10^9 + 7.

Example 1:

Input: G = 5, P = 3, group = [2,2], profit = [2,3]

Output: 2

Explanation:

To make a profit of at least 3, the gang could either commit crimes 0 and 1, or just crime 1.

In total, there are 2 schemes.

Example 2:

Input: G = 10, P = 5, group = [2,3,5], profit = [6,7,8]

Output: 7

Explanation:

To make a profit of at least 5, the gang could commit any crimes, as long as they commit one.

There are 7 possible schemes: (0), (1), (2), (0,1), (0,2), (1,2), and (0,1,2).

Note:

1 <= G <= 100

0 <= P <= 100

1 <= group[i] <= 100

0 <= profit[i] <= 100

1 <= group.length = profit.length <= 100

Knapsack problem

The knapsack problem or rucksack problem is a problem in combinatorial optimization:

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that

the total weight is less than or equal to a given limit and the total value is as large as possible.

Method 1: 3D DP

class Solution {

public int profitableSchemes(int G, int P, int[] group, int[] profit) {

int MOD = (int)1e9 + 7;

int N = group.length;

long[][][] dp = new long[N+1][G+1][P+1]; //at iteration k, the schema number of i members and j profit

dp[0][0][0] = 1;

for (int k = 0; k < N; k++){

int currG = group[k];

int currP = profit[k];

for (int i = 0; i <= G - currG; i++){

for (int j = 0; j <= P; j++){

dp[k+1][i+currG][Math.min(j+currP, P)] = (dp[k][i][j] + dp[k][i+currG][Math.min(j+currP, P)]) % MOD;

}

}

}

long res = 0;

for (int i = 0; i <= G; i++){

res = (res + dp[N][i][P]) % MOD;

}

return (int) res;

}

}

Method 2: Best solution 2D DP

We use backward to iteration to save one dimension because the bottom right elements depends on the old top left elements.

class Solution {

public int profitableSchemes(int G, int P, int[] group, int[] profit) {

int MOD = (int)1e9 + 7;

int N = group.length;

long[][] dp = new long[G+1][P+1]; // the schema number of i members and j profit

dp[0][0] = 1;

for (int k = 0; k < N; k++){

int currG = group[k];

int currP = profit[k];

for (int i = G- currG; i >= 0; i--){

for (int j = P; j>= 0; j--){

dp[i+currG][Math.min(j+currP, P)] = (dp[i][j] + dp[i+currG][Math.min(j+currP, P)]) % MOD;

}

}

}

long res = 0;

for (int i = 0; i <= G; i++){

res = (res + dp[i][P]) % MOD;

}

return (int) res;

}

}

## 88\_MergeSortedArray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

Note:

You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2.

The number of elements initialized in nums1 and nums2 are m and n respectively.

class Solution {

public void merge(int[] nums1, int m, int[] nums2, int n) {

int index = m + n - 1;

int i = m - 1;

int j = n - 1;

while (index >= 0 && i >= 0 && j >= 0){

if (nums1[i] <= nums2[j]){

nums1[index--] = nums2[j--];

}else{

nums1[index--] = nums1[i--];

}

}

while (j >= 0){

nums1[index--] = nums2[j--];

}

}

}

## 880\_Decoded.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

An encoded string S is given. To find and write the decoded string to a tape, the encoded string is read one character at a time and the following steps are taken:

If the character read is a letter, that letter is written onto the tape.

If the character read is a digit (say d), the entire current tape is repeatedly written d-1 more times in total.

Now for some encoded string S, and an index K, find and return the K-th letter (1 indexed) in the decoded string.

Example 1:

Input: S = "leet2code3", K = 10

Output: "o"

Explanation:

The decoded string is "leetleetcodeleetleetcodeleetleetcode".

The 10th letter in the string is "o".

Example 2:

Input: S = "ha22", K = 5

Output: "h"

Explanation:

The decoded string is "hahahaha". The 5th letter is "h".

Example 3:

Input: S = "a2345678999999999999999", K = 1

Output: "a"

Explanation:

The decoded string is "a" repeated 8301530446056247680 times. The 1st letter is "a".

Note:

2 <= S.length <= 100

S will only contain lowercase letters and digits 2 through 9.

S starts with a letter.

1 <= K <= 10^9

The decoded string is guaranteed to have less than 2^63 letters.

https://leetcode.com/problems/decoded-string-at-index/discuss/156747/C%2B%2BPython-O(N)-Time-O(1)-Space

If it's S[i] = d is a digit, then N = N / d before repeat and K = K % N is what we want.

If it's S[i] = c is a character, we return c if K == 0 or K == N

class Solution {

public String decodeAtIndex(String S, int K) {

long N = 0L;

int i = 0;

for (i = 0; N < K; i++){

char c = S.charAt(i);

if (Character.isLetter(c)){

N++;

}else{

N \*= (int) (c - '0');

}

}

i--;

while (i >= 0){

char c = S.charAt(i);

if (Character.isLetter(c) && (K % N == 0)){

return String.valueOf(c);

}else if (Character.isDigit(c)){

N /= (int)(c - '0');

K %= N;

}else{

N--;

}

i--;

}

return "";

}

}

## 881\_Boats.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The i-th person has weight people[i], and each boat can carry a maximum weight of limit.

Each boat carries at most 2 people at the same time, provided the sum of the weight of those people is at most limit.

Return the minimum number of boats to carry every given person. (It is guaranteed each person can be carried by a boat.)

Example 1:

Input: people = [1,2], limit = 3

Output: 1

Explanation: 1 boat (1, 2)

Example 2:

Input: people = [3,2,2,1], limit = 3

Output: 3

Explanation: 3 boats (1, 2), (2) and (3)

Example 3:

Input: people = [3,5,3,4], limit = 5

Output: 4

Explanation: 4 boats (3), (3), (4), (5)

Note:

1 <= people.length <= 50000

1 <= people[i] <= limit <= 30000

Two points greedy

class Solution {

public int numRescueBoats(int[] people, int limit) {

int res = 0;

Arrays.sort(people);

int i = 0;

int j = people.length - 1;

while (i < j){

if (people[j] > limit || people[i] + people[j] > limit){

res++;

j--;

}else if (people[i] + people[j] <= limit){

res++;

i++;

j--;

}

}

if (i == j){

return res + 1;

}

return res;

}

}

class Solution {

public int numRescueBoats(int[] people, int limit) {

int res = 0;

Arrays.sort(people);

int i = 0;

int j = people.length - 1;

while (i <= j){

res++;

if (people[i] + people[j] <= limit){

i++;

}

j--;

}

return res;

}

}

## 883\_Projection.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a N \* N grid, we place some 1 \* 1 \* 1 cubes that are axis-aligned with the x, y, and z axes.

Each value v = grid[i][j] represents a tower of v cubes placed on top of grid cell (i, j).

Now we view the projection of these cubes onto the xy, yz, and zx planes.

A projection is like a shadow, that maps our 3 dimensional figure to a 2 dimensional plane.

Here, we are viewing the "shadow" when looking at the cubes from the top, the front, and the side.

Return the total area of all three projections.

Example 1:

Input: [[2]]

Output: 5

Example 2:

Input: [[1,2],[3,4]]

Output: 17

Explanation:

Here are the three projections ("shadows") of the shape made with each axis-aligned plane.

Example 3:

Input: [[1,0],[0,2]]

Output: 8

Example 4:

Input: [[1,1,1],[1,0,1],[1,1,1]]

Output: 14

Example 5:

Input: [[2,2,2],[2,1,2],[2,2,2]]

Output: 21

Method 1:

class Solution {

public int projectionArea(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

int countZ = 0;//project along z

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] != 0){

countZ++;

}

}

}

int countY = 0; // project along y

for (int i = 0; i < m; i++){

int max = 0;

for (int j = 0; j < n; j++){

max = Math.max(max, grid[i][j]);

}

countY += max;

}

int countX = 0; //project along x

for (int j = 0; j < n; j++){

int max = 0;

for (int i = 0; i < m; i++){

max = Math.max(max, grid[i][j]);

}

countX += max;

}

return countX + countY + countZ;

}

}

Method 2:

class Solution {

public int projectionArea(int[][] grid) {

int N = grid.length;

int ans = 0;

for (int i = 0; i < N; ++i) {

int bestRow = 0; // largest of grid[i][j]

int bestCol = 0; // largest of grid[j][i]

for (int j = 0; j < N; ++j) {

if (grid[i][j] > 0) ans++; // top shadow

bestRow = Math.max(bestRow, grid[i][j]);

bestCol = Math.max(bestCol, grid[j][i]);

}

ans += bestRow + bestCol;

}

return ans;

}

}

## 884\_Uncommon.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given two sentences A and B. (A sentence is a string of space separated words. Each word consists only of lowercase letters.)

A word is uncommon if it appears exactly once in one of the sentences, and does not appear in the other sentence.

Return a list of all uncommon words.

You may return the list in any order.

Example 1:

Input: A = "this apple is sweet", B = "this apple is sour"

Output: ["sweet","sour"]

Example 2:

Input: A = "apple apple", B = "banana"

Output: ["banana"]

Note:

0 <= A.length <= 200

0 <= B.length <= 200

A and B both contain only spaces and lowercase letters.

Similar as 819. Most Common Word

class Solution {

public String[] uncommonFromSentences(String A, String B) {

String C = A + " " + B;

List<String> list = new ArrayList<>();

String[] strs = C.split("\\s+");

Map<String, Integer> map = new HashMap<>();

for (String str : strs){

map.put(str, map.getOrDefault(str, 0) + 1);

}

for (String str : map.keySet()){

if (map.get(str) == 1){

list.add(str);

}

}

String[] res = new String[list.size()];

for (int i = 0; i < list.size(); i++){

res[i] = list.get(i);

}

return res;

}

}

## 885\_Spiral.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a 2 dimensional grid with R rows and C columns, we start at (r0, c0) facing east.

Here, the north-west corner of the grid is at the first row and column, and the south-east corner of the grid is at the last

row and column.

Now, we walk in a clockwise spiral shape to visit every position in this grid.

Whenever we would move outside the boundary of the grid, we continue our walk outside the grid (but may return to the grid

boundary later.)

Eventually, we reach all R \* C spaces of the grid.

Return a list of coordinates representing the positions of the grid in the order they were visited.

Example 1:

Input: R = 1, C = 4, r0 = 0, c0 = 0

Output: [[0,0],[0,1],[0,2],[0,3]]

Example 2:

Input: R = 5, C = 6, r0 = 1, c0 = 4

Output: [[1,4],[1,5],[2,5],[2,4],[2,3],[1,3],[0,3],[0,4],[0,5],[3,5],[3,4],[3,3],[3,2],[2,2],[1,2],[0,2],[4,5],

[4,4],[4,3],[4,2],[4,1],[3,1],[2,1],[1,1],[0,1],[4,0],[3,0],[2,0],[1,0],[0,0]]

Note:

1 <= R <= 100

1 <= C <= 100

0 <= r0 < R

0 <= c0 < C

Intuition

We can walk in a spiral shape from the starting square, ignoring whether we stay in the grid or not. Eventually,

we must have reached every square in the grid.

Algorithm

Examining the lengths of our walk in each direction, we find the following pattern: 1, 1, 2, 2, 3, 3, 4, 4, ...

That is, we walk 1 unit east, then 1 unit south, then 2 units west, then 2 units north, then 3 units east, etc.

Because our walk is self-similar, this pattern repeats in the way we expect.

class Solution {

public int[][] spiralMatrixIII(int R, int C, int r0, int c0) {

int[][] res = new int[R\*C][2];

int[] dx = {0, 1, 0, -1};

int[] dy = {1, 0, -1, 0};

int count = 1;

int step = 1;

res[0][0] = r0;

res[0][1] = c0;

int r = r0;

int c = c0;

while (count < R \* C){

int i = 0;

while (i < 2){

int j = 0;

while (j < step){

r += dx[i];

c += dy[i];

if (r >= 0 && r < R && c >= 0 && c < C){

res[count][0] = r;

res[count][1] = c;

count++;

}

j++;

}

i++;

}

step++;

while (i < 4){

int j = 0;

while (j < step){

r += dx[i];

c += dy[i];

if (r >= 0 && r < R && c >= 0 && c < C){

res[count][0] = r;

res[count][1] = c;

count++;

}

j++;

}

i++;

}

step++;

}

return res;

}

}

https://leetcode.com/problems/spiral-matrix-iii/discuss/158970/C%2B%2BJavaPython-112233-Steps

## 886\_Possible.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a set of N people (numbered 1, 2, ..., N), we would like to split everyone into two groups of any size.

Each person may dislike some other people, and they should not go into the same group.

Formally, if dislikes[i] = [a, b], it means it is not allowed to put the people numbered a and b into the same group.

Return true if and only if it is possible to split everyone into two groups in this way.

Example 1:

Input: N = 4, dislikes = [[1,2],[1,3],[2,4]]

Output: true

Explanation: group1 [1,4], group2 [2,3]

Example 2:

Input: N = 3, dislikes = [[1,2],[1,3],[2,3]]

Output: false

Example 3:

Input: N = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]

Output: false

Note:

1 <= N <= 2000

0 <= dislikes.length <= 10000

1 <= dislikes[i][j] <= N

dislikes[i][0] < dislikes[i][1]

There does not exist i != j for which dislikes[i] == dislikes[j].

DFS

Time complexity: O(V + E)

class Solution {

public boolean possibleBipartition(int N, int[][] dislikes) {

//corner case

if (dislikes == null || dislikes.length == 0){

return true;

}

//build graph

Map<Integer, Set<Integer>> graph = new HashMap<>();

for (int[] d : dislikes){

if (!graph.containsKey(d[0])){

graph.put(d[0], new HashSet<>());

}

graph.get(d[0]).add(d[1]);

if (!graph.containsKey(d[1])){

graph.put(d[1], new HashSet<>());

}

graph.get(d[1]).add(d[0]);

}

//now it will be the same as Leetcode #785

int[] color = new int[N+1];

for (int i = 1; i <= N; i++){

if (!graph.containsKey(i)){

continue;

}

if (color[i] == 0){

color[i] = 1;

}

if (isConflict(graph, color, i)){

return false;

}

}

return true;

}

private boolean isConflict(Map<Integer, Set<Integer>> graph, int[] color, int start){

for (int nei : graph.get(start)){

if (color[nei] == 0){

color[nei] = -color[start];

if (isConflict(graph, color, nei)){

return true;

}

}else{

if (color[nei] == color[start]){

return true;

}

}

}

return false;

}

}

Better version: the same as https://github.com/optimisea/Leetcode/blob/master/Java/785\_Is.java

class Solution {

public boolean possibleBipartition(int N, int[][] dislikes) {

Map<Integer, Set<Integer>> map = new HashMap<>();

for (int i = 1; i <= N; i++){

map.put(i, new HashSet<>());

}

for (int[] dislike : dislikes){

map.get(dislike[0]).add(dislike[1]);

map.get(dislike[1]).add(dislike[0]);

}

int[] group = new int[N+1];

for (int i = 1; i <= N; i++){

if (group[i] == 0){

if (!possible(map, group, i, 1)){

return false;

}

}

}

return true;

}

private boolean possible(Map<Integer, Set<Integer>> map, int[] group, int node, int color){

group[node] = color;

for (int nei : map.get(node)){

if (group[nei] == color){

return false;

}else if (group[nei] == 0){

if (!possible(map, group, nei, -color)){

return false;

}

}

}

return true;

}

}

## 887\_Super.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given K eggs, and you have access to a building with N floors from 1 to N.

Each egg is identical in function, and if an egg breaks, you cannot drop it again.

You know that there exists a floor F with 0 <= F <= N such that any egg dropped at a floor higher than F will break, and any egg

dropped at or below floor F will not break.

Each move, you may take an egg (if you have an unbroken one) and drop it from any floor X (with 1 <= X <= N).

Your goal is to know with certainty what the value of F is.

What is the minimum number of moves that you need to know with certainty what F is, regardless of the initial value of F?

Example 1:

Input: K = 1, N = 2

Output: 2

Explanation:

Drop the egg from floor 1. If it breaks, we know with certainty that F = 0.

Otherwise, drop the egg from floor 2. If it breaks, we know with certainty that F = 1.

If it didn't break, then we know with certainty F = 2.

Hence, we needed 2 moves in the worst case to know what F is with certainty.

Example 2:

Input: K = 2, N = 6

Output: 3

Example 3:

Input: K = 3, N = 14

Output: 4

Note:

1 <= K <= 100

1 <= N <= 10000

Method 1: DP + Bruce Force

Time complexity: O(KN^2)

https://algorithms.tutorialhorizon.com/dynamic-programming-egg-dropping-problem/

class Solution {

public int superEggDrop(int K, int N) {

int[][] cache = new int[K+1][N+1];

for (int i = 0; i <= K; i++){

Arrays.fill(cache[i], -1);

}

return minNum(K, N, cache);

}

private int minNum(int K, int N, int[][] cache){

if (cache[K][N] != -1){

return cache[K][N];

}

if (K == 1 || N <= 1){

return N;

}

int min = Integer.MAX\_VALUE;

for (int i = 1; i <= N; i++){

int eggBreak = minNum(K-1, i-1, cache);

int notBreak = minNum(K, N-i, cache);

min = Math.min(min, Math.max(eggBreak, notBreak) + 1);

}

cache[K][N] = min;

return min;

}

}

Method 2: DP + binary search

Time complexity: O(KNlogN)

https://leetcode.com/problems/super-egg-drop/solution/

https://leetcode.com/problems/super-egg-drop/discuss/159055/Java-DP-solution-from-O(KN2)-to-O(KNlogN)

Method 3: DP

Time complexity: KLogN

https://leetcode.com/problems/super-egg-drop/discuss/158974/C++JavaPython-2D-and-1D-DP-O(KlogN)

dp[M][K]means that, given K eggs and M moves, what is the maximum number of floor that we can check.

dp[m-1][k-1]: egg breaks

dp[m-1][k]: egg survives

dp[m][k] is similar to the number of combinations and it increase exponentially to N

class Solution {

public int superEggDrop(int K, int N) {

int[][] dp = new int[N+1][K+1];//dp means the maximum number of floor that we can check when K eggs and M moves

int m = 0;

while (dp[m][K] < N){

m++;

for (int k = 1; k <= K; k++){

dp[m][k] = dp[m-1][k-1] + dp[m-1][k] + 1;

}

}

return m;

}

}

## 888\_Fair.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Alice and Bob have candy bars of different sizes: A[i] is the size of the i-th bar of candy that Alice has, and B[j] is the size of

the j-th bar of candy that Bob has.

Since they are friends, they would like to exchange one candy bar each so that after the exchange, they both have the same total

amount of candy. (The total amount of candy a person has is the sum of the sizes of candy bars they have.)

Return an integer array ans where ans[0] is the size of the candy bar that Alice must exchange, and ans[1] is the size of the

candy bar that Bob must exchange.

If there are multiple answers, you may return any one of them. It is guaranteed an answer exists.

Example 1:

Input: A = [1,1], B = [2,2]

Output: [1,2]

Example 2:

Input: A = [1,2], B = [2,3]

Output: [1,2]

Example 3:

Input: A = [2], B = [1,3]

Output: [2,3]

Example 4:

Input: A = [1,2,5], B = [2,4]

Output: [5,4]

Note:

1 <= A.length <= 10000

1 <= B.length <= 10000

1 <= A[i] <= 100000

1 <= B[i] <= 100000

It is guaranteed that Alice and Bob have different total amounts of candy.

It is guaranteed there exists an answer.

Method 1:

class Solution {

public int[] fairCandySwap(int[] A, int[] B) {

int[] res = new int[2];

int sumA = 0;

Set<Integer> setA = new HashSet<>();

for (int i : A){

sumA += i;

setA.add(i);

}

int sumB = 0;

Set<Integer> setB = new HashSet<>();

for (int i : B){

sumB += i;

setB.add(i);

}

int avg = (sumA + sumB)/2;

if (sumA > avg){

for (int i : A){

if (setB.contains(i-(sumA-avg))){

res[0] = i;

res[1] = i - (sumA-avg);

break;

}

}

}else if (sumA < avg){

for (int i : A){

if (setB.contains(i+(avg-sumA))){

res[0] = i;

res[1] = i + (avg - sumA);

break;

}

}

}else{

res[0] = 0;

res[1] = 0;

}

return res;

}

}

Method 2: Better solution

class Solution {

public int[] fairCandySwap(int[] A, int[] B) {

int[] res = new int[2];

int sumA = 0;

Set<Integer> setA = new HashSet<>();

for (int i : A){

sumA += i;

setA.add(i);

}

int sumB = 0;

for (int i : B){

sumB += i;

}

int diff = (sumA - sumB)/2;

for (int i : B){

if (setA.contains(i+diff)){

res[0] = i + diff;

res[1] = i;

return res;

}

}

return new int[0];

}

}

class Solution {

public int[] fairCandySwap(int[] A, int[] B) {

int[] res = new int[2];

Set<Integer> setA = new HashSet<>();

int sumA = 0;

for (int i : A){

sumA += i;

setA.add(i);

}

int sumB = 0;

Set<Integer> setB = new HashSet<>();

for (int i : B){

sumB += i;

setB.add(i);

}

int avg = (sumA + sumB) / 2;

if (sumA > avg){

int diff = sumA - avg;

for (int i : A){

if (setB.contains(i-diff)){

res[0] = i ;

res[1] = i - diff;

return res;

}

}

}else{

int diff = sumB - avg;

for (int i : B){

if (setA.contains(i-diff)){

res[0] = i - diff;

res[1] = i;

return res;

}

}

}

return res;

}

}

## 889\_Construct.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Return any binary tree that matches the given preorder and postorder traversals.

Values in the traversals pre and post are distinct positive integers.

Example 1:

Input: pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]

Output: [1,2,3,4,5,6,7]

Note:

1 <= pre.length == post.length <= 30

pre[] and post[] are both permutations of 1, 2, ..., pre.length.

It is guaranteed an answer exists. If there exists multiple answers, you can return any of them.

Method 1: Recursion

Time complexity: O(nlogn)

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode constructFromPrePost(int[] pre, int[] post) {

return dfs(pre, 0, pre.length - 1, post, 0, post.length - 1);

}

private TreeNode dfs(int[] pre, int preStart, int preEnd, int[] post, int postStart, int postEnd){

if (preStart > preEnd || postStart > postEnd){

return null;

}

TreeNode root = new TreeNode(pre[preStart]);

if (preStart + 1 <= preEnd){

int deltaIndex = findIndex(post, postStart, postEnd - 1, pre[preStart+1]);

root.left = dfs(pre, preStart + 1, preStart + 1 + deltaIndex, post, postStart, postStart + deltaIndex);

root.right = dfs(pre, preStart + 1 + deltaIndex + 1, preEnd, post, postStart + deltaIndex + 1, postEnd - 1);

}

return root;

}

private int findIndex(int[] post, int postStart, int postEnd, int target){

for (int i = postStart; i <= postEnd; i++){

if (post[i] == target){

return i - postStart;

}

}

return -1;

}

}

Method 2: HashMap + Recursion (Better)

Time complexity: O(n + logn)

class Solution {

public TreeNode constructFromPrePost(int[] pre, int[] post) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < post.length; i++){

map.put(post[i], i);

}

return dfs(pre, 0, pre.length - 1, post, 0, post.length - 1, map);

}

private TreeNode dfs(int[] pre, int preStart, int preEnd, int[] post, int postStart, int postEnd, Map<Integer, Integer> map){

if (preStart > preEnd || postStart > postEnd){

return null;

}

TreeNode root = new TreeNode(pre[preStart]);

if (preStart + 1 <= preEnd){

int deltaIndex = map.get(pre[preStart+1]) - postStart;

root.left = dfs(pre, preStart + 1, preStart + 1 + deltaIndex, post, postStart, postStart + deltaIndex, map);

root.right = dfs(pre, preStart + 1 + deltaIndex + 1, preEnd, post, postStart + deltaIndex + 1, postEnd - 1, map);

}

return root;

}

}

Key idea: use the next element after root to find the split index

class Solution {

public TreeNode constructFromPrePost(int[] pre, int[] post) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < post.length; i++){

map.put(post[i], i);

}

return construct(pre, 0, pre.length - 1, post, 0, post.length - 1, map);

}

private TreeNode construct(int[] pre, int preStart, int preEnd, int[] post, int postStart, int postEnd, Map<Integer, Integer> map){

if (preStart > preEnd || postStart > postEnd){

return null;

}

int pivot = pre[preStart];

TreeNode root = new TreeNode(pivot);

if (preStart + 1 <= preEnd){

int delta = map.get(pre[preStart+1]) - postStart + 1;

root.left = construct(pre, preStart + 1, preStart + delta, post, postStart, postStart + delta-1, map);

root.right = construct(pre, preStart + delta + 1, preEnd, post, postStart + delta, postEnd - 1, map);

}

return root;

}

}

## 89\_GrayCode.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code.

A gray code sequence must begin with 0.

For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:

00 - 0

01 - 1

11 - 3

10 - 2

Note:

For a given n, a gray code sequence is not uniquely defined.

class Solution {

public List<Integer> grayCode(int n) {

List<Integer> result = new ArrayList<>();

for (int i = 0; i < 1<<n ; i++){

result.add(i ^ i>>1);

}

return result;

}

}

Better version:

backtrack

https://leetcode.com/problems/gray-code/discuss/29880/Backtracking-C++-solution

https://www.tutorialspoint.com/java/java\_bitset\_class.htm

class Solution {

public List<Integer> grayCode(int n) {

List<Integer> res = new ArrayList<>();

backtrack(res, n, new BitSet());

return res;

}

private void backtrack(List<Integer> res, int n, BitSet chosen){

if (n == 0){

res.add(convert(chosen));

return;

}

backtrack(res, n-1, chosen);

chosen.flip(n-1);

backtrack(res, n-1, chosen);

}

private int convert(BitSet bits){

int res = 0;

for (int i = 0; i < bits.length(); i++){

res += bits.get(i) ? (1 << i) : 0;

}

return res;

}

}

## 89\_kSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n distinct positive integers, integer k (k <= n) and a number target.

Find k numbers where sum is target. Calculate how many solutions there are?

Have you met this question in a real interview?

Example

Given [1,2,3,4], k = 2, target = 5.

There are 2 solutions: [1,4] and [2,3].

Return 2.

n个数,取k个数,组成和为target

• State: f[i][j][t]前i个数取j个数出来能否和为t

• Function: f[i][j][t] = f[i - 1][j - 1][t - a[i-1]] + f[i - 1][j][t]

• Intialization: f[i][0][0] = 1

• Answer: f[n][k][target]

Method 1:

Space complexity: O(A.length\*k\*target)

public class Solution {

/\*\*

\* @param A: An integer array

\* @param k: A positive integer (k <= length(A))

\* @param target: An integer

\* @return: An integer

\*/

public int kSum(int[] A, int k, int target) {

int n = A.length;

int[][][] dp = new int[n+1][k+1][target+1];

for (int i = 0; i <= n; i++){

dp[i][0][0] = 1;

}

for (int i = 1; i <= n; i++){

for (int j = 1; j <= k; j++){

for (int t = 1; t <= target; t++){

if (t >= A[i-1]){

dp[i][j][t] = dp[i-1][j-1][t-A[i-1]] + dp[i-1][j][t];

}else{

dp[i][j][t] = dp[i-1][j][t];

}

}

}

}

return dp[n][k][target];

}

}

Method 2:

Space complexity: O(k\*target)

public class Solution {

/\*\*

\* @param A: An integer array

\* @param k: A positive integer (k <= length(A))

\* @param target: An integer

\* @return: An integer

\*/

public int kSum(int[] A, int k, int target) {

int n = A.length;

int[][][] dp = new int[2][k+1][target+1];

for (int i = 0; i <= 1; i++){

dp[i][0][0] = 1;

}

for (int i = 1; i <= n; i++){

for (int j = 1; j <= k; j++){

for (int t = 1; t <= target; t++){

if (t >= A[i-1]){

dp[i%2][j][t] = dp[(i-1)%2][j-1][t-A[i-1]] + dp[(i-1)%2][j][t];

}else{

dp[i%2][j][t] = dp[(i-1)%2][j][t];

}

}

}

}

return dp[n%2][k][target];

}

}

## 890\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have a list of words and a pattern, and you want to know which words in words matches the pattern.

A word matches the pattern if there exists a permutation of letters p so that after replacing every letter x in the pattern with p(x), we get the desired word.

(Recall that a permutation of letters is a bijection from letters to letters: every letter maps to another letter, and no two letters map to the same letter.)

Return a list of the words in words that match the given pattern.

You may return the answer in any order.

Example 1:

Input: words = ["abc","deq","mee","aqq","dkd","ccc"], pattern = "abb"

Output: ["mee","aqq"]

Explanation: "mee" matches the pattern because there is a permutation {a -> m, b -> e, ...}.

"ccc" does not match the pattern because {a -> c, b -> c, ...} is not a permutation,

since a and b map to the same letter.

Note:

1 <= words.length <= 50

1 <= pattern.length = words[i].length <= 20

Method 1: Two maps

class Solution {

public List<String> findAndReplacePattern(String[] words, String pattern) {

List<String> res = new ArrayList<>();

for (String word : words){

if (isMatch(word, pattern)){

res.add(word);

}

}

return res;

}

private boolean isMatch(String word, String pattern){

if (word.length() != pattern.length()){

return false;

}

char[] wordArr = word.toCharArray();

char[] patternArr = pattern.toCharArray();

Map<Character, Character> forward = new HashMap<>();

for (int i = 0; i < wordArr.length; i++){

if (!forward.containsKey(wordArr[i])){

forward.put(wordArr[i], patternArr[i]);

}else{

if (forward.get(wordArr[i]) != patternArr[i]){

return false;

}

}

}

Map<Character, Character> backward = new HashMap<>();

for (int i = 0; i < wordArr.length; i++){

if (!backward.containsKey(patternArr[i])){

backward.put(patternArr[i], wordArr[i]);

}else{

if (backward.get(patternArr[i]) != wordArr[i]){

return false;

}

}

}

return true;

}

}

class Solution {

public List<String> findAndReplacePattern(String[] words, String pattern) {

List<String> res = new ArrayList<>();

for (String word : words){

if (isMatch(word, pattern)){

res.add(word);

}

}

return res;

}

private boolean isMatch(String word, String pattern){

if (word.length() != pattern.length()){

return false;

}

char[] wordArr = word.toCharArray();

char[] patternArr = pattern.toCharArray();

Map<Character, Character> forward = new HashMap<>();

Map<Character, Character> backward = new HashMap<>();

for (int i = 0; i < wordArr.length; i++){

if (!forward.containsKey(wordArr[i])){

forward.put(wordArr[i], patternArr[i]);

}

if (!backward.containsKey(patternArr[i])){

backward.put(patternArr[i], wordArr[i]);

}

if (forward.get(wordArr[i]) != patternArr[i] || backward.get(patternArr[i]) != wordArr[i]){

return false;

}

}

return true;

}

}

One map

class Solution {

public List<String> findAndReplacePattern(String[] words, String pattern) {

List<String> res = new ArrayList<>();

for (String word : words){

if (isMatch(word, pattern)){

res.add(word);

}

}

return res;

}

private boolean isMatch(String word, String pattern){

if (word.length() != pattern.length()){

return false;

}

char[] wordArr = word.toCharArray();

char[] patternArr = pattern.toCharArray();

Map<Character, Character> forward = new HashMap<>();

for (int i = 0; i < wordArr.length; i++){

if (!forward.containsKey(wordArr[i])){

forward.put(wordArr[i], patternArr[i]);

}else{

if (forward.get(wordArr[i]) != patternArr[i]){

return false;

}

}

}

forward.clear();

for (int i = 0; i < wordArr.length; i++){

if (!forward.containsKey(patternArr[i])){

forward.put(patternArr[i], wordArr[i]);

}else{

if (forward.get(patternArr[i]) != wordArr[i]){

return false;

}

}

}

return true;

}

}

## 891\_Sum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers A, consider all non-empty subsequences of A.

For any sequence S, let the width of S be the difference between the maximum and minimum element of S.

Return the sum of the widths of all subsequences of A.

As the answer may be very large, return the answer modulo 10^9 + 7.

Example 1:

Input: [2,1,3]

Output: 6

Explanation:

Subsequences are [1], [2], [3], [2,1], [2,3], [1,3], [2,1,3].

The corresponding widths are 0, 0, 0, 1, 1, 2, 2.

The sum of these widths is 6.

Note:

1 <= A.length <= 20000

1 <= A[i] <= 20000

The order in initial arrays doesn't matter, So sort it.

class Solution {

public int sumSubseqWidths(int[] A) {

long MOD = (long)1e9 + 7;

int N = A.length;

long res = 0;

Arrays.sort(A);

long c = 1;

for (int i = 0; i < N; i++){

//res = (res + (A[i] \* (long)(Math.pow(2, i) - (long)Math.pow(2, N-1-i) % MOD)) %MOD) % MOD;

res = (res + (A[i] \* c - A[N-1-i] \* c)) % MOD;

c = (c << 1) % MOD;

}

return (int)((res + MOD) % MOD);

}

}

## 892\_Surface.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a N \* N grid, we place some 1 \* 1 \* 1 cubes.

Each value v = grid[i][j] represents a tower of v cubes placed on top of grid cell (i, j).

Return the total surface area of the resulting shapes.

Example 1:

Input: [[2]]

Output: 10

Example 2:

Input: [[1,2],[3,4]]

Output: 34

Example 3:

Input: [[1,0],[0,2]]

Output: 16

Example 4:

Input: [[1,1,1],[1,0,1],[1,1,1]]

Output: 32

Example 5:

Input: [[2,2,2],[2,1,2],[2,2,2]]

Output: 46

class Solution {

public int surfaceArea(int[][] grid) {

int N = grid.length;

int res = 0;

for (int i = 0; i < N; i++){

for (int j = 0; j < N; j++){

if (grid[i][j] == 0){

continue;

}

res += grid[i][j] \* 4 + 2;

if (i + 1 < N && grid[i+1][j] != 0){

res -= 2 \* Math.min(grid[i][j], grid[i+1][j]);

}

if (j + 1 < N && grid[i][j+1] != 0){

res -= 2 \* Math.min(grid[i][j], grid[i][j+1]);

}

}

}

return res;

}

}

893\_Groups.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given an array A of strings.

Two strings S and T are special-equivalent if after any number of moves, S == T.

A move consists of choosing two indices i and j with i % 2 == j % 2, and swapping S[i] with S[j].

Now, a group of special-equivalent strings from A is a non-empty subset S of A such that any string not in S is not special-equivalent

with any string in S.

Return the number of groups of special-equivalent strings from A.

Example 1:

Input: ["a","b","c","a","c","c"]

Output: 3

Explanation: 3 groups ["a","a"], ["b"], ["c","c","c"]

Example 2:

Input: ["aa","bb","ab","ba"]

Output: 4

Explanation: 4 groups ["aa"], ["bb"], ["ab"], ["ba"]

Example 3:

Input: ["abc","acb","bac","bca","cab","cba"]

Output: 3

Explanation: 3 groups ["abc","cba"], ["acb","bca"], ["bac","cab"]

Example 4:

Input: ["abcd","cdab","adcb","cbad"]

Output: 1

Explanation: 1 group ["abcd","cdab","adcb","cbad"]

Note:

1 <= A.length <= 1000

1 <= A[i].length <= 20

All A[i] have the same length.

All A[i] consist of only lowercase letters.

Key point: convert to str when you try to compare two hashmap

class Solution {

public int numSpecialEquivGroups(String[] A) {

Set<String> set = new HashSet<>();

for (String S : A){

int[] odd = new int[26];

int[] even = new int[26];

for (int i = 0; i < S.length(); i++){

char c = S.charAt(i);

if (i % 2 == 1){

odd[c - 'a']++;

}else{

even[c - 'a']++;

}

}

String str = Arrays.toString(odd) + Arrays.toString(even); // convert to str when you try to compare two hashmap

set.add(str);

}

return set.size();

}

}

894\_All.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A full binary tree is a binary tree where each node has exactly 0 or 2 children.

Return a list of all possible full binary trees with N nodes. Each element of the answer is the root node of one possible tree.

Each node of each tree in the answer must have node.val = 0.

You may return the final list of trees in any order.

Example 1:

Input: 7

Output: [[0,0,0,null,null,0,0,null,null,0,0],[0,0,0,null,null,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,null,null,null,null,0,0],[0,0,0,0,0,null,null,0,0]]

Explanation:

Note:

1 <= N <= 20

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

Method 1: without memo

class Solution {

public List<TreeNode> allPossibleFBT(int N) {

List<TreeNode> res = new ArrayList<>();

if (N % 2 == 0){

return res;

}

return getAll(N);

}

private List<TreeNode> getAll(int N){

List<TreeNode> res = new ArrayList<>();

if (N == 1){

res.add(new TreeNode(0));

return res;

}

for (int i = 1; i < N; i+= 2){

List<TreeNode> leftList = getAll(i);

List<TreeNode> rightList = getAll(N-1-i);

for (TreeNode left : leftList){

for (TreeNode right : rightList){

TreeNode root = new TreeNode(0);

root.left = left;

root.right = right;

res.add(root);

}

}

}

return res;

}

}

class Solution {

public List<TreeNode> allPossibleFBT(int N) {

List<TreeNode> res = new ArrayList<>();

if (N <= 0 && N % 2 == 0){

return res;

}

if (N == 1){

TreeNode root = new TreeNode(0);

res.add(root);

return res;

}

for (int i = 1; i < N; i++){

List<TreeNode> left = allPossibleFBT(i);

List<TreeNode> right = allPossibleFBT(N-1-i);

for (TreeNode l : left){

for (TreeNode r : right){

TreeNode root = new TreeNode(0);

root.left = l;

root.right = r;

res.add(root);

}

}

}

return res;

}

}

Method 2: add memo

Time complexity: O(2^N)

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<TreeNode> allPossibleFBT(int N) {

List<TreeNode> res = new ArrayList<>();

if (N % 2 == 0){

return res;

}

return getAll(N);

}

Map<Integer, List<TreeNode>> memo = new HashMap<>();

private List<TreeNode> getAll(int N){

List<TreeNode> res = new ArrayList<>();

if (N == 1){

res.add(new TreeNode(0));

return res;

}

if (memo.containsKey(N)){

return memo.get(N);

}

for (int i = 1; i < N; i+= 2){

List<TreeNode> leftList = getAll(i);

List<TreeNode> rightList = getAll(N-1-i);

for (TreeNode left : leftList){

for (TreeNode right : rightList){

TreeNode root = new TreeNode(0);

root.left = left;

root.right = right;

res.add(root);

}

}

}

memo.put(N, res);

return res;

}

}

class Solution {

Map<Integer, List<TreeNode>> map = new HashMap<>();

public List<TreeNode> allPossibleFBT(int N) {

if (map.containsKey(N)){

return map.get(N);

}

List<TreeNode> res = new ArrayList<>();

if (N <= 0 && N % 2 == 0){

return res;

}

if (N == 1){

TreeNode root = new TreeNode(0);

res.add(root);

map.put(1, res);

return res;

}

for (int i = 1; i < N; i++){

List<TreeNode> left = allPossibleFBT(i);

List<TreeNode> right = allPossibleFBT(N-1-i);

for (TreeNode l : left){

for (TreeNode r : right){

TreeNode root = new TreeNode(0);

root.left = l;

root.right = r;

res.add(root);

}

}

}

map.put(N, res);

return res;

}

}

895\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implement FreqStack, a class which simulates the operation of a stack-like data structure.

FreqStack has two functions:

push(int x), which pushes an integer x onto the stack.

pop(), which removes and returns the most frequent element in the stack.

If there is a tie for most frequent element, the element closest to the top of the stack is removed and returned.

Example 1:

Input:

["FreqStack","push","push","push","push","push","push","pop","pop","pop","pop"],

[[],[5],[7],[5],[7],[4],[5],[],[],[],[]]

Output: [null,null,null,null,null,null,null,5,7,5,4]

Explanation:

After making six .push operations, the stack is [5,7,5,7,4,5] from bottom to top. Then:

pop() -> returns 5, as 5 is the most frequent.

The stack becomes [5,7,5,7,4].

pop() -> returns 7, as 5 and 7 is the most frequent, but 7 is closest to the top.

The stack becomes [5,7,5,4].

pop() -> returns 5.

The stack becomes [5,7,4].

pop() -> returns 4.

The stack becomes [5,7].

Note:

Calls to FreqStack.push(int x) will be such that 0 <= x <= 10^9.

It is guaranteed that FreqStack.pop() won't be called if the stack has zero elements.

The total number of FreqStack.push calls will not exceed 10000 in a single test case.

The total number of FreqStack.pop calls will not exceed 10000 in a single test case.

The total number of FreqStack.push and FreqStack.pop calls will not exceed 150000 across all test cases.

Method 1: TLE

Push: O(1)

Pop: O(n)

class FreqStack {

Map<Integer, Integer> map;

Stack<Integer> stack;

public FreqStack() {

map = new HashMap<>();

stack = new Stack<>();

}

public void push(int x) {

map.put(x, map.getOrDefault(x, 0) + 1);

stack.push(x);

}

public int pop() {

int res = 0;

Stack<Integer> temp = new Stack<>();

Set<Integer> set = new HashSet<>();

int maxFreq = 0;

for (int i : map.keySet()){

maxFreq = Math.max(maxFreq, map.get(i));

}

for (int i : map.keySet()){

if (maxFreq == map.get(i)){

set.add(i);

}

}

while (!stack.isEmpty()){

if (set.contains(stack.peek())){

res = stack.pop();

map.put(res, map.get(res) - 1);

while (!temp.isEmpty()){

stack.push(temp.pop());

}

break;

}else{

temp.push(stack.pop());

}

}

return res;

}

}

/\*\*

\* Your FreqStack object will be instantiated and called as such:

\* FreqStack obj = new FreqStack();

\* obj.push(x);

\* int param\_2 = obj.pop();

\*/

Method 2: Best solution

Push: O(1)

Pop: O(1)

class FreqStack {

Map<Integer, Integer> freqMap; // key: num; value: frequency

Map<Integer, Stack<Integer>> stackMap; // key: frequency; value: stack of num

int maxFreq;

public FreqStack() {

freqMap = new HashMap<>();

stackMap = new HashMap<>();

maxFreq = 0;

}

public void push(int x) {

int freq = freqMap.getOrDefault(x, 0) + 1;

maxFreq = Math.max(maxFreq, freq);

freqMap.put(x, freq);

if (!stackMap.containsKey(freq)){

stackMap.put(freq, new Stack<Integer>());

}

stackMap.get(freq).push(x);

}

public int pop() {

int val = stackMap.get(maxFreq).pop();

freqMap.put(val, freqMap.get(val) - 1);

if (stackMap.get(maxFreq).isEmpty()){

maxFreq--;

}

return val;

}

}

896\_Monotonic.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

An array is monotonic if it is either monotone increasing or monotone decreasing.

An array A is monotone increasing if for all i <= j, A[i] <= A[j]. An array A is monotone decreasing if for all i <= j, A[i] >= A[j].

Return true if and only if the given array A is monotonic.

Example 1:

Input: [1,2,2,3]

Output: true

Example 2:

Input: [6,5,4,4]

Output: true

Example 3:

Input: [1,3,2]

Output: false

Example 4:

Input: [1,2,4,5]

Output: true

Example 5:

Input: [1,1,1]

Output: true

Note:

1 <= A.length <= 50000

-100000 <= A[i] <= 100000

Method 1: one pass

class Solution {

public boolean isMonotonic(int[] A) {

if (A.length == 1){

return true;

}

boolean isFlat = true;

boolean increase = false;

for (int i = 1; i < A.length; i++){

if (isFlat){

if (A[i] > A[i-1]){

increase = true;

isFlat = false;

}else if (A[i] < A[i-1]){

increase = false;

isFlat = false;

}

}else{

if (increase && A[i] < A[i-1] || !increase && A[i] > A[i-1]){

return false;

}

}

}

return true;

}

}

class Solution {

public boolean isMonotonic(int[] A) {

if (A.length == 1){

return true;

}

boolean increase = true;

boolean decrease = true;

for (int i = 1; i < A.length; i++){

increase &= A[i] >= A[i-1];

decrease &= A[i] <= A[i-1];

}

return increase || decrease;

}

}

class Solution {

public boolean isMonotonic(int[] A) {

boolean inc = true;

boolean dec = true;

for (int i = 1; i < A.length; i++){

inc = inc && (A[i] >= A[i-1]);

dec = dec && (A[i] <= A[i-1]);

}

return inc || dec;

}

}

Method 2: two pass

class Solution {

public boolean isMonotonic(int[] A) {

return increasing(A) || decreasing(A);

}

public boolean increasing(int[] A) {

for (int i = 0; i < A.length - 1; ++i)

if (A[i] > A[i+1]) return false;

return true;

}

public boolean decreasing(int[] A) {

for (int i = 0; i < A.length - 1; ++i)

if (A[i] < A[i+1]) return false;

return true;

}

}

897\_Increasing.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a tree, rearrange the tree in in-order so that the leftmost node in the tree is now the root of the tree, and every node has no left child and only 1 right child.

Example 1:

Input: [5,3,6,2,4,null,8,1,null,null,null,7,9]

5

/ \

3 6

/ \ \

2 4 8

/ / \

1 7 9

Output: [1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]

1

\

2

\

3

\

4

\

5

\

6

\

7

\

8

\

9

Note:

The number of nodes in the given tree will be between 1 and 100.

Each node will have a unique integer value from 0 to 1000.

Method 1:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode increasingBST(TreeNode root) {

List<Integer> list = new ArrayList<>();

inOrder(root, list);

TreeNode newRoot = new TreeNode(list.get(0));

TreeNode node = newRoot;

for (int i = 1; i < list.size(); ++i){

node.right = new TreeNode(list.get(i));

node = node.right;

}

return newRoot;

}

private void inOrder(TreeNode root, List<Integer> list){

if (root == null){

return;

}

inOrder(root.left, list);

list.add(root.val);

inOrder(root.right, list);

}

}

Best solution:

class Solution {

public TreeNode increasingBST(TreeNode root) {

return dfs(root, null);

}

private TreeNode dfs(TreeNode root, TreeNode tail){

if (root == null){

return tail;

}

TreeNode res = dfs(root.left, root);

root.left = null;

root.right = dfs(root.right, tail);

return res;

}

}

Iteration: based on inorder traversal template

class Solution {

public TreeNode increasingBST(TreeNode root) {

Stack<TreeNode> stack = new Stack<>();

TreeNode node = root;

TreeNode newRoot = null;

TreeNode prev = null;

while (node != null || !stack.isEmpty()){

while (node != null){

stack.push(node);

node = node.left;

}

TreeNode curr = stack.pop();

if (prev == null){

newRoot = curr;

}else{

prev.right = curr;

}

curr.left = null;

prev = curr;

node = curr.right;

}

return newRoot;

}

}

898\_Bitwise.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have an array A of non-negative integers.

For every (contiguous) subarray B = [A[i], A[i+1], ..., A[j]] (with i <= j), we take the bitwise OR of all the elements in B, obtaining a result A[i] | A[i+1] | ... | A[j].

Return the number of possible results. (Results that occur more than once are only counted once in the final answer.)

Example 1:

Input: [0]

Output: 1

Explanation:

There is only one possible result: 0.

Example 2:

Input: [1,1,2]

Output: 3

Explanation:

The possible subarrays are [1], [1], [2], [1, 1], [1, 2], [1, 1, 2].

These yield the results 1, 1, 2, 1, 3, 3.

There are 3 unique values, so the answer is 3.

Example 3:

Input: [1,2,4]

Output: 6

Explanation:

The possible results are 1, 2, 3, 4, 6, and 7.

Note:

1 <= A.length <= 50000

0 <= A[i] <= 10^9

Method 1: Brute Force (TLE)

Time complexity: O(N^3)

Space complexity: O(N)

class Solution {

public int subarrayBitwiseORs(int[] A) {

Set<Integer> set = new HashSet<>();

for (int i = 0; i < A.length; i++){

for (int j = i; j < A.length; j++){

int res = 0;

int k = i;

while (k <= j){

res |= A[k];

k++;

}

set.add(res);

}

}

return set.size();

}

}

Method 2: Brute Force (TLE)

Time complexity: O(N^2)

Space complexity: O(N)

class Solution {

public int subarrayBitwiseORs(int[] A) {

Set<Integer> set = new HashSet<>();

for (int i = 0; i < A.length; i++){

int res = A[i];

for (int j = i; j < A.length; j++){

res |= A[j];

set.add(res);

}

}

return set.size();

}

}

Method 3:

Time complexity: O(30\*N)

https://leetcode.com/problems/bitwise-ors-of-subarrays/discuss/165881/C++JavaPython-O(30N)

Assume B[i][j] = A[i] | A[i+1] | ... | A[j]

Hash set cur stores all wise B[0][i], B[1][i], B[2][i], B[i][i].

When we handle the A[i+1], we want to update cur

So we need operate bitwise OR on all elements in cur.

Also we need to add A[i+1] to cur.

In each turn, we add all elements in cur to res.

class Solution {

public int subarrayBitwiseORs(int[] A) {

Set<Integer> res = new HashSet<>();

Set<Integer> curr = new HashSet<>();

for (int i : A){

Set<Integer> next = new HashSet<>(); //store all the res for B[0][i] .... B[i][i]

next.add(i);

for (int j : curr){

next.add(i|j);

}

res.addAll(next);

curr = next;

}

return res.size();

}

}

899\_Orderly.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A string S of lowercase letters is given. Then, we may make any number of moves.

In each move, we choose one of the first K letters (starting from the left), remove it, and place it at the end of the string.

Return the lexicographically smallest string we could have after any number of moves.

Example 1:

Input: S = "cba", K = 1

Output: "acb"

Explanation:

In the first move, we move the 1st character ("c") to the end, obtaining the string "bac".

In the second move, we move the 1st character ("b") to the end, obtaining the final result "acb".

Example 2:

Input: S = "baaca", K = 3

Output: "aaabc"

Explanation:

In the first move, we move the 1st character ("b") to the end, obtaining the string "aacab".

In the second move, we move the 3rd character ("c") to the end, obtaining the final result "aaabc".

Note:

1 <= K <= S.length <= 1000

S consists of lowercase letters only.

When K > 1, we can move char to any position

class Solution {

public String orderlyQueue(String S, int K) {

if (K > 1){

char[] arr = S.toCharArray();

Arrays.sort(arr);

return new String(arr);

}

String res = S;

for (int i = 1; i < S.length(); i++){

String temp = S.substring(i) + S.substring(0, i);

if (res.compareTo(temp) > 0){

res = temp;

}

}

return res;

}

}

9\_PalindromeNumber.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Determine whether an integer is a palindrome. Do this without extra space.

click to show spoilers.

Some hints:

Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer",

you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

Now the question is, how do we know that we've reached the half of the number?

Since we divided the number by 10, and multiplied the reversed number by 10, when the original

number is less than the reversed number, it means we've processed half of the number digits.

class Solution {

public boolean isPalindrome(int x) {

if (x < 0 || (x % 10 == 0 && x != 0 )){

return false;

}

int reverse = 0;

while (x > reverse){

reverse = reverse \* 10 + x % 10;

x /= 10;

}

return x == reverse || x == reverse / 10;

}

}

Better version:

class Solution {

public boolean isPalindrome(int x) {

if (x < 0){

return false;

}

int reverse = 0;

int orig = x;

while (x > 0){

reverse = reverse \* 10 + x%10;

x /= 10;

}

return reverse == orig;

}

}

90\_Subsets\_II.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a collection of integers that might contain duplicates, nums, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

For example,

If nums = [1,2,2], a solution is:

[

[2],

[1],

[1,2,2],

[2,2],

[1,2],

[]

]

class Solution {

public List<List<Integer>> subsetsWithDup(int[] nums) {

List<List<Integer>> result = new ArrayList<>();

Arrays.sort(nums);

boolean[] visited = new boolean[nums.length];

dfs(result, new ArrayList<>(), nums, 0, visited);

return result;

}

private void dfs(List<List<Integer>> result , List<Integer> item, int[] nums, int start, boolean[] visited){

result.add(new ArrayList<>(item));

for (int i = start; i < nums.length; i++){

if (i != 0 && nums[i-1] == nums[i] && !visited[i-1]){

continue;

}

item.add(nums[i]);

visited[i] = true;

dfs(result, item, nums, i+1, visited);

item.remove(item.size() - 1);

visited[i] = false;

}

}

}

class Solution {

public List<List<Integer>> subsetsWithDup(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

Arrays.sort(nums);

dfs(res, new ArrayList<Integer>(), nums, 0);

return res;

}

private void dfs(List<List<Integer>> res, List<Integer> item, int[] nums, int start){

res.add(new ArrayList<Integer>(item));

for (int i = start; i < nums.length; i++){

if (i > start && nums[i] == nums[i-1]){

continue;

}

item.add(nums[i]);

dfs(res, item, nums, i + 1);

item.remove(item.size() - 1);

}

}

}

900\_RLE.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write an iterator that iterates through a run-length encoded sequence.

The iterator is initialized by RLEIterator(int[] A), where A is a run-length encoding of some sequence. More specifically, for all even i, A[i] tells us the number of times that the non-negative integer value A[i+1] is repeated in the sequence.

The iterator supports one function: next(int n), which exhausts the next n elements (n >= 1) and returns the last element exhausted in this way. If there is no element left to exhaust, next returns -1 instead.

For example, we start with A = [3,8,0,9,2,5], which is a run-length encoding of the sequence [8,8,8,5,5]. This is because the sequence can be read as "three eights, zero nines, two fives".

Example 1:

Input: ["RLEIterator","next","next","next","next"], [[[3,8,0,9,2,5]],[2],[1],[1],[2]]

Output: [null,8,8,5,-1]

Explanation:

RLEIterator is initialized with RLEIterator([3,8,0,9,2,5]).

This maps to the sequence [8,8,8,5,5].

RLEIterator.next is then called 4 times:

.next(2) exhausts 2 terms of the sequence, returning 8. The remaining sequence is now [8, 5, 5].

.next(1) exhausts 1 term of the sequence, returning 8. The remaining sequence is now [5, 5].

.next(1) exhausts 1 term of the sequence, returning 5. The remaining sequence is now [5].

.next(2) exhausts 2 terms, returning -1. This is because the first term exhausted was 5,

but the second term did not exist. Since the last term exhausted does not exist, we return -1.

Note:

0 <= A.length <= 1000

A.length is an even integer.

0 <= A[i] <= 10^9

There are at most 1000 calls to RLEIterator.next(int n) per test case.

Each call to RLEIterator.next(int n) will have 1 <= n <= 10^9.

Method 1: Memory Limit Exceeded

class RLEIterator {

Queue<Integer> queue;

public RLEIterator(int[] A) {

queue = new LinkedList<>();

for (int i = 0; i < A.length; i += 2){

int dup = A[i];

int val = A[i+1];

for (int j = 0; j < dup; j++){

queue.offer(val);

}

}

}

public int next(int n) {

int size = queue.size();

if (size < n){

return -1;

}

for (int i = 0; i < n - 1; i++){

queue.poll();

}

return queue.poll();

}

}

/\*\*

\* Your RLEIterator object will be instantiated and called as such:

\* RLEIterator obj = new RLEIterator(A);

\* int param\_1 = obj.next(n);

\*/

Solution:

class RLEIterator {

int[] A;

int index;

public RLEIterator(int[] A) {

this.A = A;

index = 0;

}

public int next(int n) {

while (index < A.length && A[index] < n){

n -= A[index];

index += 2;

}

if (index >= A.length){

return -1;

}

A[index] -= n;

return A[index+1];

}

}

/\*\*

\* Your RLEIterator object will be instantiated and called as such:

\* RLEIterator obj = new RLEIterator(A);

\* int param\_1 = obj.next(n);

\*/

901\_Online.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a class StockSpanner which collects daily price quotes for some stock, and returns the span of that stock's price for the current

day.

The span of the stock's price today is defined as the maximum number of consecutive days (starting from today and going backwards)

for which the price of the stock was less than or equal to today's price.

For example, if the price of a stock over the next 7 days were [100, 80, 60, 70, 60, 75, 85], then the stock spans would be

[1, 1, 1, 2, 1, 4, 6].

Example 1:

Input: ["StockSpanner","next","next","next","next","next","next","next"], [[],[100],[80],[60],[70],[60],[75],[85]]

Output: [null,1,1,1,2,1,4,6]

Explanation:

First, S = StockSpanner() is initialized. Then:

S.next(100) is called and returns 1,

S.next(80) is called and returns 1,

S.next(60) is called and returns 1,

S.next(70) is called and returns 2,

S.next(60) is called and returns 1,

S.next(75) is called and returns 4,

S.next(85) is called and returns 6.

Note that (for example) S.next(75) returned 4, because the last 4 prices

(including today's price of 75) were less than or equal to today's price.

Note:

Calls to StockSpanner.next(int price) will have 1 <= price <= 10^5.

There will be at most 10000 calls to StockSpanner.next per test case.

There will be at most 150000 calls to StockSpanner.next across all test cases.

The total time limit for this problem has been reduced by 75% for C++, and 50% for all other languages.

Method 1: Brute Force TLE

Time compleixty of next: O(n)

class StockSpanner {

List<Integer> list;

public StockSpanner() {

list = new ArrayList<>();

}

public int next(int price) {

int count = 1;

int i = list.size() - 1;

while (i >= 0){

if (list.get(i) <= price){

count++;

}else{

break;

}

i--;

}

list.add(price);

return count;

}

}

Method 2: Monotonic decreasing stack

Time compleixty of next : O(1)

class StockSpanner {

Stack<int[]> stack;

public StockSpanner() {

stack = new Stack<>();

}

public int next(int price) {

if (stack.isEmpty()){

stack.push(new int[]{price, 1});

return 1;

}

int[] curr = stack.peek();

if (price < curr[0]){

stack.push(new int[]{price, 1});

return 1;

}else{

int val = 1;

while (!stack.isEmpty() && curr[0] <= price){

stack.pop();

val += curr[1];

if (!stack.isEmpty()){

curr = stack.peek();

}

}

stack.push(new int[]{price, val});

return val;

}

}

}

Best version:

class StockSpanner {

Stack<int[]> stack;

public StockSpanner() {

stack = new Stack<>();

}

public int next(int price) {

int val = 1;

while (!stack.isEmpty() && stack.peek()[0] <= price){

val += stack.pop()[1];

}

stack.push(new int[]{price, val});

return val;

}

}

903\_Valid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given S, a length n string of characters from the set {'D', 'I'}. (These letters stand for "decreasing" and "increasing".)

A valid permutation is a permutation P[0], P[1], ..., P[n] of integers {0, 1, ..., n}, such that for all i:

If S[i] == 'D', then P[i] > P[i+1], and;

If S[i] == 'I', then P[i] < P[i+1].

How many valid permutations are there? Since the answer may be large, return your answer modulo 10^9 + 7.

Example 1:

Input: "DID"

Output: 5

Explanation:

The 5 valid permutations of (0, 1, 2, 3) are:

(1, 0, 3, 2)

(2, 0, 3, 1)

(2, 1, 3, 0)

(3, 0, 2, 1)

(3, 1, 2, 0)

Note:

1 <= S.length <= 200

S consists only of characters from the set {'D', 'I'}.

https://leetcode.com/problems/valid-permutations-for-di-sequence/discuss/168278/C++JavaPython-DP-Solution-O(N2)?page=2

Method: DP

Time complexity: O(N^2)

dp[i][j] means the number of possible permutations of first i + 1 digits,

where the i + 1th digit is j + 1th smallest in the rest of digits.

class Solution {

public int numPermsDISequence(String S) {

int mod = (int)1e9 + 7;

int n = S.length();

int[][] dp = new int[n+1][n+1];

for (int j = 0; j <= n; j++){

dp[0][j] = 1;

}

for (int i = 1; i <= n; i++){

int cursum = 0; //use cursum to accumulate the sum to avoid one more dimensional iteration

if (S.charAt(i-1) == 'D'){

for (int j = n-i; j >= 0; j--){//backward

dp[i][j] = (cursum + dp[i-1][j+1]) % mod;

cursum = (cursum + dp[i-1][j+1]) % mod;

}

}else {//forward

for (int j = 0; j < n-i+1; j++){

dp[i][j] = (cursum + dp[i-1][j]) % mod;

cursum = (cursum + dp[i-1][j]) % mod;

}

}

}

return dp[n][0];

}

}

904\_Fruit.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a row of trees, the i-th tree produces fruit with type tree[i].

You start at any tree of your choice, then repeatedly perform the following steps:

Add one piece of fruit from this tree to your baskets. If you cannot, stop.

Move to the next tree to the right of the current tree. If there is no tree to the right, stop.

Note that you do not have any choice after the initial choice of starting tree: you must perform step 1, then step 2, then back

to step 1, then step 2, and so on until you stop.

You have two baskets, and each basket can carry any quantity of fruit, but you want each basket to only carry one type of fruit each.

What is the total amount of fruit you can collect with this procedure?

Example 1:

Input: [1,2,1]

Output: 3

Explanation: We can collect [1,2,1].

Example 2:

Input: [0,1,2,2]

Output: 3

Explanation: We can collect [1,2,2].

If we started at the first tree, we would only collect [0, 1].

Example 3:

Input: [1,2,3,2,2]

Output: 4

Explanation: We can collect [2,3,2,2].

If we started at the first tree, we would only collect [1, 2].

Example 4:

Input: [3,3,3,1,2,1,1,2,3,3,4]

Output: 5

Explanation: We can collect [1,2,1,1,2].

If we started at the first tree or the eighth tree, we would only collect 4 fruits.

Note:

1 <= tree.length <= 40000

0 <= tree[i] < tree.length

Method 1: Two Points

Time complexity: O(N)

Space complexity: O(1)

class Solution {

public int totalFruit(int[] tree) {

int N = tree.length;

int type1 = -1;

int type2 = -1;

int start1 = 0;

int start2 = 0;

int res = -1;

int i = 0;

while (i < N){

if (tree[i] != type1 && tree[i] != type2){

if (type1 == -1){

type1 = tree[i];

start1 = i;

}else if (type2 == -1){

type2 = tree[i];

start2 = i;

}else{

res = Math.max(res, i - start1);

type1 = type2;

start1 = start2;

i = start2;

type2 = -1;

}

}

i++;

}

return Math.max(res, i - start1);

}

}

Method 2: Longest Subarray With 2 Elements: Find out the longest length of subarrays with at most 2 different numbers

Sliding Window

Time complexity: O(N)

Space complexity: O(1)

class Solution {

public int totalFruit(int[] tree) {

int res = 0;

int a = -1;

int b = -1;

int count\_b = 0; // last type

int curr = 0; // current longest length

for (int c : tree){

if (c == a || c == b){

curr++;

}

if (c == b){

count\_b++;

}

if (c == a){

count\_b = 1;

a = b;

b = c;

}

if (c != a && c != b){

curr = count\_b + 1;

count\_b = 1;

a = b;

b = c;

}

res = Math.max(res, curr);

}

return res;

}

}

Method 3: Slidnig window with hashMap

class Solution {

public int totalFruit(int[] tree) {

Map<Integer, Integer> map = new HashMap<>();

int N = tree.length;

int res = 0;

int start = 0;

int end = 0;

while (end < N){

map.put(tree[end], map.getOrDefault(tree[end], 0) + 1);

while (map.size() > 2){

map.put(tree[start], map.get(tree[start]) - 1);

if (map.get(tree[start]) == 0){

map.remove(tree[start]);

}

start++;

}

res = Math.max(res, end - start + 1);

end++;

}

return res;

}

}

Best solution from template (sliding window)

Find the longest subarray with at most two different numbers

class Solution {

public int totalFruit(int[] tree) {

Map<Integer, Integer> map = new HashMap<>();

int res = 0;

int start = 0;

int end = 0;

int k = 2;

int count = 0;

while (end < tree.length){

map.put(tree[end], map.getOrDefault(tree[end], 0) + 1);

if (map.get(tree[end]) == 1){

count++;

}

end++;

while (count > k){

if (map.get(tree[start]) == 1){

count--;

}

map.put(tree[start], map.get(tree[start]) - 1);

start++;

}

res = Math.max(res, end - start);

}

return res;

}

}

905\_Sort.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of non-negative integers, return an array consisting of all the even elements of A, followed by all the odd elements of A.

You may return any answer array that satisfies this condition.

Example 1:

Input: [3,1,2,4]

Output: [2,4,3,1]

The outputs [4,2,3,1], [2,4,1,3], and [4,2,1,3] would also be accepted.

Note:

1 <= A.length <= 5000

0 <= A[i] <= 5000

Method 1:

Time complexity: O(N)

class Solution {

public int[] sortArrayByParity(int[] A) {

int N = A.length;

int[] res = new int[N];

int l = 0;

int r = A.length - 1;

for (int i = 0; i < A.length; i++){

if (A[i] % 2 == 0){

res[l++] = A[i];

}else{

res[r--] = A[i];

}

}

return res;

}

}

Method 2:

In place swap: O(N)

class Solution {

public int[] sortArrayByParity(int[] A) {

int l = 0;

int r = A.length - 1;

while (l < r){

if (A[l] % 2 == 0){

l++;

}else{

int temp = A[l];

A[l] = A[r];

A[r] = temp;

r--;

}

}

return A;

}

}

class Solution {

public int[] sortArrayByParity(int[] A) {

int i = 0;

int j = A.length - 1;

while (i < j){

while (i < j && A[i] % 2 == 0){

i++;

}

while (j > i && A[j] % 2 == 1){

j--;

}

if (i < j){

swap(A, i, j);

i++;

j--;

}

}

return A;

}

private void swap(int[] A, int i, int j){

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

907\_Sum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers A, find the sum of min(B), where B ranges over every (contiguous) subarray of A.

Since the answer may be large, return the answer modulo 10^9 + 7.

Example 1:

Input: [3,1,2,4]

Output: 17

Explanation: Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].

Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1. Sum is 17.

Note:

1 <= A.length <= 30000

1 <= A[i] <= 30000

Method 1: Brute Force (TLE)

Time complexity: O(N^2)

Space complexity: O(1)

class Solution {

public int sumSubarrayMins(int[] A) {

int mod = (int)1e9 + 7;

int n = A.length;

int res = 0;

for (int i = 0; i < n; i++){

int min = A[i];

for (int j = i; j < n; j++){

min = Math.min(min, A[j]);

res = (res + min) % mod;

}

}

return res;

}

}

Method 2: DP + Monotonic stack (store index but value in ascending)

Time complexity: O(N)

Space complexity: O(N)

https://leetcode.com/problems/sum-of-subarray-minimums/discuss/170769/Java-O(n)-monotone-stack-with-DP

class Solution {

public int sumSubarrayMins(int[] A) {

int mod = (int)1e9 + 7;

int n = A.length;

int res = 0;

Stack<Integer> stack = new Stack<>();//monotonic stack and store index, but the value of index in ascending

int[] dp = new int[n+1]; //dp[i] indicates the sum of subarray minimum ending at i-1

stack.push(0); //dummy index is set to 0

for (int i = 1; i <= n; i++){

while (stack.peek() != 0 && A[stack.peek()-1] >= A[i-1]){

stack.pop();

}

dp[i] = (dp[stack.peek()] + (i - stack.peek()) \* A[i-1]) % mod;

stack.push(i);//stack stores the index + 1

res = (res + dp[i]) % mod;

}

return res;

}

}

Monotonic Stack Template

https://leetcode.com/problems/sum-of-subarray-minimums/discuss/178876/stack-solution-with-very-detailed-explanation-step-by-step

Method 3: Monotonic Stack

Best solution

class Solution {

public int sumSubarrayMins(int[] A) {

int mod = (int)1e9 + 7;

int n = A.length;

// left is for the distance to previous less element

// right is for the distance to next less element

int[] left = new int[n];

int[] right = new int[n];

for(int i = 0; i < A.length; i++) {

left[i] = i + 1;

right[i] = n - i;

}

Stack<Integer> stack = new Stack<>();

// previous less element

for (int i = 0; i < n; i++){

while (!stack.isEmpty() && A[stack.peek()] > A[i]){

stack.pop();

}

left[i] = stack.isEmpty() ? i + 1 : i - stack.peek();

stack.push(i);

}

//next less element

stack = new Stack<>();

for (int i = 0; i < n; i++){

while (!stack.isEmpty() && A[stack.peek()] > A[i]){

int index = stack.pop();

right[index] = i - index;

}

stack.push(i);

}

int res = 0;

for (int i = 0; i < n; i++){

res = (res + A[i] \* left[i] \* right[i]) % mod;

}

return res;

}

}

Best of Best:

Similar as Leetcode 901 Online Stock Span

class Solution {

public int sumSubarrayMins(int[] A) {

int res = 0, n = A.length, mod = (int)1e9 + 7;

int[] left = new int[n], right = new int[n];

Stack<int[]> s1 = new Stack<>(), s2 = new Stack<>();

for (int i = 0; i < n; ++i) {

int count = 1;

while (!s1.isEmpty() && s1.peek()[0] > A[i])

count += s1.pop()[1];

s1.push(new int[] {A[i], count});

left[i] = count;

}

for (int i = n - 1; i >= 0; --i) {

int count = 1;

while (!s2.isEmpty() && s2.peek()[0] >= A[i]) // use == to handle duplicate elements, only count one time either in left or right

count += s2.pop()[1];

s2.push(new int[] {A[i], count});

right[i] = count;

}

for (int i = 0; i < n; ++i)

res = (res + A[i] \* left[i] \* right[i]) % mod;

return res;

}

}

908\_Smallest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of integers, for each integer A[i] we may choose any x with -K <= x <= K, and add x to A[i].

After this process, we have some array B.

Return the smallest possible difference between the maximum value of B and the minimum value of B.

Example 1:

Input: A = [1], K = 0

Output: 0

Explanation: B = [1]

Example 2:

Input: A = [0,10], K = 2

Output: 6

Explanation: B = [2,8]

Example 3:

Input: A = [1,3,6], K = 3

Output: 0

Explanation: B = [3,3,3] or B = [4,4,4]

Note:

1 <= A.length <= 10000

0 <= A[i] <= 10000

0 <= K <= 10000

class Solution {

public int smallestRangeI(int[] A, int K) {

int min = A[0];

int max = A[0];

for (int i : A){

min = Math.min(min, i);

max = Math.max(max, i);

}

return Math.max(0, max - min - 2 \* K);

}

}

909\_Snakes.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On an N x N board, the numbers from 1 to N\*N are written boustrophedonically starting from the bottom left of the board, and alternating direction each row. For example, for a 6 x 6 board, the numbers are written as follows:

You start on square 1 of the board (which is always in the last row and first column). Each move, starting from square x, consists of the following:

You choose a destination square S with number x+1, x+2, x+3, x+4, x+5, or x+6, provided this number is <= N\*N.

(This choice simulates the result of a standard 6-sided die roll: ie., there are always at most 6 destinations.)

If S has a snake or ladder, you move to the destination of that snake or ladder. Otherwise, you move to S.

A board square on row r and column c has a "snake or ladder" if board[r][c] != -1. The destination of that snake or ladder is board[r][c].

Note that you only take a snake or ladder at most once per move: if the destination to a snake or ladder is the start of another snake or ladder, you do not continue moving. (For example, if the board is `[[4,-1],[-1,3]]`, and on the first move your destination square is `2`, then you finish your first move at `3`, because you do not continue moving to `4`.)

Return the least number of moves required to reach square N\*N. If it is not possible, return -1.

Example 1:

Input: [

[-1,-1,-1,-1,-1,-1],

[-1,-1,-1,-1,-1,-1],

[-1,-1,-1,-1,-1,-1],

[-1,35,-1,-1,13,-1],

[-1,-1,-1,-1,-1,-1],

[-1,15,-1,-1,-1,-1]]

Output: 4

Explanation:

At the beginning, you start at square 1 [at row 5, column 0].

You decide to move to square 2, and must take the ladder to square 15.

You then decide to move to square 17 (row 3, column 5), and must take the snake to square 13.

You then decide to move to square 14, and must take the ladder to square 35.

You then decide to move to square 36, ending the game.

It can be shown that you need at least 4 moves to reach the N\*N-th square, so the answer is 4.

Note:

2 <= board.length = board[0].length <= 20

board[i][j] is between 1 and N\*N or is equal to -1.

The board square with number 1 has no snake or ladder.

The board square with number N\*N has no snake or ladder.

https://leetcode.com/problems/snakes-and-ladders/discuss/173403/Change-to-1D-array-then-BFS

Change to 1D array and then BFS

class Solution {

public int snakesAndLadders(int[][] board) {

int n = board.length;

int[] arr = new int[n\*n];

int index = 0;

int i = n-1;

int j = 0;

int inc = 1;

while (index < n \* n){

arr[index++] = board[i][j];

if (j == n - 1 && inc == 1){

i--;

inc = -1;

}else if (j == 0 && inc == -1){

i--;

inc = 1;

}else{

j += inc;

}

}

Queue<Integer> queue = new LinkedList<>();//store index

boolean[] visited = new boolean[n\*n];

int start = arr[0] > -1 ? arr[0] - 1: 0;

queue.offer(start);

visited[start] = true;

int step = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int l = 0; l < size; l++){

int curr = queue.poll();

if (curr == n\*n - 1){

return step;

}

for (int k = curr + 1; k <= Math.min(curr+6, n\*n-1); k++){

int next = arr[k] > -1 ? arr[k] - 1 : k;

if (!visited[next]){

queue.offer(next);

visited[next] = true;

}

}

}

step++;

}

return -1;

}

}

91\_Decode.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' -> 1

'B' -> 2

...

'Z' -> 26

Given a non-empty string containing only digits, determine the total number of ways to decode it.

Example 1:

Input: "12"

Output: 2

Explanation: It could be decoded as "AB" (1 2) or "L" (12).

Example 2:

Input: "226"

Output: 3

Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

class Solution {

public int numDecodings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int n = s.length();

int[] dp = new int[n + 1];

dp[0] = 1; //can't be 0

dp[1] = s.charAt(0) == '0' ? 0 : 1;

for (int i = 2; i <= n; i++){

if (s.charAt(i-1) != '0'){

dp[i] = dp[i-1];

}

int twoDigit = (s.charAt(i-2) - '0') \* 10 + (s.charAt(i-1) - '0');

if (twoDigit <= 26 && twoDigit >= 10){

dp[i] += dp[i-2];

}

}

return dp[n];

}

}

Best solution:

class Solution {

public int numDecodings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int n = s.length();

if (n == 1){

if (s.charAt(0) != '0'){

return 1;

}else{

return 0;

}

}

if (s.charAt(0) == '0'){

return 0;

}

int[] dp = new int[n+1];

dp[0] = 1;

dp[1] = 1;

for (int i = 2; i <= n; i++){

int curr = (int)(s.charAt(i-1) - '0');

int prev = (int)(s.charAt(i-2) - '0');

if (curr != 0){

dp[i] += dp[i-1];

}

int num = prev \* 10 + curr;

if (num >= 10 && num <= 26){

dp[i] += dp[i-2];

}

}

return dp[n];

}

}

91\_DecodeWays.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' -> 1

'B' -> 2

...

'Z' -> 26

Given an encoded message containing digits, determine the total number of ways to decode it.

For example,

Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).

The number of ways decoding "12" is 2.

class Solution {

public int numDecodings(String s) {

if (s == null || s.length() == 0){

return 0;

}

int[] f = new int[s.length() + 1];

f[0] = 1;

if (s.charAt(0) == '0'){

f[1] = 0;

}else{

f[1] = 1;

}

for (int i = 2; i <= s.length(); i++){

char s1 = s.charAt(i-1);

char s2 = s.charAt(i-2);

if (s1 != '0' ){

f[i] += f[i-1];

}

int v = s1 -'0' + (s2 - '0') \*10; // convert char to integer

if (10 <= v && v <= 26){

f[i] += f[i-2];

}

}

return f[s.length()];

}

}

910\_Smallest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of integers, for each integer A[i] we need to choose either x = -K or x = K, and add x to A[i] (only once).

After this process, we have some array B.

Return the smallest possible difference between the maximum value of B and the minimum value of B.

Example 1:

Input: A = [1], K = 0

Output: 0

Explanation: B = [1]

Example 2:

Input: A = [0,10], K = 2

Output: 6

Explanation: B = [2,8]

Example 3:

Input: A = [1,3,6], K = 3

Output: 3

Explanation: B = [4,6,3]

Note:

1 <= A.length <= 10000

0 <= A[i] <= 10000

0 <= K <= 10000

Method 1:

The same as

Add 0 or 2\*K

class Solution {

public int smallestRangeII(int[] A, int K) {

Arrays.sort(A);

int n = A.length;

int max = A[n-1];

int min = A[0];

int res = A[n-1] - A[0];

for (int i = 0; i < n - 1; i++){

max = Math.max(max, A[i] + 2 \* K);

min = Math.min(A[i+1], A[0] + 2 \* K);

res = Math.min(res, max - min);

}

return res;

}

}

911\_Online.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In an election, the i-th vote was cast for persons[i] at time times[i].

Now, we would like to implement the following query function: TopVotedCandidate.q(int t) will return the number of the person that was leading the election at time t.

Votes cast at time t will count towards our query. In the case of a tie, the most recent vote (among tied candidates) wins.

Example 1:

Input: ["TopVotedCandidate","q","q","q","q","q","q"], [[[0,1,1,0,0,1,0],[0,5,10,15,20,25,30]],[3],[12],[25],[15],[24],[8]]

Output: [null,0,1,1,0,0,1]

Explanation:

At time 3, the votes are [0], and 0 is leading.

At time 12, the votes are [0,1,1], and 1 is leading.

At time 25, the votes are [0,1,1,0,0,1], and 1 is leading (as ties go to the most recent vote.)

This continues for 3 more queries at time 15, 24, and 8.

Note:

1 <= persons.length = times.length <= 5000

0 <= persons[i] <= persons.length

times is a strictly increasing array with all elements in [0, 10^9].

TopVotedCandidate.q is called at most 10000 times per test case.

TopVotedCandidate.q(int t) is always called with t >= times[0].

class TopVotedCandidate {

int[] times;

Map<Integer, Integer> map;

public TopVotedCandidate(int[] persons, int[] times) {

this.times = times;

map = new HashMap<>();

int[] hash = new int[persons.length+1];

int cand = 0;

for (int i = 0; i < times.length; i++){

int curr = persons[i];

hash[curr]++;

if (hash[curr] >= hash[cand]){

cand = curr;

}

map.put(times[i], cand);

}

}

public int q(int t) {//binary search

int start = 0;

int end = times.length - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (times[mid] == t){

return map.get(times[mid]);

}else if (times[mid] > t){

end = mid;

}else{

start = mid;

}

}

if (times[end] <= t){

return map.get(times[end]);

}

return map.get(times[start]);

}

}

class TopVotedCandidate {

int[] times;

Map<Integer, Integer> map;

public TopVotedCandidate(int[] persons, int[] times) {

this.times = times;

map = new HashMap<>();

int[] hash = new int[persons.length+1];

int cand = 0;

for (int i = 0; i < times.length; i++){

int curr = persons[i];

hash[curr]++;

if (hash[curr] >= hash[cand]){

cand = curr;

}

map.put(times[i], cand);

}

}

public int q(int t) {//binary search

int i = Arrays.binarySearch(times, t);

return i < 0 ? map.get(times[-i-2]) : map.get(times[i]);

}

}

class TopVotedCandidate {

TreeMap<Integer, Integer> map;

public TopVotedCandidate(int[] persons, int[] times) {

map = new TreeMap<>();

Map<Integer, Integer> count = new HashMap<>();

int cand = 0;

for (int i = 0; i < persons.length; i++){

int time = times[i];

int person = persons[i];

count.put(person, count.getOrDefault(person, 0) + 1);

if (count.get(person) >= count.get(cand)){

cand = person;

}

map.put(time, cand);

}

}

public int q(int t) {

Integer key = map.floorKey(t);

return map.get(key);

}

}

914\_gcd.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a deck of cards, each card has an integer written on it.

Return true if and only if you can choose X >= 2 such that it is possible to split the entire deck into 1 or more groups of cards, where:

Each group has exactly X cards.

All the cards in each group have the same integer.

Example 1:

Input: [1,2,3,4,4,3,2,1]

Output: true

Explanation: Possible partition [1,1],[2,2],[3,3],[4,4]

Example 2:

Input: [1,1,1,2,2,2,3,3]

Output: false

Explanation: No possible partition.

Example 3:

Input: [1]

Output: false

Explanation: No possible partition.

Example 4:

Input: [1,1]

Output: true

Explanation: Possible partition [1,1]

Example 5:

Input: [1,1,2,2,2,2]

Output: true

Explanation: Possible partition [1,1],[2,2],[2,2]

Note:

1 <= deck.length <= 10000

0 <= deck[i] < 10000

class Solution {

public boolean hasGroupsSizeX(int[] deck) {

Map<Integer, Integer> map = new HashMap<>();

for (int i : deck){

map.put(i, map.getOrDefault(i, 0) + 1);

}

int res = 0;

for (int val : map.values()){

res = gcd(res, val);

}

return res > 1;

}

private int gcd(int a, int b){

return b == 0 ? a : gcd(b, a%b);

}

}

How to calculate greatest common divisor

private int gcd (int a, int b){

return b == 0 ? a : gcd (b, a%b);

}

915\_Partition.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A, partition it into two (contiguous) subarrays left and right so that:

Every element in left is less than or equal to every element in right.

left and right are non-empty.

left has the smallest possible size.

Return the length of left after such a partitioning. It is guaranteed that such a partitioning exists.

Example 1:

Input: [5,0,3,8,6]

Output: 3

Explanation: left = [5,0,3], right = [8,6]

Example 2:

Input: [1,1,1,0,6,12]

Output: 4

Explanation: left = [1,1,1,0], right = [6,12]

Note:

2 <= A.length <= 30000

0 <= A[i] <= 10^6

It is guaranteed there is at least one way to partition A as described.

Method 1: Brute Force

Time complexity: O(n^2)

Space complexity: O(1)

class Solution {

public int partitionDisjoint(int[] A) {

int n = A.length;

for (int i = 1; i < n; i++){

int leftMax = Integer.MIN\_VALUE;

int rightMin = Integer.MAX\_VALUE;

for (int j = 0; j < i; j++){

leftMax = Math.max(leftMax, A[j]);

}

for (int j = i; j < n; j++){

rightMin = Math.min(rightMin, A[j]);

}

if (leftMax <= rightMin){

return i;

}

}

return -1;

}

}

Method 2: Best

https://leetcode.com/problems/partition-array-into-disjoint-intervals/discuss/175945/Java-one-pass-7-lines

suppose the original left subarray is from 0 to cand, the max value of that is localMax. If it is a valid partition,

every value from partitionIdx + 1 to end should be >= localMax. But if we find a value in the right part, a[i],

is smaller than localMax, which means the partition is not correct and we have to incorporate a[i] to form the left subarray.

So the partitionIdx is set to be i, and we have to recalculate the max value of the new left subarray.(recorded in max)

Time complexity: O(n)

Space complexity: O(1)

class Solution {

public int partitionDisjoint(int[] A) {

int leftMax = A[0];

int cand = 0;

int max = leftMax;

for (int i = 1; i < A.length; i++){

if (leftMax > A[i]){

leftMax = max;

cand = i;

}else{

max = Math.max(max, A[i]);

}

}

return cand + 1;

}

}

Better version:

class Solution {

public int partitionDisjoint(int[] A) {

int cand = 0;

int maxToCand = A[0];

int maxToI = A[0];

for (int i = 1; i < A.length; i++){

maxToI = Math.max(maxToI, A[i]);

if (A[i] < maxToCand){

cand = i;

maxToCand = maxToI;

}

}

return cand + 1;

}

}

Method 3:

Time complexity: O(n)

Space complexity: O(n)

https://leetcode.com/problems/partition-array-into-disjoint-intervals/discuss/175849/C++JavaPython-Straight-Forward

Use minFromBack to store min value from i to end and then check from front

class Solution {

public int partitionDisjoint(int[] A) {

int n = A.length;

int[] minFromBack = new int[n];

minFromBack[n-1] = A[n-1];

for (int i = n - 2; i > 0; i--){

minFromBack[i] = Math.min(minFromBack[i+1], A[i]);

}

int leftMax = 0;

for (int i = 1; i < n; i++){

leftMax = Math.max(leftMax, A[i-1]);

if (leftMax <= minFromBack[i]){

return i;

}

}

return -1;

}

}

Best solution: the same as Leetcode 768 Max Chunks To Make Sorted II

https://github.com/optimisea/Leetcode/blob/master/Java/768\_Max.java

class Solution {

public int partitionDisjoint(int[] A) {

int n = A.length;

int[] left = new int[n+1];

int[] right = new int[n+1];

left[0] = Integer.MIN\_VALUE;

right[n] = Integer.MAX\_VALUE;

for (int i =1 ; i <= n; i++){

left[i] = Math.max(left[i-1], A[i-1]);

}

for (int i = n-1; i>= 0; i--){

right[i] = Math.min(right[i+1], A[i]);

}

for (int i = 1; i < n; i++){

if (left[i] <= right[i]){

return i;

}

}

return -1;

}

}

916\_Word.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given two arrays A and B of words. Each word is a string of lowercase letters.

Now, say that word b is a subset of word a if every letter in b occurs in a, including multiplicity. For example, "wrr" is a

subset of "warrior", but is not a subset of "world".

Now say a word a from A is universal if for every b in B, b is a subset of a.

Return a list of all universal words in A. You can return the words in any order.

Example 1:

Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["e","o"]

Output: ["facebook","google","leetcode"]

Example 2:

Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["l","e"]

Output: ["apple","google","leetcode"]

Example 3:

Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["e","oo"]

Output: ["facebook","google"]

Example 4:

Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["lo","eo"]

Output: ["google","leetcode"]

Example 5:

Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["ec","oc","ceo"]

Output: ["facebook","leetcode"]

Note:

1 <= A.length, B.length <= 10000

1 <= A[i].length, B[i].length <= 10

A[i] and B[i] consist only of lowercase letters.

All words in A[i] are unique: there isn't i != j with A[i] == A[j].

Method 1: Brute Force TLE

Time complexity: O(A\*B)

Space complexity: O(1)

class Solution {

public List<String> wordSubsets(String[] A, String[] B) {

List<String> res = new ArrayList<>();

for (String a : A){

boolean isUniversal = true;

int[] hash = new int[26];

for (char c : a.toCharArray()){

hash[c - 'a']++;

}

for (String b : B){

int[] temp = new int[26];

for (int i = 0; i < 26; i++){

temp[i] = hash[i];

}

for (char c : b.toCharArray()){

temp[c - 'a']--;

if (temp[c - 'a'] < 0){

isUniversal = false;

break;

}

}

if (!isUniversal){

break;

}

}

if (isUniversal){

res.add(a);

}

}

return res;

}

}

Similar as Leetcode 1002 Find Common Character

https://github.com/optimisea/Leetcode/blob/master/Java/1002\_Find.java

Method 2: Best solution

Time complexity: O(A + B)

Space complexity: O(1)

class Solution {

public List<String> wordSubsets(String[] A, String[] B) {

List<String> res = new ArrayList<>();

int[] hash = new int[26];

for (String b : B){

int[] temp = new int[26];

for (char c : b.toCharArray()){

temp[c - 'a']++;

}

for (int i = 0; i < 26; i++){

hash[i] = Math.max(hash[i], temp[i]); // here is the trick, get the max number of each character

}

}

for (String a : A){

boolean isUniversal = true;

int[] temp = new int[26];

for (char c : a.toCharArray()){

temp[c - 'a']++;

}

for (int i = 0; i < 26; i++){

if (temp[i] < hash[i]){

isUniversal = false;

break;

}

}

if (isUniversal){

res.add(a);

}

}

return res;

}

}

917\_Reverse.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S, return the "reversed" string where all characters that are not a letter stay in the same place, and all letters reverse their positions.

Example 1:

Input: "ab-cd"

Output: "dc-ba"

Example 2:

Input: "a-bC-dEf-ghIj"

Output: "j-Ih-gfE-dCba"

Example 3:

Input: "Test1ng-Leet=code-Q!"

Output: "Qedo1ct-eeLg=ntse-T!"

Note:

S.length <= 100

33 <= S[i].ASCIIcode <= 122

S doesn't contain \ or "

Method 1: Two pointers

Time complexity: O(n)

Space complexity: O(n)

class Solution {

public String reverseOnlyLetters(String S) {

char[] chars = S.toCharArray();

int left = 0;

int right = chars.length - 1;

while (left < right){

while (left < right && !Character.isLetter(chars[left])){

left++;

}

while (left < right && !Character.isLetter(chars[right])){

right--;

}

char temp = chars[left];

chars[left] = chars[right];

chars[right] = temp;

left++;

right--;

}

return new String(chars);

}

}

Method 2:

class Solution {

public String reverseOnlyLetters(String S) {

StringBuilder sb = new StringBuilder(S);

int left = 0;

int right = S.length() - 1;

while (left < right){

while (left < right && !Character.isLetter(sb.charAt(left))){

left++;

}

while (left < right && !Character.isLetter(sb.charAt(right))){

right--;

}

sb.setCharAt(left, S.charAt(right));

sb.setCharAt(right, S.charAt(left));

left++;

right--;

}

return sb.toString();

}

}

918\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a circular array C of integers represented by A, find the maximum possible sum of a non-empty subarray of C.

Here, a circular array means the end of the array connects to the beginning of the array.

(Formally, C[i] = A[i] when 0 <= i < A.length, and C[i+A.length] = C[i] when i >= 0.)

Also, a subarray may only include each element of the fixed buffer A at most once.

(Formally, for a subarray C[i], C[i+1], ..., C[j], there does not exist i <= k1, k2 <= j with k1 % A.length = k2 % A.length.)

Example 1:

Input: [1,-2,3,-2]

Output: 3

Explanation: Subarray [3] has maximum sum 3

Example 2:

Input: [5,-3,5]

Output: 10

Explanation: Subarray [5,5] has maximum sum 5 + 5 = 10

Example 3:

Input: [3,-1,2,-1]

Output: 4

Explanation: Subarray [2,-1,3] has maximum sum 2 + (-1) + 3 = 4

Example 4:

Input: [3,-2,2,-3]

Output: 3

Explanation: Subarray [3] and [3,-2,2] both have maximum sum 3

Example 5:

Input: [-2,-3,-1]

Output: -1

Explanation: Subarray [-1] has maximum sum -1

Note:

-30000 <= A[i] <= 30000

1 <= A.length <= 30000

https://www.jianshu.com/p/e09a051f5420

那么碰到循环数组的问题，无外乎三个套路。

拆分（HOUSE ROBBER那道，循环道路时用的是这个技巧）

倍增（就是在原数组后补到2倍长度，在2倍长数组的里处理，那么原本的N长的数组，我们可以变成N个新的长度为N的数组）

0->N-1， 1- >N， 2- >N+1 ..... N-1 -> 2\*n-2

分别对每种循环的可能做处理，最后汇总得到解，这题可以运用这个策略，但是时间复杂度为N^2

取反（求最大，变成求最小）

在这道题里的运用，就是如果是2端加起来最大。那么必定我们需要在中间找一个最小。然后用SUM去减掉中间最小。就得到2端最大。

Method 1: 倍增

Time complexity: O(N^2)

Space Complexity: O(N)

class Solution {

public int maxSubarraySumCircular(int[] A) {

int n = A.length;

int[] B = new int[2\*n];

for (int i = 0; i < 2\*n; i++){

B[i] = A[i%n];

}

int res = Integer.MIN\_VALUE;

int[] preSum = new int[2\*n+1];

preSum[0] = 0;

for (int i = 1; i <= 2\*n; i++){

preSum[i] = preSum[i-1] + B[i-1];

}

for (int i = 0; i <= 2\*n; i++){

for (int j = 0; j < i; j++){

if (i - j <= n){

res = Math.max(res, preSum[i] - preSum[j]);

}

}

}

return res;

}

}

Method 2: 取反

class Solution {

public int maxSubarraySumCircular(int[] A) {

//two cases

//case 1: max exits in the middle, standard greedy method to find max

int max = Integer.MIN\_VALUE;

int sum = 0;

for (int i : A){

sum += i;

max = Math.max(max, sum);

if (sum < 0){

sum = 0;

}

}

// case 2: max exits at two sides, standard greed method to find min and sum - max

int min = Integer.MAX\_VALUE;

sum = 0;

for (int i : A){

sum += i;

min = Math.min(min, sum);

if (sum > 0){

sum = 0;

}

}

sum = 0;

for (int i : A){

sum += i;

}

//Corner case

//if max is negative, it indicates that all the numbers are negative.

if (max < 0){

return max;

}

return Math.max(max, sum - min);

}

}

Corner case:

If all number are negative,

return the maximum one,

(which equals to the max subarray sum)

Best solution: O(N)

class Solution {

public int maxSubarraySumCircular(int[] A) {

int max = Integer.MIN\_VALUE;

int min = Integer.MAX\_VALUE;

int sum = 0;

int localMax = 0;

int localMin = 0;

for (int i : A){

sum += i;

localMax = Math.max(i, localMax + i);

max = Math.max(max, localMax);

localMin = Math.min(i, localMin + i);

min = Math.min(min, localMin);

}

if (max < 0){

return max;

}

return Math.max(max, sum - min);

}

}

Side note: Refer to leetcode 53

If it is not circular array,

https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/

Largest Sum Contiguous Subarray

Method 1: greedy

int maxSubArraySum(int[] A){

int max = Integer.MIN\_VALUE;

int sum = 0;

for (int i : A){

sum += i;

max = Math.max(max, sum);

if (sum < 0){

sum = 0;

}

}

return max;

}

Method 2: DP

class Solution {

public int maxSubArray(int[] nums) {

int local = 0;

int global = Integer.MIN\_VALUE;

for (int num : nums){

local = Math.max(local + num, num);

global = Math.max(global, local);

}

return global;

}

}

919\_Complete.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as

far left as possible.

Write a data structure CBTInserter that is initialized with a complete binary tree and supports the following operations:

CBTInserter(TreeNode root) initializes the data structure on a given tree with head node root;

CBTInserter.insert(int v) will insert a TreeNode into the tree with value node.val = v so that the tree remains complete, and

returns the value of the parent of the inserted TreeNode;

CBTInserter.get\_root() will return the head node of the tree.

Example 1:

Input: inputs = ["CBTInserter","insert","get\_root"], inputs = [[[1]],[2],[]]

Output: [null,1,[1,2]]

Example 2:

Input: inputs = ["CBTInserter","insert","insert","get\_root"], inputs = [[[1,2,3,4,5,6]],[7],[8],[]]

Output: [null,3,4,[1,2,3,4,5,6,7,8]]

Note:

The initial given tree is complete and contains between 1 and 1000 nodes.

CBTInserter.insert is called at most 10000 times per test case.

Every value of a given or inserted node is between 0 and 5000.

Key idea: Complete Binary Tree ==> HashMap + Label Node

class CBTInserter {

Map<Integer, TreeNode> map;

int count;

TreeNode head;

public CBTInserter(TreeNode root) {

head = root;

map = new HashMap<>();

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(head);

count = 1;

map.put(count++, head);

while (!queue.isEmpty()){

TreeNode node = queue.poll();

if (node.left != null){

map.put(count++, node.left);

queue.offer(node.left);

if (node.right != null){

map.put(count++, node.right);

queue.offer(node.right);

}

}

}

}

public int insert(int v) {

int parentInt = count / 2;

TreeNode parent = map.get(parentInt);

TreeNode node = new TreeNode(v);

if (count % 2 == 0){//insert as left node

parent.left = node;

}else{//insert as right node

parent.right = node;

}

map.put(count++, node);

return parent.val;

}

public TreeNode get\_root() {

return head;

}

}

92\_Backpack.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n items with size Ai, an integer m denotes the size of a backpack. How full you can fill this backpack?

Notice

You can not divide any item into small pieces.

Have you met this question in a real interview?

Example

If we have 4 items with size [2, 3, 5, 7], the backpack size is 11, we can select [2, 3, 5], so that the max size

we can fill this backpack is 10. If the backpack size is 12. we can select [2, 3, 7] so that we can fulfill the backpack.

You function should return the max size we can fill in the given backpack.

Challenge

O(n x m) time and O(m) memory.

O(n x m) memory is also acceptable if you do not know how to optimize memory.

Method 1:非滚动

Time complexity: O(n\*m)

Space complexity: O(n\*m)

public class Solution {

/\*\*

\* @param m: An integer m denotes the size of a backpack

\* @param A: Given n items with size A[i]

\* @return: The maximum size

\*/

public int backPack(int m, int[] A) {

int n = A.length;

boolean[][] dp = new boolean[n+1][m+1];

for (int i = 0; i <= n; i++){

dp[i][0] = true;

}

for (int j = 1; j <= m; j++){

dp[0][j] = false;

}

for (int i = 1; i <= n; i++){

for (int j = 1; j <= m; j++){

if (j >= A[i-1]){

dp[i][j] = dp[i-1][j] || dp[i-1][j - A[i-1]];

}else{

dp[i][j] = dp[i-1][j];

}

}

}

for (int i = m; i >= 0; i--){

if (dp[n][i]){

return i;

}

}

return 0;

}

}

Method 2:

Method 1:滚动数组优化

Time complexity: O(n\*m)

Space complexity: O(m)

State: f[i][S] “前i”个物品,取出一些能否组成和为S

Function: a[i-1] 是第i个物品下标是i-1

f[i][S] = f[i-1][S - a[i-1]] or f[i-1][S]

Intialize: f[i][0] = true; f[0][1..target] = false

Answer: 检查所有的f[n][j]

public class Solution {

/\*\*

\* @param m: An integer m denotes the size of a backpack

\* @param A: Given n items with size A[i]

\* @return: The maximum size

\*/

public int backPack(int m, int[] A) {

int n = A.length;

boolean[][] dp = new boolean[2][m+1];

for (int i = 0; i <= 1; i++){

dp[i][0] = true;

}

for (int j = 1; j <= m; j++){

dp[0][j] = false;

}

for (int i = 1; i <= n; i++){

for (int j = 1; j <= m; j++){

if (j >= A[i-1]){

dp[i%2][j] = dp[(i-1)%2][j] || dp[(i-1)%2][j - A[i-1]];

}else{

dp[i%2][j] = dp[(i-1)%2][j];

}

}

}

for (int i = m; i >= 0; i--){

if (dp[n%2][i]){

return i;

}

}

return 0;

}

}

92\_Reverse.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Reverse a linked list from position m to n. Do it in one-pass.

Note: 1 ≤ m ≤ n ≤ length of list.

Example:

Input: 1->2->3->4->5->NULL, m = 2, n = 4

Output: 1->4->3->2->5->NULL

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode reverseBetween(ListNode head, int m, int n) {

if (head == null || head.next == null){

return head;

}

ListNode dummy = new ListNode(-1);

ListNode node = dummy;

dummy.next = head;

int i = 0;

while (node != null){

if (i == m - 1){

break;

}

i++;

node = node.next;

}

ListNode first = node;

node = node.next;

ListNode last = node;

ListNode prev = null;

while (node != null){

ListNode temp = node.next;

node.next = prev;

prev = node;

node = temp;

i++;

if (i == n){

break;

}

}

first.next = prev;

last.next = node;

return dummy.next;

}

}

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode reverseBetween(ListNode head, int m, int n) {

if (head == null || head.next == null){

return head;

}

ListNode dummy = new ListNode(-1);

ListNode node = dummy;

dummy.next = head;

int i = 0;

while (node != null){

if (i == m - 1){

break;

}

i++;

node = node.next;

}

ListNode first = node;

node = node.next;

i++;

ListNode last = node;

ListNode prev = null;

while (node != null){

ListNode temp = node.next;

node.next = prev;

prev = node;

node = temp;

i++;

if (i == n+1){

break;

}

}

first.next = prev;

last.next = node;

return dummy.next;

}

}

92\_ReverseLinkedListII.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example:

Given 1->2->3->4->5->NULL, m = 2 and n = 4,

return 1->4->3->2->5->NULL.

Note:

Given m, n satisfy the following condition:

1 ≤ m ≤ n ≤ length of list.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

Method:

since the first node could be reversed, dummy is needed and then start from 0.

class Solution {

public ListNode reverseBetween(ListNode head, int m, int n) {

if (head == null || head.next == null){

return head;

}

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode cur = dummy;

for (int i = 0; i < m - 1; i++){

cur = cur.next;

}

ListNode start = cur;

ListNode tail = cur.next;

int i = m;

ListNode prev = null;

cur = cur.next;

while (cur != null && i <= n){

ListNode temp = cur.next;

cur.next = prev;

prev = cur;

cur = temp;

i++;

}

start.next = prev;

tail.next = cur;

return dummy.next;

}

}

920\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Your music player contains N different songs and she wants to listen to L (not necessarily different) songs during your trip.

You create a playlist so that:

Every song is played at least once

A song can only be played again only if K other songs have been played

Return the number of possible playlists. As the answer can be very large, return it modulo 10^9 + 7.

Example 1:

Input: N = 3, L = 3, K = 1

Output: 6

Explanation: There are 6 possible playlists. [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].

Example 2:

Input: N = 2, L = 3, K = 0

Output: 6

Explanation: There are 6 possible playlists. [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2, 1], [2, 1, 2], [1, 2, 2]

Example 3:

Input: N = 2, L = 3, K = 1

Output: 2

Explanation: There are 2 possible playlists. [1, 2, 1], [2, 1, 2]

Note:

0 <= K < N <= L <= 100

dp[i][j] denotes the solution of i songs with j different songs. So the final answer should be dp[L][N]

Think one step before the last one, there are only cases for the answer of dp[i][j]

case 1 (the last added one is new song): listen i - 1 songs with j - 1 different songs, then the last one is definitely

new song with the choices of N - (j - 1).

Case 2 (the last added one is old song): listen i - 1 songs with j different songs, then the last one is definitely old

song with the choices of j

if without the constraint of K, the status equation will be

dp[i][j] = dp[i-1][j-1] \* (N - (j-1)) + dp[i-1][j] \* j

If with the constaint of K, there are also two cases

Case 1: no changes since the last added one is new song. Hence, there is no conflict

Case 2: now we don't have choices of j for the last added old song. It should be updated j - k because k songs can't be

chosed from j - 1 to j - k. However, if j <= K, this case will be 0 because only after choosing K different other songs,

old song can be chosen.

if (j > k)

dp[i][j] = dp[i-1][j-1] \* (N- (j-1)) + dp[i-1][j] \* (j-k)

else

dp[i][j] = dp[i-1][j-1] \* (N- (j-1))

https://www.jianshu.com/p/e09a051f5420

Method 1: 2D dp

Time complexity: O(L\*N)

Space complexity: O(L\*N)

class Solution {

public int numMusicPlaylists(int N, int L, int K) {

int mod = (int)Math.pow(10, 9) + 7;

long[][] dp = new long[L+1][N+1];

dp[0][0] = 1;

for (int i = 1; i <= L; i++){

for (int j = 1; j <= N; j++){

dp[i][j] = (dp[i-1][j-1] \* (N - (j-1)))%mod;

if (j > K){

dp[i][j] = (dp[i][j] + (dp[i-1][j] \* (j-K))%mod)%mod;

}

}

}

return (int)dp[L][N];

}

}

921\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S of '(' and ')' parentheses, we add the minimum number of parentheses ( '(' or ')', and in any positions ) so that the resulting parentheses string is valid.

Formally, a parentheses string is valid if and only if:

It is the empty string, or

It can be written as AB (A concatenated with B), where A and B are valid strings, or

It can be written as (A), where A is a valid string.

Given a parentheses string, return the minimum number of parentheses we must add to make the resulting string valid.

Example 1:

Input: "())"

Output: 1

Example 2:

Input: "((("

Output: 3

Example 3:

Input: "()"

Output: 0

Example 4:

Input: "()))(("

Output: 4

Note:

S.length <= 1000

S only consists of '(' and ')' characters.

class Solution {

public int minAddToMakeValid(String S) {

int left = 0;

int right = 0;

int res = 0;

for (char c : S.toCharArray()){

if (c == '('){

if (right > left){

right = 0;

left = 0;

}

left++;

}else{

right++;

}

if (right > left){

res++;

}

}

return left > right ? res + (left - right) : res;

}

}

Method 2:

class Solution {

public int minAddToMakeValid(String S) {

int diff = 0; //left - right;

int res = 0;

for (char c : S.toCharArray()){

if (c == '('){

diff++;

}else if (diff == 0){ // ensure that diff is always equal or greater than 0

res++;

}else{

diff--;

}

}

return res + diff;

}

}

Best solution:

class Solution {

public int minAddToMakeValid(String S) {

int count = 0;

int res = 0;

for (char c : S.toCharArray()){

if (c == '('){

count++;

}else{

count--;

}

if (count < 0){

res++;

count = 0;

}

}

return res + count;

}

}

Similar as the isValid Function of Leetcode 301 https://github.com/optimisea/Leetcode/blob/master/Java/301\_Remove.java

private boolean isValid(String str){

int count = 0;

for (char c : str.toCharArray()){

if (c == '('){

count++;

}else if (c == ')'){

count--;

if (count < 0){

return false;

}

}

}

return count == 0;

}

}

922\_Sort.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of non-negative integers, half of the integers in A are odd, and half of the integers are even.

Sort the array so that whenever A[i] is odd, i is odd; and whenever A[i] is even, i is even.

You may return any answer array that satisfies this condition.

Example 1:

Input: [4,2,5,7]

Output: [4,5,2,7]

Explanation: [4,7,2,5], [2,5,4,7], [2,7,4,5] would also have been accepted.

Note:

2 <= A.length <= 20000

A.length % 2 == 0

0 <= A[i] <= 1000

Method 1:

class Solution {

public int[] sortArrayByParityII(int[] A) {

int odd = 1;

int even = 0;

int[] res = new int[A.length];

for (int i = 0; i < A.length; i++){

if (A[i] % 2 == 0){

res[even] = A[i];

even += 2;

}else{

res[odd] = A[i];

odd += 2;

}

}

return res;

}

}

Method 2: In-place

Find the odd index with even value, and the even index with odd value, swap them

class Solution {

public int[] sortArrayByParityII(int[] A) {

int odd = 1;

int even = 0;

int n = A.length;

while (odd < n && even < n){

while (even < n && A[even] % 2 == 0){

even += 2;

}

while (odd < n && A[odd] % 2 == 1){

odd += 2;

}

if (odd < n && even < n){

swap(A, odd, even);

even += 2;

odd += 2;

}

}

return A;

}

private void swap(int[] A, int i, int j){

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

923\_3Sum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer array A, and an integer target, return the number of tuples i, j, k such that i < j < k and

A[i] + A[j] + A[k] == target.

As the answer can be very large, return it modulo 10^9 + 7.

Example 1:

Input: A = [1,1,2,2,3,3,4,4,5,5], target = 8

Output: 20

Explanation:

Enumerating by the values (A[i], A[j], A[k]):

(1, 2, 5) occurs 8 times;

(1, 3, 4) occurs 8 times;

(2, 2, 4) occurs 2 times;

(2, 3, 3) occurs 2 times.

Example 2:

Input: A = [1,1,2,2,2,2], target = 5

Output: 12

Explanation:

A[i] = 1, A[j] = A[k] = 2 occurs 12 times:

We choose one 1 from [1,1] in 2 ways,

and two 2s from [2,2,2,2] in 6 ways.

Note:

3 <= A.length <= 3000

0 <= A[i] <= 100

0 <= target <= 300

Method 1:

Time complexity: O(N^2)

Space complexity: O(N)

class Solution {

public int threeSumMulti(int[] A, int target) {

Map<Integer, Long> map = new HashMap<>();

for (int i : A){

map.put(i, map.getOrDefault(i, 0L) + 1L);

}

int n = map.size();

int[] B = new int[n];

Arrays.sort(A);

///remove duplicates from sorted array

int index = 0;

B[0] = A[0];

for (int i = 0; i < A.length; i++){

if (B[index] != A[i]){

index++;

B[index] = A[i];

}

}

//////////////////////////////////

int mod = (int)Math.pow(10, 9) + 7;

long res = 0;

for (int i = 0; i < n; i++){

int sumTarget = target - B[i];

if (sumTarget >= B[i]){

for (int j = i; j < n; j++){

if (sumTarget - B[j] >= B[j] && map.containsKey(sumTarget - B[j])){

int first = B[i];

int second = B[j];

int third = sumTarget - B[j];

long num = map.get(second);

System.out.println(num);

if (first < second && second < third){

res = (res + ((map.get(first) \* map.get(second)) \* map.get(third))%mod)%mod;

}else if (first == second && second < third){

res = (res + ((num \* (num - 1) / 2) % mod \* map.get(third))%mod)%mod;

}else if (first < second && second == third){

res = (res + (map.get(first) \* ((num \* (num - 1) / 2)%mod))%mod)%mod;

}else if (first == second && second == third){

res = (res + (num \* (num - 1) \* (num - 2) / 6)%mod)%mod;

}

}

}

}

}

return (int)res;

}

}

Method 2:

Build a map for counting different sums of two numbers.

class Solution {

public int threeSumMulti(int[] A, int target) {

Map<Integer, Integer> map = new HashMap<>();

int res = 0;

int mod = (int) Math.pow(10, 9) + 7;

for (int i = 0; i < A.length; i++){

res = (res + map.getOrDefault(target - A[i], 0)) % mod;

for (int j = 0; j < i; j++){

map.put(A[i] + A[j], map.getOrDefault(A[i] + A[j], 0) + 1);

}

}

return res;

}

}

924\_Minimize.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a network of nodes, each node i is directly connected to another node j if and only if graph[i][j] = 1.

Some nodes initial are initially infected by malware. Whenever two nodes are directly connected and at least one of those two nodes

is infected by malware, both nodes will be infected by malware. This spread of malware will continue until no more nodes can be

infected in this manner.

Suppose M(initial) is the final number of nodes infected with malware in the entire network, after the spread of malware stops.

We will remove one node from the initial list. Return the node that if removed, would minimize M(initial). If multiple nodes

could be removed to minimize M(initial), return such a node with the smallest index.

Note that if a node was removed from the initial list of infected nodes, it may still be infected later as a result of the malware

spread.

Example 1:

Input: graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]

Output: 0

Example 2:

Input: graph = [[1,0,0],[0,1,0],[0,0,1]], initial = [0,2]

Output: 0

Example 3:

Input: graph = [[1,1,1],[1,1,1],[1,1,1]], initial = [1,2]

Output: 1

Note:

1 < graph.length = graph[0].length <= 300

0 <= graph[i][j] == graph[j][i] <= 1

graph[i][i] = 1

1 <= initial.length < graph.length

0 <= initial[i] < graph.length

Union Find: the complexity of Union Find with path compression is O(1): return parent[x] = find(parent[x]);

O(N^2)

class Solution {

class UF {

int[] parent;

int[] size;

public UF(int N){

parent = new int[N];

size = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

private int find(int x){

if (x == parent[x]){

return x;

}

return parent[x] = find(parent[x]);

}

private void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

}

}

}

public int minMalwareSpread(int[][] graph, int[] initial) {

int res = -1;

int N = graph.length;

UF uf = new UF(N);

for (int i = 0; i < graph.length; i++){

for (int j = 0; j < graph[i].length; j++){

if (graph[i][j] == 1){

uf.union(i, j);

}

}

}

int max = 0;

for (int i : initial){

int root = uf.find(i);

int size = uf.size[root];

if (max < size){

max = size;

res = i;

}else if (max == size && i < res){

res = i;

}

}

return res;

}

}

925\_Long.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Your friend is typing his name into a keyboard. Sometimes, when typing a character c, the key might get long pressed, and the

character will be typed 1 or more times.

You examine the typed characters of the keyboard. Return True if it is possible that it was your friends name, with some

characters (possibly none) being long pressed.

Example 1:

Input: name = "alex", typed = "aaleex"

Output: true

Explanation: 'a' and 'e' in 'alex' were long pressed.

Example 2:

Input: name = "saeed", typed = "ssaaedd"

Output: false

Explanation: 'e' must have been pressed twice, but it wasn't in the typed output.

Example 3:

Input: name = "leelee", typed = "lleeelee"

Output: true

Example 4:

Input: name = "laiden", typed = "laiden"

Output: true

Explanation: It's not necessary to long press any character.

Note:

name.length <= 1000

typed.length <= 1000

The characters of name and typed are lowercase letters.

class Solution {

public boolean isLongPressedName(String name, String typed) {

int m = name.length();

int n = typed.length();

int i = 0;

int j = 0;

while (i < m && j < n){

if (name.charAt(i) != typed.charAt(j)){

return false;

}

i++;

j++;

int c1 = 1;

int c2 = 1;

while (i < m && name.charAt(i) == name.charAt(i-1)){

i++;

c1++;

}

while (j < n && typed.charAt(j) == typed.charAt(j-1)){

j++;

c2++;

}

if (c2 < c1){

return false;

}

}

if (i == m && j == n){

return true;

}

return false;

}

}

926\_Flip.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A string of '0's and '1's is monotone increasing if it consists of some number of '0's (possibly 0), followed by some number of '1's

(also possibly 0.)

We are given a string S of '0's and '1's, and we may flip any '0' to a '1' or a '1' to a '0'.

Return the minimum number of flips to make S monotone increasing.

Example 1:

Input: "00110"

Output: 1

Explanation: We flip the last digit to get 00111.

Example 2:

Input: "010110"

Output: 2

Explanation: We flip to get 011111, or alternatively 000111.

Example 3:

Input: "00011000"

Output: 2

Explanation: We flip to get 00000000.

Note:

1 <= S.length <= 20000

S only consists of '0' and '1' characters.

Time complexity: O(N)

Space complexity: O(N)

This is DP partition problem

dp[i] denotes the minimum swap needed to ensure all 0 before index i and all 1 after and inlcude index i

The key idea to find the cut point where the left of the point will be 0 and the right of the point will be 1. So we can

use DP to find the point which yields the minimum flip. This is a top down DP solution

cut pointt: 0 1 2 3 4 5 6 7

array: 1 0 0 1 1 0 1

Note that if the length of the array is n, then there will be n+1 cut point. So dp length will be n+1

Step 1: initialization. the first cut point is at index 0. All the elements need to be 1. So the flip number should be equal to

the number of 0

Step 2: Iteration.

If the element is '1', it needs to be changed to '0', and the right side of the cut point are no changes.

So dp[i] = dp[i-1] + 1; If the element is '0', it does not to be changed, however, this element has to be changed to '1' in

previous cut point dp[i-1] because it is '0'. So compared to previouis step dp[i-1], The number of flip is reduced by 1.

Hence, dp[i] = dp[i-1] - 1;

Step 3: Top down DP to find minimum

Also check Leetcode 1004 Max Consecutive Ones III

https://github.com/optimisea/Leetcode/blob/master/Java/1004\_Max.java

Method 1: DP

class Solution {

public int minFlipsMonoIncr(String S) {

int n = S.length();

int[] dp = new int[n+1];

int zeros = 0;

int ones = 0;

for (int i = 0; i < n; i++){

if (S.charAt(i) == '0'){

zeros++;

}else{

ones++;

}

}

dp[0] = zeros; //all '0' need to be '1'

for (int i = 1; i <= n; i++){

if (S.charAt(i-1) == '1'){

dp[i] = dp[i-1] + 1; //compare to previoius step, one '0' conversion is reduced

}else{

dp[i] = dp[i-1] - 1; //compare to previous step, one '1' conversion is added

}

}

int min = n;

for (int i = 0; i <= n; i++){

min = Math.min(min, dp[i]);

}

return min;

}

}

Method 2: Array

https://leetcode.com/problems/flip-string-to-monotone-increasing/discuss/183896/Prefix-Suffix-Java-O(N)-One-Pass-Solution-Space-O(1)

927\_Three.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of 0s and 1s, divide the array into 3 non-empty parts such that all of these parts represent the same binary value.

If it is possible, return any [i, j] with i+1 < j, such that:

A[0], A[1], ..., A[i] is the first part;

A[i+1], A[i+2], ..., A[j-1] is the second part, and

A[j], A[j+1], ..., A[A.length - 1] is the third part.

All three parts have equal binary value.

If it is not possible, return [-1, -1].

Note that the entire part is used when considering what binary value it represents. For example, [1,1,0] represents 6 in decimal,

not 3. Also, leading zeros are allowed, so [0,1,1] and [1,1] represent the same value.

Example 1:

Input: [1,0,1,0,1]

Output: [0,3]

Example 2:

Input: [1,1,0,1,1]

Output: [-1,-1]

Note:

3 <= A.length <= 30000

A[i] == 0 or A[i] == 1

class Solution {

public int[] threeEqualParts(int[] A) {

int n = A.length;

int ones = 0;

for (int i : A){

if (i == 1){

ones++;

}

}

if (ones % 3 != 0){

return new int[]{-1, -1};

}

//corner case: all zeros

if (ones == 0){

return new int[]{0, n-1};

}

int targetOnes = ones / 3;

//construct the target string from right most

StringBuilder sb = new StringBuilder();

int j = n - 1;

int rightOnes = 0;

while (j >= 0 && rightOnes < targetOnes){

sb.append(A[j]);

if (A[j] == 1){

rightOnes++;

}

j--;

}

String target = sb.reverse().toString();

//skip leading zeros at left

int i = 0;

while (i < n && A[i] == 0){

i++;

}

//match target string at left

int k = 0;

while (k < target.length()){

if (A[i+k] == target.charAt(k) - '0'){

k++;

}else{

return new int[]{-1, -1};

}

}

int left = i + k - 1;

//skip leading zeros at middle

i = i + k;

while (i < n && A[i] == 0){

i++;

}

//match target string at middle

k = 0;

while (k < target.length()){

if (A[i+k] == target.charAt(k) - '0'){

k++;

}else{

return new int[]{-1, -1};

}

}

return new int[]{left, i + k};

}

}

928\_Minimize.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(This problem is the same as Minimize Malware Spread, with the differences bolded.)

In a network of nodes, each node i is directly connected to another node j if and only if graph[i][j] = 1.

Some nodes initial are initially infected by malware. Whenever two nodes are directly connected and at least one of those two nodes

is infected by malware, both nodes will be infected by malware. This spread of malware will continue until no more nodes can be

infected in this manner.

Suppose M(initial) is the final number of nodes infected with malware in the entire network, after the spread of malware stops.

We will remove one node from the initial list, completely removing it and any connections from this node to any other node.

Return the node that if removed, would minimize M(initial). If multiple nodes could be removed to minimize M(initial),

return such a node with the smallest index.

Example 1:

Input: graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]

Output: 0

Example 2:

Input: graph = [[1,1,0],[1,1,1],[0,1,1]], initial = [0,1]

Output: 1

Example 3:

Input: graph = [[1,1,0,0],[1,1,1,0],[0,1,1,1],[0,0,1,1]], initial = [0,1]

Output: 1

Note:

1 < graph.length = graph[0].length <= 300

0 <= graph[i][j] == graph[j][i] <= 1

graph[i][i] = 1

1 <= initial.length < graph.length

0 <= initial[i] < graph.length

This is the similar question as Leetcode 924. The code could be reused.

Please refer to https://leetcode.com/problems/minimize-malware-spread/discuss/181470/Very-Simple-Union-Find-O(N2)-no-others

for the solution of Leetcode 924

The key difference here is that the removed point will be completely isolated. So we can create union find for each case that the

initial element is removed, and calculate the total infected number in this case, then find the minimum case. If equal, find the

smallest index.

So most of the code in Leetcode 924 could be reused here. The time complexity in this question will be O(mNN) compared to O(N\*N)

in Leetcode 924.

class Solution {

class UF {

int[] parent;

int[] size;

public UF(int N){

parent = new int[N];

size = new int[N];

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (parent[x] == x){

return x;

}

return parent[x] = find(parent[x]);

}

public void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

}

}

}

public int minMalwareSpread(int[][] graph, int[] initial) {

int N = graph.length;

if (initial.length == 1){

return initial[0];

}

int max = N;

int res = -1;

for (int k = 0; k < initial.length; k++){

UF uf = new UF(N);

int removePt = initial[k];

for (int i = 0; i < N; i++){

for (int j = 0; j < N; j++){

if (i == removePt || j == removePt){

continue;

}

if (graph[i][j] == 1){

uf.union(i, j);

}

}

}

int totalInfected = 0;

Set<Integer> seen = new HashSet<>();

for (int init : initial){

if (init == removePt){

continue;

}

int root = uf.find(init);

int size = uf.size[root];

if (!seen.contains(root)){

seen.add(root);

totalInfected += size;

}

}

if (max > totalInfected){

res = removePt;

max = totalInfected;

}else if (totalInfected == max && removePt < res){

res = removePt;

}

}

return res;

}

}

929\_Unique.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Every email consists of a local name and a domain name, separated by the @ sign.

For example, in alice@leetcode.com, alice is the local name, and leetcode.com is the domain name.

Besides lowercase letters, these emails may contain '.'s or '+'s.

If you add periods ('.') between some characters in the local name part of an email address, mail sent there will be forwarded to

the same address without dots in the local name. For example, "alice.z@leetcode.com" and "alicez@leetcode.com" forward to the same

email address. (Note that this rule does not apply for domain names.)

If you add a plus ('+') in the local name, everything after the first plus sign will be ignored. This allows certain emails to be

filtered, for example m.y+name@email.com will be forwarded to my@email.com. (Again, this rule does not apply for domain names.)

It is possible to use both of these rules at the same time.

Given a list of emails, we send one email to each address in the list. How many different addresses actually receive mails?

Example 1:

Input: ["test.email+alex@leetcode.com","test.e.mail+bob.cathy@leetcode.com","testemail+david@lee.tcode.com"]

Output: 2

Explanation: "testemail@leetcode.com" and "testemail@lee.tcode.com" actually receive mails

Note:

1 <= emails[i].length <= 100

1 <= emails.length <= 100

Each emails[i] contains exactly one '@' character.

class Solution {

public int numUniqueEmails(String[] emails) {

Set<String> set = new HashSet<>();

for (String email : emails){

String[] str = email.split("@");

String local = str[0];

String domain = str[1];

StringBuilder sb = new StringBuilder();

for (char c : local.toCharArray()){

if (c == '.'){

continue;

}

if (c == '+'){

break;

}

sb.append(c);

}

sb.append("@" + domain);

set.add(sb.toString());

}

return set.size();

}

}

93\_Restore.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:

Given "25525511135",

return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

class Solution {

public List<String> restoreIpAddresses(String s) {

List<String> result = new ArrayList<>();

if (s.length() > 12){

return result;

}

dfs(result, s, "", 0, 4);

return result;

}

private void dfs(List<String> result, String s, String item, int start, int k){

if (k == 0 && start == s.length()){

result.add(item);

return;

}

for (int i = start; i < s.length() && i - start < 3 ; i++){

String str = s.substring(start, i+1);

if (str.length() > 1 && str.startsWith("0")){

break;

}

int num = Integer.parseInt(str);

if (num <= 255 && num >= 0){

dfs(result, s, start == 0 ? str : item + "." + str, i + 1, k-1);

}

}

}

}

class Solution {

public List<String> restoreIpAddresses(String s) {

List<String> res = new ArrayList<>();

if (s.length() > 12){

return res;

}

dfs(res, "", s, 0, 4);

return res;

}

private void dfs(List<String> res, String item, String s, int start, int k){

if (k == 0 && start == s.length()){

res.add(item);

return;

}

for (int i = start; i < s.length() && i - start < 3; i++){

String str = s.substring(start, i+1);

if (str.length() > 1 && str.charAt(0) == '0'){

break;

}

int num = Integer.parseInt(str);

if (num > 255){

break;

}

dfs(res, start == 0 ? str : item + "." + str, s, i+1, k-1);

}

}

}

https://leetcode.com/problems/restore-ip-addresses/discuss/30944/Very-simple-DFS-solution

930\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In an array A of 0s and 1s, how many non-empty subarrays have sum S?

Example 1:

Input: A = [1,0,1,0,1], S = 2

Output: 4

Explanation:

The 4 subarrays are bolded below:

[1,0,1,0,1]

[1,0,1,0,1]

[1,0,1,0,1]

[1,0,1,0,1]

Note:

A.length <= 30000

0 <= S <= A.length

A[i] is either 0 or 1.

This is the same as 560. Subarray Sum Equals K

Also check 974

class Solution {

public int numSubarraysWithSum(int[] A, int S) {

int count = 0;

Map<Integer, Integer> map = new HashMap<>();

map.put(0, 1);

int sum = 0;

for (int i : A){

sum += i;

if (map.containsKey(sum - S)){

count += map.get(sum - S);

}

map.put(sum, map.getOrDefault(sum, 0) + 1);

}

return count;

}

}

931\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a square array of integers A, we want the minimum sum of a falling path through A.

A falling path starts at any element in the first row, and chooses one element from each row. The next row's choice must be in a column that is different from the previous row's column by at most one.

Example 1:

Input: [[1,2,3],[4,5,6],[7,8,9]]

Output: 12

Explanation:

The possible falling paths are:

[1,4,7], [1,4,8], [1,5,7], [1,5,8], [1,5,9]

[2,4,7], [2,4,8], [2,5,7], [2,5,8], [2,5,9], [2,6,8], [2,6,9]

[3,5,7], [3,5,8], [3,5,9], [3,6,8], [3,6,9]

The falling path with the smallest sum is [1,4,7], so the answer is 12.

Note:

1 <= A.length == A[0].length <= 100

-100 <= A[i][j] <= 100

Top down DP Solution which is very straightforward.

dp solution:

1) dp[i][j] denotes the minimum path sum at point [i, j].

2) The formula of dp is below

dp[i][j] = A[i][j] + Min(dp[i-1][j-1], dp[i-1][j]. dp[i-1][j+1])

To take care of the boundary case, we need use two if conditions (i.e. if j >= 1 and if j < n -1)

3) Initialize the dp with the first row of A matrix.

4) So the solution will be the minimum value at the last row of dp matrix (i.e., dp[m-1][j]).

```

class Solution {

public int minFallingPathSum(int[][] A) {

int m = A.length;

int n = A[0].length;

int[][] dp = new int[m][n];

for (int j = 0; j < n; j++){

dp[0][j] = A[0][j];

}

for (int i = 1; i < m; i++){

for (int j = 0; j < n; j++){

dp[i][j] = dp[i-1][j];

if (j >= 1){

dp[i][j] = Math.min(dp[i][j], dp[i-1][j-1]);

}

if (j < n - 1){

dp[i][j] = Math.min(dp[i][j], dp[i-1][j+1]);

}

dp[i][j] += A[i][j];

}

}

int res = Integer.MAX\_VALUE;

for (int j = 0; j < n; j++){

res = Math.min(res, dp[m-1][j]);

}

return res;

}

}

```

933\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a class RecentCounter to count recent requests.

It has only one method: ping(int t), where t represents some time in milliseconds.

Return the number of pings that have been made from 3000 milliseconds ago until now.

Any ping with time in [t - 3000, t] will count, including the current ping.

It is guaranteed that every call to ping uses a strictly larger value of t than before.

Example 1:

Input: inputs = ["RecentCounter","ping","ping","ping","ping"], inputs = [[],[1],[100],[3001],[3002]]

Output: [null,1,2,3,3]

Note:

Each test case will have at most 10000 calls to ping.

Each test case will call ping with strictly increasing values of t.

Each call to ping will have 1 <= t <= 10^9.

Method 1: TreeMap

Time complexity: O(logn)

class RecentCounter {

TreeMap<Integer, Integer> map;

public RecentCounter() {

map = new TreeMap<>();

}

public int ping(int t) {

Integer i = map.lowerKey(t);

if (i == null){

map.put(t, 1);

return 1;

}

map.put(t, map.get(i) + 1);

int start = 0;

if (map.containsKey(t-3000)){

start = map.get(t-3000);

return map.get(t) - start + 1;

}

Integer j = map.lowerKey(t-3000);

if (j == null){

return map.get(t);

}

return map.get(t) - map.get(j);

}

}

/\*\*

\* Your RecentCounter object will be instantiated and called as such:

\* RecentCounter obj = new RecentCounter();

\* int param\_1 = obj.ping(t);

\*/

class RecentCounter {

TreeMap<Integer, Integer> map;

public RecentCounter() {

map = new TreeMap<>();

}

public int ping(int t) {

map.put(t, 1 + map.size());

return map.get(t) - map.ceilingEntry(t-3000).getValue() + 1;

}

}

/\*\*

\* Your RecentCounter object will be instantiated and called as such:

\* RecentCounter obj = new RecentCounter();

\* int param\_1 = obj.ping(t);

\*/

934\_Shortest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a given 2D binary array A, there are two islands. (An island is a 4-directionally connected group of 1s not connected to any other 1s.)

Now, we may change 0s to 1s so as to connect the two islands together to form 1 island.

Return the smallest number of 0s that must be flipped. (It is guaranteed that the answer is at least 1.)

Example 1:

Input: [[0,1],[1,0]]

Output: 1

Example 2:

Input: [[0,1,0],[0,0,0],[0,0,1]]

Output: 2

Example 3:

Input: [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]

Output: 1

Note:

1 <= A.length = A[0].length <= 100

A[i][j] == 0 or A[i][j] == 1

Method 1: BFS

Time complexity: O((mn)^2)

class Solution {

public int shortestBridge(int[][] A) {

int m = A.length;

int n = A[0].length;

boolean found = false;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (A[i][j] == 1){

bfs(A, i, j);

found = true;

break;

}

}

if (found){

break;

}

}

int res = Integer.MAX\_VALUE;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (A[i][j] == 1){

boolean[][] visited = new boolean[m][n];

int flip = bfsConnect(A, i, j, visited);

res = Math.min(res, flip);

}

}

}

return res;

}

private void bfs(int[][] A, int x, int y){

int m = A.length;

int n = A[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

queue.offer(new int[]{x, y});

A[x][y] = -1;

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && A[nx][ny] == 1){

A[nx][ny] = -1;

queue.offer(new int[]{nx, ny});

}

}

}

}

private int bfsConnect(int[][] A, int x, int y, boolean[][] visited){

int target = -A[x][y];

int m = A.length;

int n = A[0].length;

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

Queue<int[]> queue = new LinkedList<>();

queue.offer(new int[]{x, y});

int res = 0;

while (!queue.isEmpty()){

int size = queue.size();

for (int i = 0; i < size; i++){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

if (A[nx][ny] == target){

return res;

}else if (A[nx][ny] == 0){

queue.offer(new int[]{nx, ny});

visited[nx][ny] = true;

}

}

}

}

res++;

}

return Integer.MAX\_VALUE;

}

}

Method 2: better version

Time complexity: O(mn)

Store islandQueue and expand it using BFS

class Solution {

public int shortestBridge(int[][] A) {

int m = A.length;

int n = A[0].length;

Queue<int[]> islandQueue = new LinkedList<>();

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

boolean[][] visited = new boolean[m][n];

boolean found = false;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (A[i][j] == 1){

bfs(A, i, j, islandQueue, visited);

found = true;

break;

}

}

if (found){

break;

}

}

int res = 0;

while (!islandQueue.isEmpty()){

int size = islandQueue.size();

for (int i = 0; i < size; i++){

int[] curr = islandQueue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny]){

if (A[nx][ny] == 1){

return res;

}

islandQueue.offer(new int[]{nx, ny});

visited[nx][ny] = true;

}

}

}

res++;

}

return -1;

}

private void bfs(int[][] A, int x, int y, Queue<int[]> islandQueue, boolean[][] visited){

int m = A.length;

int n = A[0].length;

Queue<int[]> queue = new LinkedList<>();

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

queue.offer(new int[]{x, y});

visited[x][y] = true;

islandQueue.offer(new int[]{x, y});

while (!queue.isEmpty()){

int[] curr = queue.poll();

for(int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && !visited[nx][ny] && A[nx][ny] == 1){

queue.offer(new int[]{nx, ny});

visited[nx][ny] = true;

islandQueue.offer(new int[]{nx, ny});

}

}

}

}

}

935\_Knight.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This time, we place our chess knight on any numbered key of a phone pad (indicated above), and the knight makes N-1 hops. Each hop must be from one key to another numbered key.

Each time it lands on a key (including the initial placement of the knight), it presses the number of that key, pressing N digits total.

How many distinct numbers can you dial in this manner?

Since the answer may be large, output the answer modulo 10^9 + 7.

Example 1:

Input: 1

Output: 10

Example 2:

Input: 2

Output: 20

Example 3:

Input: 3

Output: 46

Note:

1 <= N <= 5000

Top down DP

Probably this is the simplest top down dynamic programming solution

dp[i][j] denotes solution (e.g., distinct numbers) the i-th move and j-th key. The relationship between dp[i][j] and last step dp[i-1]

are explicitly written as below. This solution could be further simplified as 1D DP and array to respresent different cases more

conciesly. But I think it is more intutivie to put it like this way below.

class Solution {

public int knightDialer(int N) {

int mod = (int)Math.pow(10, 9) + 7;

int[][] dp = new int[N][10];

for (int j = 0; j <= 9; j++){

dp[0][j] = 1;

}

for (int i = 1; i < N; i++){

for (int j = 0;j <= 9; j++){

if (j == 0){

dp[i][j] = (dp[i-1][4] + dp[i-1][6]) % mod;

}else if (j == 1){

dp[i][j] = (dp[i-1][6] + dp[i-1][8]) % mod;

}else if (j == 2){

dp[i][j] = (dp[i-1][7] + dp[i-1][9]) % mod;

}else if (j == 3){

dp[i][j] = (dp[i-1][4] + dp[i-1][8]) % mod;

}else if (j == 4){

dp[i][j] = ((dp[i-1][0] + dp[i-1][3]) % mod + dp[i-1][9]) % mod;

}else if (j == 5){

dp[i][j] = 0;

}else if (j == 6){

dp[i][j] = ((dp[i-1][0] + dp[i-1][1]) % mod + dp[i-1][7]) % mod;

}else if (j == 7){

dp[i][j] = (dp[i-1][2] + dp[i-1][6]) % mod;

}else if (j == 8){

dp[i][j] = (dp[i-1][1] + dp[i-1][3]) % mod;

}else if (j == 9){

dp[i][j] = (dp[i-1][2] + dp[i-1][4]) % mod;

}

}

}

int res = 0;

for (int j = 0; j <= 9; j++){

res = (res + dp[N-1][j]) % mod;

}

return res;

}

}

937\_Reorder.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have an array of logs. Each log is a space delimited string of words.

For each log, the first word in each log is an alphanumeric identifier. Then, either:

Each word after the identifier will consist only of lowercase letters, or;

Each word after the identifier will consist only of digits.

We will call these two varieties of logs letter-logs and digit-logs. It is guaranteed that each log has at least one word

after its identifier.

Reorder the logs so that all of the letter-logs come before any digit-log. The letter-logs are ordered lexicographically

ignoring identifier, with the identifier used in case of ties. The digit-logs should be put in their original order.

Return the final order of the logs.

Example 1:

Input: ["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key dog","a8 act zoo"]

Output: ["g1 act car","a8 act zoo","ab1 off key dog","a1 9 2 3 1","zo4 4 7"]

Note:

0 <= logs.length <= 100

3 <= logs[i].length <= 100

logs[i] is guaranteed to have an identifier, and a word after the identifier.

class Solution {

public String[] reorderLogFiles(String[] logs) {

Arrays.sort(logs, new Comparator<String>(){

public int compare (String log1, String log2){

String[] s1 = log1.split(" ");

String[] s2 = log2.split(" ");

boolean s1Digit = Character.isDigit(s1[1].charAt(0));

boolean s2Digit = Character.isDigit(s2[1].charAt(0));

if (s1Digit && s2Digit){

return 0;

}else if (s1Digit && !s2Digit){

return 1;

}else if (!s1Digit && s2Digit){

return -1;

}else{

int index1 = log1.indexOf(" ");

String str1 = log1.substring(index1 + 1);

int index2 = log2.indexOf(" ");

String str2 = log2.substring(index2+ 1);

if (str1.equals(str2)){

return log1.substring(0, index1).compareTo(log2.substring(0, index2));

}

return str1.compareTo(str2);

}

}

});

return logs;

}

}

938\_Range.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the root node of a binary search tree, return the sum of values of all nodes with value between L and R (inclusive).

The binary search tree is guaranteed to have unique values.

Example 1:

Input: root = [10,5,15,3,7,null,18], L = 7, R = 15

Output: 32

Example 2:

Input: root = [10,5,15,3,7,13,18,1,null,6], L = 6, R = 10

Output: 23

Note:

The number of nodes in the tree is at most 10000.

The final answer is guaranteed to be less than 2^31.

Method 1: recusion

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int rangeSumBST(TreeNode root, int L, int R) {

if (root == null){

return 0;

}

if (root.val < L){

return rangeSumBST(root.right, L, R);

}

if (root.val > R){

return rangeSumBST(root.left, L, R);

}

return root.val + rangeSumBST(root.left, L, root.val) + rangeSumBST(root.right, root.val, R);

}

}

Method 2: in order traversal

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public int rangeSumBST(TreeNode root, int L, int R) {

List<Integer> list = new ArrayList<>();

inOrder(root, list);

int sum = 0;

boolean startAdd = false;

for(int i = 0; i < list.size(); i++){

if (list.get(i) == L){

startAdd = true;

}

if (startAdd == true){

sum += list.get(i);

}

if (list.get(i) == R){

startAdd = false;

}

}

return sum;

}

private void inOrder(TreeNode root, List<Integer> list){

if (root == null){

return;

}

inOrder(root.left, list);

list.add(root.val);

inOrder(root.right, list);

}

}

939\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a set of points in the xy-plane, determine the minimum area of a rectangle formed from these points, with sides parallel

to the x and y axes.

If there isn't any rectangle, return 0.

Example 1:

Input: [[1,1],[1,3],[3,1],[3,3],[2,2]]

Output: 4

Example 2:

Input: [[1,1],[1,3],[3,1],[3,3],[4,1],[4,3]]

Output: 2

Note:

1 <= points.length <= 500

0 <= points[i][0] <= 40000

0 <= points[i][1] <= 40000

All points are distinct.

Method 1: Brute Force

Time complexity: O(N^2)

Space complexity: O(N)

class Solution {

public int minAreaRect(int[][] points) {

int max = 40000;

int res = Integer.MAX\_VALUE;

Set<Integer> set = new HashSet<>();

int n = points.length;

for (int i = 0; i < n; i++){

for (int j = 0; j < n; j++){

if (points[i][0] == points[j][0] && points[i][1] == points[j][1]){

continue;

}

if (set.contains(points[i][0] \* max + points[j][1]) && set.contains(points[j][0] \* max + points[i][1])){

int area = Math.abs(points[i][0] - points[j][0]) \* Math.abs(points[i][1] - points[j][1]);

res = Math.min(res, area);

}

}

set.add(points[i][0] \* max + points[i][1]);

}

return res == Integer.MAX\_VALUE ? 0 : res;

}

}

Method 2:

class Solution {

public int minAreaRect(int[][] points) {

int res = Integer.MAX\_VALUE;

Map<Integer, Set<Integer>> map = new HashMap<>();

for (int[] p : points){

if (!map.containsKey(p[0])){

map.put(p[0], new HashSet<>());

}

map.get(p[0]).add(p[1]);

}

for (int[] p1: points){

for (int[] p2: points){

if (p1[0] == p2[0] || p1[1] == p2[1]){ // note that this can't be &&

continue;

}

if (map.get(p1[0]).contains(p2[1]) && map.get(p2[0]).contains(p1[1])){

int area = Math.abs(p1[0] - p2[0]) \* Math.abs(p1[1] - p2[1]);

res = Math.min(res, area);

}

}

}

return res == Integer.MAX\_VALUE ? 0 : res;

}

}

94\_BinaryTreeInorderTraversal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the inorder traversal of its nodes' values.

For example:

Given binary tree [1,null,2,3],

1

\

2

/

3

return [1,3,2].

Note: Recursive solution is trivial, could you do it iteratively?

Method 1: divide & conquer

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

if (root == null){

return result;

}

List<Integer> left = inorderTraversal(root.left);

List<Integer> right = inorderTraversal(root.right);

result.addAll(left);

result.add(root.val);

result.addAll(right);

return result;

}

}

Method 2: recursion

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

helper(result, root);

return result;

}

private void helper(List<Integer> result, TreeNode node){

if (node == null){

return;

}

helper(result, node.left);

result.add(node.val);

helper(result, node.right);

}

}

class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<>();

Stack<TreeNode> stack = new Stack<>();

TreeNode curr = root;

while (curr != null || !stack.isEmpty()){

if (curr != null){

stack.push(curr);

curr = curr.left;

}

if (curr == null && !stack.isEmpty()){

TreeNode node = stack.pop();

res.add(node.val);

curr = node.right;

}

}

return res;

}

}

Method 3: iteration:

Template: https://leetcode.com/problems/validate-binary-search-tree/discuss/32112/Learn-one-iterative-inorder-traversal-apply-it-to-multiple-tree-questions-(Java-Solution)

Best version:

class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<>();

Stack<TreeNode> stack = new Stack<>();

TreeNode node = root;

while (node != null || !stack.isEmpty()){

while (node != null){

stack.push(node);

node = node.left;

}

TreeNode curr = stack.pop();

res.add(curr.val);

node = curr.right;

}

return res;

}

}

Tree Iteration template

https://leetcode.com/problems/validate-binary-search-tree/discuss/32112/Learn-one-iterative-inorder-traversal-apply-it-to-multiple-tree-questions-(Java-Solution)

940\_Distinct.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S, count the number of distinct, non-empty subsequences of S .

Since the result may be large, return the answer modulo 10^9 + 7.

Example 1:

Input: "abc"

Output: 7

Explanation: The 7 distinct subsequences are "a", "b", "c", "ab", "ac", "bc", and "abc".

Example 2:

Input: "aba"

Output: 6

Explanation: The 6 distinct subsequences are "a", "b", "ab", "ba", "aa" and "aba".

Example 3:

Input: "aaa"

Output: 3

Explanation: The 3 distinct subsequences are "a", "aa" and "aaa".

https://www.jianshu.com/p/02501f516437

Dynamic Programming

class Solution {

public int distinctSubseqII(String S) {

int mod = (int)1e9 + 7;

int[] end = new int[26];

for (int i = 0; i < S.length(); i++){

char c = S.charAt(i);

int sum = 0;

for (int j = 0; j < 26; j++){

sum = (sum + end[j]) % mod;

}

end[c - 'a'] = sum + 1;// 1 means new SINGLE character

}

int res = 0;

for (int i = 0; i < 26; i++){

res = (res + end[i]) % mod;

}

return res;

}

}

Method 2: general form

class Solution {

public int distinctSubseqII(String S) {

int mod = (int)1e9 + 7;

Map<Character, Integer> map = new HashMap<>();

for (char c : S.toCharArray()){

int sum = 0;

for (int val : map.values()){

sum = (sum + val)%mod;

}

map.put(c, sum + 1);

}

int res = 0;

for (int val : map.values()){

res = (res + val) % mod;

}

return res;

}

}

Method 2:

https://leetcode.com/problems/distinct-subsequences-ii/discuss/192030/Java-DP-O(N2)-time-greater-O(N)-time-greater-O(1)-space

In the original post,

if s[j] == s[i], do nothing to avoid duplicates.

The sentence is really confusing and misleading. and conceptually wrong.

When s[j] == s[i], the correct explanation is to add existing substring with one more letter, so the number will be the same as previous sum and hence no need to add more.

Take abcc as example,

ending:

a : a

b: ab b

c: ac abc bc c

c: ac abc bc c ====when scan to the first c, add one more letter, acc abcc bcc cc

O(n^2)

class Solution {

public int distinctSubseqII(String S) {

int mod = (int)1e9 + 7;

int n = S.length();

int[] dp = new int[n];

Arrays.fill(dp, 1);

int res = 0;

for (int i = 0; i < n; i++){

for (int j = 0; j < i; j++){

if (S.charAt(i) != S.charAt(j)){

dp[i] = (dp[i] + dp[j])%mod;

}

}

res = (res + dp[i]) % mod;

}

return res;

}

}

The refer to Method 1 for O(N) solution

941\_Valid.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of integers, return true if and only if it is a valid mountain array.

Recall that A is a mountain array if and only if:

A.length >= 3

There exists some i with 0 < i < A.length - 1 such that:

A[0] < A[1] < ... A[i-1] < A[i]

A[i] > A[i+1] > ... > A[B.length - 1]

Example 1:

Input: [2,1]

Output: false

Example 2:

Input: [3,5,5]

Output: false

Example 3:

Input: [0,3,2,1]

Output: true

Note:

0 <= A.length <= 10000

0 <= A[i] <= 10000

they will meet if they climb on the same mountain

class Solution {

public boolean validMountainArray(int[] A) {

int n = A.length;

int i = 0;

int j = n - 1;

while (i < n - 1 && A[i+1] > A[i]){

i++;

}

while (j > 0 && A[j] < A[j-1]){

j--;

}

return i < n - 1 && j > 0 && i == j;

}

}

942\_DIString.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a string S that only contains "I" (increase) or "D" (decrease), let N = S.length.

Return any permutation A of [0, 1, ..., N] such that for all i = 0, ..., N-1:

If S[i] == "I", then A[i] < A[i+1]

If S[i] == "D", then A[i] > A[i+1]

Example 1:

Input: "IDID"

Output: [0,4,1,3,2]

Example 2:

Input: "III"

Output: [0,1,2,3]

Example 3:

Input: "DDI"

Output: [3,2,0,1]

Note:

1 <= S.length <= 10000

S only contains characters "I" or "D".

class Solution {

public int[] diStringMatch(String S) {

int N = S.length();

int[] A = new int[N+1];

int left = 0;

int right = N;

for (int i = 0; i < N; i++){

if (S.charAt(i) == 'D'){

A[i] = right;

right--;

}else{

A[i] = left;

left++;

}

}

A[N] = left;

return A;

}

}

944\_Delete.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given an array A of N lowercase letter strings, all of the same length.

Now, we may choose any set of deletion indices, and for each string, we delete all the characters in those indices.

For example, if we have a string "abcdef" and deletion indices {0, 2, 3}, then the final string after deletion is "bef".

Suppose we chose a set of deletion indices D such that after deletions, each remaining column in A is in non-decreasing sorted order.

Formally, the c-th column is [A[0][c], A[1][c], ..., A[A.length-1][c]]

Return the minimum possible value of D.length.

Example 1:

Input: ["cba","daf","ghi"]

Output: 1

Example 2:

Input: ["a","b"]

Output: 0

Example 3:

Input: ["zyx","wvu","tsr"]

Output: 3

Note:

1 <= A.length <= 100

1 <= A[i].length <= 1000

class Solution {

public int minDeletionSize(String[] A) {

int m = A.length;

int n = A[0].length();

int count = 0;

for (int j = 0; j < n; j++){

for (int i = 1; i < m; i++){

if (A[i].charAt(j) < A[i-1].charAt(j)){

count++;

break;

}

}

}

return count;

}

}

945\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers A, a move consists of choosing any A[i], and incrementing it by 1.

Return the least number of moves to make every value in A unique.

Example 1:

Input: [1,2,2]

Output: 1

Explanation: After 1 move, the array could be [1, 2, 3].

Example 2:

Input: [3,2,1,2,1,7]

Output: 6

Explanation: After 6 moves, the array could be [3, 4, 1, 2, 5, 7].

It can be shown with 5 or less moves that it is impossible for the array to have all unique values.

Note:

0 <= A.length <= 40000

0 <= A[i] < 40000

Method:

Sort the array.

Compared with previous number,

the current number need to be at least prev + 1.

Time Complexity: O(NlogN)

class Solution {

public int minIncrementForUnique(int[] A) {

Arrays.sort(A);

int res = 0;

for (int i = 0; i < A.length - 1; i++){

int val = Math.max(A[i] + 1, A[i+1]);

res += val - A[i+1];

A[i+1] = val;

}

return res;

}

}

class Solution {

public int minIncrementForUnique(int[] A) {

if (A.length <= 1){

return 0;

}

Arrays.sort(A);

int res = 0;

int max = A[0]; // maintain max value before the curr

for (int i = 1; i < A.length; i++){

max = Math.max(max+1, A[i]);

res += max - A[i];

// A[i] = max; //not needed

}

return res;

}

}

Better version:

class Solution {

public int minIncrementForUnique(int[] A) {

Arrays.sort(A);

int res = 0;

int need = 0;

for (int a : A){

res += Math.max(need - a, 0);

need = Math.max(need, a) + 1;

}

return res;

}

}

946\_Validate.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two sequences pushed and popped with distinct values, return true if and only if this could have been the result of a sequence

of push and pop operations on an initially empty stack.

Example 1:

Input: pushed = [1,2,3,4,5], popped = [4,5,3,2,1]

Output: true

Explanation: We might do the following sequence:

push(1), push(2), push(3), push(4), pop() -> 4,

push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

Example 2:

Input: pushed = [1,2,3,4,5], popped = [4,3,5,1,2]

Output: false

Explanation: 1 cannot be popped before 2.

Note:

0 <= pushed.length == popped.length <= 1000

0 <= pushed[i], popped[i] < 1000

pushed is a permutation of popped.

pushed and popped have distinct values.

Method: Simulate Stack

class Solution {

public boolean validateStackSequences(int[] pushed, int[] popped) {

Stack<Integer> stack = new Stack<>();

int index = 0;

for (int num : pushed){

stack.push(num);

while (!stack.isEmpty() && stack.peek() == popped[index]){

stack.pop();

index++;

}

}

return index == popped.length;

}

}

947\_Most.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a 2D plane, we place stones at some integer coordinate points. Each coordinate point may have at most one stone.

Now, a move consists of removing a stone that shares a column or row with another stone on the grid.

What is the largest possible number of moves we can make?

Example 1:

Input: stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]

Output: 5

Example 2:

Input: stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]

Output: 3

Example 3:

Input: stones = [[0,0]]

Output: 0

Note:

1 <= stones.length <= 1000

0 <= stones[i][j] < 10000

Method: Uion Find

class Solution {

class UF {

int[] parent;

int[] size;

int count;

public UF(int N){

parent = new int[N];

size = new int[N];

count = N;

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (x == parent[x]){

return x;

}

return parent[x] = find(parent[x]); // path compression

}

public void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

count--;

}

}

public int getSize(){

return count;

}

}

public int removeStones(int[][] stones) {

int N = stones.length;

UF uf = new UF(N);

for(int i = 0; i < N; i++){

for (int j = i + 1; j < N; j++){

int[] a = stones[i];

int[] b = stones[j];

if (a[0] == b[0] || a[1] == b[1]){

uf.union(i, j);

}

}

}

return N - uf.getSize();

}

}

948\_Bag.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You have an initial power P, an initial score of 0 points, and a bag of tokens.

Each token can be used at most once, has a value token[i], and has potentially two ways to use it.

If we have at least token[i] power, we may play the token face up, losing token[i] power, and gaining 1 point.

If we have at least 1 point, we may play the token face down, gaining token[i] power, and losing 1 point.

Return the largest number of points we can have after playing any number of tokens.

Example 1:

Input: tokens = [100], P = 50

Output: 0

Example 2:

Input: tokens = [100,200], P = 150

Output: 1

Example 3:

Input: tokens = [100,200,300,400], P = 200

Output: 2

Note:

tokens.length <= 1000

0 <= tokens[i] < 10000

0 <= P < 10000

Method: Greedy and Two pointers

class Solution {

public int bagOfTokensScore(int[] tokens, int P) {

if (tokens == null || tokens.length == 0){

return 0;

}

Arrays.sort(tokens);

int left = 0;

int right = tokens.length - 1;

int pts = 0;

int pow = P;

if (P < tokens[0]){

return 0;

}

while (left <= right){

while (left <= right && pow >= tokens[left]){

pow -= tokens[left];

pts++;

left++;

}

if (left < right){

pow += tokens[right];

pts--;

right--;

}else{

break;

}

}

return pts;

}

}

class Solution {

public int bagOfTokensScore(int[] tokens, int P) {

Arrays.sort(tokens);

int left = 0;

int right = tokens.length - 1;

int pts = 0;

int pow = P;

int res = 0;

while (left <= right){

if (pow >= tokens[left]){

pow -= tokens[left];

left++;

pts++;

res = Math.max(res, pts);

}else if (pts > 0){

pow += tokens[right];

right--;

pts--;

}else{

break;

}

}

return res;

}

}

949\_Largest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of 4 digits, return the largest 24 hour time that can be made.

The smallest 24 hour time is 00:00, and the largest is 23:59. Starting from 00:00, a time is larger if more time has elapsed since

midnight.

Return the answer as a string of length 5. If no valid time can be made, return an empty string.

Example 1:

Input: [1,2,3,4]

Output: "23:41"

Example 2:

Input: [5,5,5,5]

Output: ""

Note:

A.length == 4

0 <= A[i] <= 9

Method 1: backtracking

class Solution {

int max = Integer.MIN\_VALUE;

List<Integer> result = new ArrayList<>();

public String largestTimeFromDigits(int[] A) {

if (A == null || A.length == 0){

return "";

}

boolean[] visited = new boolean[4];

List<Integer> res = new ArrayList<>();

dfs(A, res, visited);

if (max == Integer.MIN\_VALUE){

return "";

}

return String.valueOf(result.get(0)) + String.valueOf(result.get(1)) + ":" + String.valueOf(result.get(2)) + String.valueOf(result.get(3));

}

private void dfs(int[] A, List<Integer> res, boolean[] visited){

if (res.size() == A.length){

if (res.get(0) \* 10 + res.get(1) <= 23 && res.get(2) \* 10 + res.get(3) <= 59 &&

res.get(0) \* 1000 + res.get(1) \* 100 + res.get(2) \* 10 + res.get(3) > max){

max = Math.max(max, res.get(0) \* 1000 + res.get(1) \* 100 + res.get(2) \* 10 + res.get(3));

result = new ArrayList<>(res);

}

return;

}

for (int i = 0; i < A.length; i++){

if (!visited[i]){

visited[i] = true;

res.add(A[i]);

dfs(A, res, visited);

visited[i] = false;

res.remove(res.size() - 1);

}

}

}

}

class Solution {

int max = Integer.MIN\_VALUE;

public String largestTimeFromDigits(int[] A) {

if (A == null || A.length == 0){

return "";

}

boolean[] visited = new boolean[4];

List<List<Integer>> resList = new ArrayList<>();

backtrack(resList, A, new ArrayList<Integer>(), visited);

if (max == Integer.MIN\_VALUE){

return "";

}

List<Integer> res = resList.get(resList.size() - 1);

return res.get(0) + String.valueOf(res.get(1)) + ":" + String.valueOf(res.get(2)) + res.get(3);

}

private void backtrack(List<List<Integer>> res, int[] A, List<Integer> item, boolean[] visited){

if (item.size() == A.length){

int cand = item.get(0) \* 1000 + item.get(1) \* 100 + item.get(2) \* 10 + item.get(3);

if (item.get(0) \* 10 + item.get(1) < 24 && item.get(2) \* 10 + item.get(3) < 60 && cand > max){

max = cand;

res.add(new ArrayList<>(item));

}

return;

}

for (int i = 0; i < A.length; i++){

if (!visited[i]){

visited[i] = true;

item.add(A[i]);

backtrack(res, A, item, visited);

visited[i] = false;

item.remove(item.size() - 1);

}

}

}

}

Method 3: Permutation code

class Solution {

public String largestTimeFromDigits(int[] A) {

if (A == null || A.length == 0){

return "";

}

boolean[] visited = new boolean[4];

List<List<Integer>> resList = new ArrayList<>();

permutation(resList, A, new ArrayList<Integer>(), visited);

int max = Integer.MIN\_VALUE;

int index = -1;

for (int i = 0; i < resList.size(); i++){

List<Integer> list = resList.get(i);

int num = convert(list);

if (num > max){

max = num;

index = i;

}

}

if (index == -1 || max == Integer.MIN\_VALUE){

return "";

}

List<Integer> res = resList.get(index);

return res.get(0) + String.valueOf(res.get(1)) + ":" + String.valueOf(res.get(2)) + res.get(3);

}

private void permutation(List<List<Integer>> res, int[] A, List<Integer> item, boolean[] visited){

if (item.size() == A.length){

res.add(new ArrayList<>(item));

return;

}

for (int i = 0; i < A.length; i++){

if (!visited[i]){

visited[i] = true;

item.add(A[i]);

permutation(res, A, item, visited);

visited[i] = false;

item.remove(item.size() - 1);

}

}

}

private int convert(List<Integer> list){

if (list.get(0) \* 10 + list.get(1) < 24 && list.get(2) \* 10 + list.get(3) < 60){

return list.get(0) \* 1000 + list.get(1) \* 100 + list.get(2) \* 10 + list.get(3);

}

return Integer.MIN\_VALUE;

}

}

Another solution Similar to Leetcode 681 Next Closest Time

Best solution:

class Solution {

public String largestTimeFromDigits(int[] A) {

if (A == null || A.length == 0){

return "";

}

boolean[] visited = new boolean[4];

List<List<Integer>> resList = new ArrayList<>();

permutation(resList, A, new ArrayList<Integer>(), visited, 0);

int max = Integer.MIN\_VALUE;

int index = -1;

for (int i = 0; i < resList.size(); i++){

List<Integer> list = resList.get(i);

int num = convert(list);

if (num > max){

max = num;

index = i;

}

}

if (index == -1 || max == Integer.MIN\_VALUE){

return "";

}

List<Integer> res = resList.get(index);

return res.get(0) + String.valueOf(res.get(1)) + ":" + String.valueOf(res.get(2)) + res.get(3);

}

private void permutation(List<List<Integer>> res, int[] A, List<Integer> item, boolean[] visited, int pos){

if (item.size() == A.length){

res.add(new ArrayList<>(item));

return;

}

for (int i = 0; i < A.length; i++){

if (pos == 0 && A[i] > 2){

continue;

}

if (pos == 1 && item.get(0) \* 10 + A[i] > 23){

continue;

}

if (pos == 2 && A[i] > 5){

continue;

}

if (pos == 3 && item.get(2) \* 10 + A[i] > 59){

continue;

}

if (!visited[i]){

visited[i] = true;

item.add(A[i]);

permutation(res, A, item, visited, pos + 1);

visited[i] = false;

item.remove(item.size() - 1);

}

}

}

private int convert(List<Integer> list){

// if (list.get(0) \* 10 + list.get(1) < 24 && list.get(2) \* 10 + list.get(3) < 60){

return list.get(0) \* 1000 + list.get(1) \* 100 + list.get(2) \* 10 + list.get(3);

// }

// return Integer.MIN\_VALUE;

}

}

95\_Unique.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an integer n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example,

Given n = 3, your program should return all 5 unique BST's shown below.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

DFS

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<TreeNode> generateTrees(int n) {

if (n == 0){

return new ArrayList<TreeNode>();

}

return dfs(1, n);

}

private List<TreeNode> dfs(int start, int end){

List<TreeNode> list = new ArrayList<>();

if (start == end){

list.add(new TreeNode(start));

return list;

}

if (start > end){

list.add(null);

return list;

}

for (int i = start; i <= end; i++){

List<TreeNode> left = dfs(start, i - 1);

List<TreeNode> right = dfs(i + 1, end);

for (TreeNode lnode : left){

for (TreeNode rnode : right){

TreeNode root = new TreeNode(i);

root.left = lnode;

root.right = rnode;

list.add(root);

}

}

}

return list;

}

}

950\_Reveal.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a deck of cards, every card has a unique integer. You can order the deck in any order you want.

Initially, all the cards start face down (unrevealed) in one deck.

Now, you do the following steps repeatedly, until all cards are revealed:

Take the top card of the deck, reveal it, and take it out of the deck.

If there are still cards in the deck, put the next top card of the deck at the bottom of the deck.

If there are still unrevealed cards, go back to step 1. Otherwise, stop.

Return an ordering of the deck that would reveal the cards in increasing order.

The first entry in the answer is considered to be the top of the deck.

Example 1:

Input: [17,13,11,2,3,5,7]

Output: [2,13,3,11,5,17,7]

Explanation:

We get the deck in the order [17,13,11,2,3,5,7] (this order doesn't matter), and reorder it.

After reordering, the deck starts as [2,13,3,11,5,17,7], where 2 is the top of the deck.

We reveal 2, and move 13 to the bottom. The deck is now [3,11,5,17,7,13].

We reveal 3, and move 11 to the bottom. The deck is now [5,17,7,13,11].

We reveal 5, and move 17 to the bottom. The deck is now [7,13,11,17].

We reveal 7, and move 13 to the bottom. The deck is now [11,17,13].

We reveal 11, and move 17 to the bottom. The deck is now [13,17].

We reveal 13, and move 17 to the bottom. The deck is now [17].

We reveal 17.

Since all the cards revealed are in increasing order, the answer is correct.

Note:

1 <= A.length <= 1000

1 <= A[i] <= 10^6

A[i] != A[j] for all i != j

Method: queue simulation

https://leetcode.com/problems/reveal-cards-in-increasing-order/discuss/200526/Java-Queue-Simulation

class Solution {

public int[] deckRevealedIncreasing(int[] deck) {

int n = deck.length;

int[] res = new int[n];

Queue<Integer> queue = new LinkedList<>(); // store the index to simulate the process

Arrays.sort(deck);

for (int i = 0; i < n; i++){

queue.offer(i);

}

for (int i = 0; i < n; i++){

res[queue.poll()] = deck[i];

queue.offer(queue.poll());

}

return res;

}

}

Method 2: reverse queue

https://leetcode.com/problems/reveal-cards-in-increasing-order/discuss/200515/JavaC%2B%2BPython-Simulate-the-Reversed-Process

class Solution {

public int[] deckRevealedIncreasing(int[] deck) {

Arrays.sort(deck);

int n = deck.length;

int[] res = new int[n];

Queue<Integer> queue = new LinkedList<>();

for (int i = n - 1; i >= 0; i--){

if (queue.size() > 0){

queue.offer(queue.poll());

}

queue.offer(deck[i]);

}

for (int i = n-1; i >= 0; i--){

res[i] = queue.poll();

}

return res;

}

}

951\_Flip.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For a binary tree T, we can define a flip operation as follows: choose any node, and swap the left and right child subtrees.

A binary tree X is flip equivalent to a binary tree Y if and only if we can make X equal to Y after some number of flip operations.

Write a function that determines whether two binary trees are flip equivalent. The trees are given by root nodes root1 and root2.

Example 1:

Input: root1 = [1,2,3,4,5,6,null,null,null,7,8], root2 = [1,3,2,null,6,4,5,null,null,null,null,8,7]

Output: true

Explanation: We flipped at nodes with values 1, 3, and 5.

Flipped Trees Diagram

Note:

Each tree will have at most 100 nodes.

Each value in each tree will be a unique integer in the range [0, 99].

Similar as LeetCode 101 Symmetric Tree

class Solution {

public boolean flipEquiv(TreeNode root1, TreeNode root2) {

return isSame(root1, root2);

}

private boolean isSame(TreeNode root1, TreeNode root2){

if (root1 == null && root2 == null){

return true;

}

if (root1 == null || root2 == null){

return false;

}

return (root1.val == root2.val) && (isSame(root1.left, root2.left) && isSame(root1.right, root2.right) ||

isSame(root1.right, root2.left) && isSame(root1.left, root2.right));

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean flipEquiv(TreeNode root1, TreeNode root2) {

if (root1 == null && root2 == null){

return true;

}

if (root1 == null || root2 == null){

return false;

}

return root1.val == root2.val && (flipEquiv(root1.left, root2.left) && flipEquiv(root1.right, root2.right) ||

flipEquiv(root1.right, root2.left) && flipEquiv(root1.left, root2.right));

}

}

952\_Largest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a non-empty array of unique positive integers A, consider the following graph:

There are A.length nodes, labelled A[0] to A[A.length - 1];

There is an edge between A[i] and A[j] if and only if A[i] and A[j] share a common factor greater than 1.

Return the size of the largest connected component in the graph.

Best Solution:

Time complexity: O(N\*sqrt(Max val of A[i]))

Union Find template. The only additional stuff is one hashmap which is used to convert factor to the node index

in A for union.

HashMap: key is the factor, val is the index in A

class Solution {

class UF {

int[] parent;

int[] size;

int max;

public UF (int N){

parent = new int[N];

size = new int[N];

max = 1;

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (x == parent[x]){

return x;

}

return parent[x] = find(parent[x]);

}

public void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

max = Math.max(max, size[rootY]);

}

}

}

public int largestComponentSize(int[] A) {

int N = A.length;

Map<Integer, Integer> map = new HashMap<>();// index is the factor, val is the node index

UF uf = new UF(N);

for (int i = 0; i < N; i++){

int a = A[i];

for (int j = 2; j \* j <= a; j++){

if (a % j == 0){

if (!map.containsKey(j)){

map.put(j, i);

}else{

uf.union(i, map.get(j));

}

if (!map.containsKey(a/j)){

map.put(a/j, i);

}else{

uf.union(i, map.get(a/j));

}

}

}

if (!map.containsKey(a)){//a could be factor too. Don't miss this

map.put(a, i);

}else{

uf.union(i, map.get(a));

}

}

return uf.max;

}

}

Method 1:

Time complexity: O(N\*sqrt(Max val of A[i]))

Union Find template. The only additional stuff is one hashmap factorToNode which is used to convert factor to the node index

in A for union.

class Solution {

class UF {

int[] parent;

int[] size;

int max;

public UF (int N){

parent = new int[N];

size = new int[N];

max = 1;

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (x == parent[x]){

return x;

}

return find(parent[x]);

}

public void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

max = Math.max(max, size[rootY]);

}

}

}

public int largestComponentSize(int[] A) {

int N = A.length;

int maxNum = 0;

for (int a : A){

maxNum = Math.max(maxNum, a);

}

int[] factorToNode = new int[maxNum+1];// index is the factor, val is the node index

Arrays.fill(factorToNode, -1);

UF uf = new UF(N);

for (int i = 0; i < N; i++){

int a = A[i];

for (int j = 2; j \* j <= a; j++){

if (a % j == 0){

if (factorToNode[j] == -1){

factorToNode[j] = i;

}else{

uf.union(i, factorToNode[j]);

}

if (factorToNode[a/j] == -1){

factorToNode[a/j] = i;

}else{

uf.union(i, factorToNode[a/j]);

}

}

}

if (factorToNode[a] == -1){ // the number itself could be one factor too. Don't miss this.

factorToNode[a] = i;

}else{

uf.union(i, factorToNode[a]);

}

}

return uf.max;

}

}

Method 2: calculate gcd or brute force and Union Find, TLE

class Solution {

class UF {

int[] parent;

int[] size;

int count;

public UF (int N){

parent = new int[N];

size = new int[N];

count = N;

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (x == parent[x]){

return x;

}

return find(parent[x]);

}

public void union(int x, int y){

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY){

parent[rootX] = rootY;

size[rootY] += size[rootX];

count--;

}

}

}

public int largestComponentSize(int[] A) {

int N = A.length;

UF uf = new UF(N);

for (int i = 0; i < N; i++){

for (int j = i + 1; j < N; j++){

if (hasCommon(A[i], A[j])){

uf.union(i, j);

}

}

}

int max = 0;

for (int i = 0; i < N; i++){

int root = uf.find(i);

int size = uf.size[root];

max = Math.max(max, size);

}

return max;

}

private boolean hasCommon(int a, int b){

if (a < b){

int temp = a;

a = b;

b = temp;

}

int gcd = 0;

while (b != 0){

gcd = b;

b = a % b;

a = gcd;

}

return gcd > 1;

}

}

953\_Verifying.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In an alien language, surprisingly they also use english lowercase letters, but possibly in a different order. The order of the alphabet is some permutation of lowercase letters.

Given a sequence of words written in the alien language, and the order of the alphabet, return true if and only if the given words are sorted lexicographicaly in this alien language.

Example 1:

Input: words = ["hello","leetcode"], order = "hlabcdefgijkmnopqrstuvwxyz"

Output: true

Explanation: As 'h' comes before 'l' in this language, then the sequence is sorted.

Example 2:

Input: words = ["word","world","row"], order = "worldabcefghijkmnpqstuvxyz"

Output: false

Explanation: As 'd' comes after 'l' in this language, then words[0] > words[1], hence the sequence is unsorted.

Example 3:

Input: words = ["apple","app"], order = "abcdefghijklmnopqrstuvwxyz"

Output: false

Explanation: The first three characters "app" match, and the second string is shorter (in size.) According to lexicographical rules "apple" > "app", because 'l' > '∅', where '∅' is defined as the blank character which is less than any other character (More info).

Note:

1 <= words.length <= 100

1 <= words[i].length <= 20

order.length == 26

All characters in words[i] and order are english lowercase letters.

Method 1:

Time complexity: O(mnk)

class Solution {

public boolean isAlienSorted(String[] words, String order) {

for (int k = 0; k < words.length - 1; k++){

String curr = words[k];

String next = words[k+1];

int i = 0;

int j = 0;

while (i < curr.length() || j < next.length()){

char c = i < curr.length() ? curr.charAt(i) : '0';

char n = j < next.length() ? next.charAt(j) : '0';

if (c == n){

i++;

j++;

}else{

int cInd = order.indexOf(c);

int nInd = order.indexOf(n);

if (nInd == -1){

return false;

}

if (cInd > nInd){

return false;

}

break;

}

}

}

return true;

}

}

954\_Array.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers A with even length, return true if and only if it is possible to reorder it such

that A[2 \* i + 1] = 2 \* A[2 \* i] for every 0 <= i < len(A) / 2.

Example 1:

Input: [3,1,3,6]

Output: false

Example 2:

Input: [2,1,2,6]

Output: false

Example 3:

Input: [4,-2,2,-4]

Output: true

Explanation: We can take two groups, [-2,-4] and [2,4] to form [-2,-4,2,4] or [2,4,-2,-4].

Example 4:

Input: [1,2,4,16,8,4]

Output: false

Note:

0 <= A.length <= 30000

A.length is even

-100000 <= A[i] <= 100000

Method 1:

Time complexity: O(nlogn)

Key: need to use treemap to ensure the logic is in ascending or descending order, otherwise we don't know to check key/2 or key\*2

need sorted array and hashmap, so choose treemap

class Solution {

public boolean canReorderDoubled(int[] A) {

if (A.length == 0){

return true;

}

TreeMap<Integer, Integer> map = new TreeMap<>();

for (int i : A){

map.put(i, map.getOrDefault(i, 0) + 1);

}

for (int key : map.keySet()){

if (map.get(key) == 0){

continue;

}

if (key < 0){

if (map.get(key/2) == null || map.get(key/2) < map.get(key)){

return false;

}else{

map.put(key/2, map.get(key/2) - map.get(key));

}

}else if (key > 0){

if (map.get(key\*2) == null || map.get(key\*2) < map.get(key)){

return false;

}else{

map.put(key\*2, map.get(key\*2) - map.get(key));

}

}else{

if (map.get(key) % 2 != 0){

return false;

}

}

}

return true;

}

}

955\_Delete.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given an array A of N lowercase letter strings, all of the same length.

Now, we may choose any set of deletion indices, and for each string, we delete all the characters in those indices.

For example, if we have an array A = ["abcdef","uvwxyz"] and deletion indices {0, 2, 3}, then the final array after deletions is

["bef","vyz"].

Suppose we chose a set of deletion indices D such that after deletions, the final array has its elements in lexicographic order

(A[0] <= A[1] <= A[2] ... <= A[A.length - 1]).

Return the minimum possible value of D.length.

Example 1:

Input: ["ca","bb","ac"]

Output: 1

Explanation:

After deleting the first column, A = ["a", "b", "c"].

Now A is in lexicographic order (ie. A[0] <= A[1] <= A[2]).

We require at least 1 deletion since initially A was not in lexicographic order, so the answer is 1.

Example 2:

Input: ["xc","yb","za"]

Output: 0

Explanation:

A is already in lexicographic order, so we don't need to delete anything.

Note that the rows of A are not necessarily in lexicographic order:

ie. it is NOT necessarily true that (A[0][0] <= A[0][1] <= ...)

Example 3:

Input: ["zyx","wvu","tsr"]

Output: 3

Explanation:

We have to delete every column.

Note:

1 <= A.length <= 100

1 <= A[i].length <= 100

All Greedy Methods

https://leetcode.com/problems/delete-columns-to-make-sorted-ii/solution/

Method 1:

Instead of thinking about column deletions, let's think about which columns we will keep in the final answer.

Start with no columns kept. For each column, if we could keep it and have a valid answer, keep it - otherwise delete it because

the requirement for next column will be weaker if we keep previous column as we only need to sort it only if previous columns share

the same letter.

Time complexity: O(mn^2)

Space complexity: O(mn)

class Solution {

public int minDeletionSize(String[] A) {

int m = A.length;

int n = A[0].length();

int res = 0;

String[] prev = new String[m];

for (int j = 0; j < n; j++){

String[] curr = Arrays.copyOf(prev, m);

for (int i = 0; i < m; i++){

curr[i] += A[i].charAt(j);

}

if (isSorted(curr)){

prev = curr;

}else{

res++;

}

}

return res;

}

private boolean isSorted(String[] A){

for (int i = 0; i < A.length - 1; i++){

if (A[i].compareTo(A[i+1]) > 0){

return false;

}

}

return true;

}

}

Method 2: Best solution

Time complexity: O(mn)

Space complexity: O(m)

https://leetcode.com/problems/delete-columns-to-make-sorted-ii/discuss/203182/JavaC%2B%2BPython-Greedy-Solution-O(MN)

class Solution {

public int minDeletionSize(String[] A) {

int m = A.length;

int n = A[0].length();

// isSorted[i] = true if and only if A[i] < A[i + 1], that is to say A[i] and A[i + 1] are sorted.

boolean[] isSorted = new boolean[m-1];

int i = 0;

int res = 0;

for (int j = 0; j < n; j++){

for (i = 0; i < m - 1; i++){

if (!isSorted[i] && A[i].charAt(j) > A[i+1].charAt(j)){

res++;

break;

}

}

//update isSorted if the column is selected

if (i == m -1){

for (int k = 0; k < m-1; k++){

if (A[k].charAt(j) < A[k+1].charAt(j)){

isSorted[k] = true;

}

}

}

}

return res;

}

}

class Solution {

public int minDeletionSize(String[] A) {

int res = 0;

int m = A.length;

int n = A[0].length();

boolean[] isSorted = new boolean[m-1];

for (int j = 0; j < n; j++){

boolean updateSort = true;

for (int i = 0; i < m-1; i++){

if (!isSorted[i] && A[i].charAt(j) > A[i+1].charAt(j)){

res++;

updateSort = false;

break;

}

}

if (updateSort){

for (int i = 0; i < m - 1; i++){

if (A[i].charAt(j) < A[i+1].charAt(j)){

isSorted[i] = true;

}

}

}

}

return res;

}

}

956\_Tallest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are installing a billboard and want it to have the largest height. The billboard will have two steel supports, one on each side. Each steel support must be an equal height.

You have a collection of rods which can be welded together. For example, if you have rods of lengths 1, 2, and 3, you can weld them together to make a support of length 6.

Return the largest possible height of your billboard installation. If you cannot support the billboard, return 0.

Example 1:

Input: [1,2,3,6]

Output: 6

Explanation: We have two disjoint subsets {1,2,3} and {6}, which have the same sum = 6.

Example 2:

Input: [1,2,3,4,5,6]

Output: 10

Explanation: We have two disjoint subsets {2,3,5} and {4,6}, which have the same sum = 10.

Example 3:

Input: [1,2]

Output: 0

Explanation: The billboard cannot be supported, so we return 0.

Note:

0 <= rods.length <= 20

1 <= rods[i] <= 1000

The sum of rods is at most 5000.

https://leetcode.com/problems/tallest-billboard/discuss/203756/short-java-solution-with-some-explanation!

Time complexity: O(N\*sum)

dp's key represents the sum of left billboard - the sum of right billboard, the value represents the largest sum of left billboard

dp

class Solution {

public int tallestBillboard(int[] rods) {

if (rods == null || rods.length == 0){

return 0;

}

Map<Integer, Integer> curr = new HashMap<>();//key: the sum of left - the sum of right; value: the largest sum of left

curr.put(0, 0);

for (int rod : rods){

Map<Integer, Integer> prev = new HashMap<>(curr);

for (int diff : prev.keySet()){

curr.put(diff + rod, Math.max(curr.getOrDefault(diff + rod, 0), prev.get(diff) + rod));

curr.put(diff - rod, Math.max(curr.getOrDefault(diff - rod, 0), prev.get(diff)));

}

}

return curr.get(0);

}

}

Method 2: Best solution

Similar as backpack DP problem as Leetcode 416. But here is more challenging because some data point may not be chosen. Moreover,

the dp definition is different. In this question,

dp[i][j] denotes the largest left sum at the case of after using i-th rod and the difference between left sum and right sum is

j - sum of all rods. In other words, j is the difference between left sum and right sum plus sum of all rods

Initially, I want to design dp as i-th rod and difference between left sum and right to be j, however, j could be negative,

use sum of all rods to offset all negative values.

So the answer should be dp[n][sum of all rods].

Time complexity: O(n \* sum)

Space complexity: O(n \* sum)

public int tallestBillboard(int[] rods) {

int sum = 0;

for (int i : rods){

sum += i;

}

int n = rods.length;

int[][] dp = new int[n+1][2\*sum+1];//largest sum of left at i-th rod and difference between

//sum of left and sum of right equals to j-sum

for (int i = 0; i <= n; i++){

Arrays.fill(dp[i], -1);// -1 means the value could not be reached.

}

dp[0][sum] = 0; //it means if there is no rods, the largest left sum could be 0, not -1.

for (int i = 1; i <= n; i++){

for (int j = 0; j <= 2\*sum; j++){

if (j - rods[i-1] >= 0 && dp[i-1][j-rods[i-1]] != -1){//this means we will add next rod (rods[i-1] to the left,

//so the largest left sum should be added by rods[i-1] from previous step

dp[i][j] = Math.max(dp[i][j], dp[i-1][j-rods[i-1]] + rods[i-1]);

}

if (j + rods[i-1] <= 2\*sum && dp[i-1][j+rods[i-1]] != -1){//this means we will add next rod(rods[i-1]) to the right,

//so largest left sum at previous step stays at dp[i-1][j+rods[i-1]]

dp[i][j] = Math.max(dp[i][j], dp[i-1][j+rods[i-1]]);

}

if (dp[i-1][j] != -1){//this means we don't use rods[i-1] but we need ensure

//previous step could be reached, so we can compare.

dp[i][j] = Math.max(dp[i][j], dp[i-1][j]);

}

}

}

return dp[n][sum];

}

957\_Prison.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are 8 prison cells in a row, and each cell is either occupied or vacant.

Each day, whether the cell is occupied or vacant changes according to the following rules:

If a cell has two adjacent neighbors that are both occupied or both vacant, then the cell becomes occupied.

Otherwise, it becomes vacant.

(Note that because the prison is a row, the first and the last cells in the row can't have two adjacent neighbors.)

We describe the current state of the prison in the following way: cells[i] == 1 if the i-th cell is occupied, else cells[i] == 0.

Given the initial state of the prison, return the state of the prison after N days (and N such changes described above.)

Example 1:

Input: cells = [0,1,0,1,1,0,0,1], N = 7

Output: [0,0,1,1,0,0,0,0]

Explanation:

The following table summarizes the state of the prison on each day:

Day 0: [0, 1, 0, 1, 1, 0, 0, 1]

Day 1: [0, 1, 1, 0, 0, 0, 0, 0]

Day 2: [0, 0, 0, 0, 1, 1, 1, 0]

Day 3: [0, 1, 1, 0, 0, 1, 0, 0]

Day 4: [0, 0, 0, 0, 0, 1, 0, 0]

Day 5: [0, 1, 1, 1, 0, 1, 0, 0]

Day 6: [0, 0, 1, 0, 1, 1, 0, 0]

Day 7: [0, 0, 1, 1, 0, 0, 0, 0]

Example 2:

Input: cells = [1,0,0,1,0,0,1,0], N = 1000000000

Output: [0,0,1,1,1,1,1,0]

Note:

cells.length == 8

cells[i] is in {0, 1}

1 <= N <= 10^9

https://leetcode.com/problems/prison-cells-after-n-days/discuss/205684/JavaPython-Find-the-Loop-Mod-14

class Solution {

public int[] prisonAfterNDays(int[] cells, int N) {

int m = cells.length;

Map<String, Integer> map = new HashMap<>();

while (N > 0){

int[] temp = new int[m];

map.put(Arrays.toString(cells), N);

N--;

for (int j = 1; j < m - 1; j++){

temp[j] = cells[j-1] == cells[j+1] ? 1 : 0;

}

cells = temp;

if (map.containsKey(Arrays.toString(cells))){

N %= map.get(Arrays.toString(cells)) - N;

}

}

return cells;

}

}

958\_Check.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, determine if it is a complete binary tree.

Definition of a complete binary tree from Wikipedia:

In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as

far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

Example 1:

Input: [1,2,3,4,5,6]

Output: true

Explanation: Every level before the last is full (ie. levels with node-values {1} and {2, 3}), and all nodes in the last

level ({4, 5, 6}) are as far left as possible.

Example 2:

Input: [1,2,3,4,5,null,7]

Output: false

Explanation: The node with value 7 isn't as far left as possible.

Note:

The tree will have between 1 and 100 nodes.

Method 1: recursion:

O(logn \* logn \* logn \* logn \* logn)

class Solution {

public boolean isCompleteTree(TreeNode root) {

if (root == null){

return true;

}

boolean left = isCompleteTree(root.left);

boolean right = isCompleteTree(root.right);

if (!left || !right){

return false;

}

int leftRight = getRightDepth(root.left);

int rightLeft = getleftDepth(root.right);

if (leftRight < rightLeft){

return false;

}

int leftLeft = getleftDepth(root.left);

int rightRight = getRightDepth(root.right);

if (leftLeft > rightRight + 1){

return false;

}

return true;

}

private int getleftDepth(TreeNode root){

if (root == null){

return 0;

}

return 1 + getleftDepth(root.left);

}

private int getRightDepth(TreeNode root){

if (root == null){

return 0;

}

return 1 + getRightDepth(root.right);

}

}

Method 2: BFS level order

O(n)

class Solution {

public boolean isCompleteTree(TreeNode root) {

if (root == null){

return true;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (queue.peek() != null){

TreeNode node = queue.poll();

queue.offer(node.left);

queue.offer(node.right);

}

while (!queue.isEmpty()){

TreeNode node = queue.poll();

if (node != null){

return false;

}

}

return true;

}

}

Best solution:

class Solution {

public boolean isCompleteTree(TreeNode root) {

if (root == null){

return true;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

boolean isEnd = false;

while (!queue.isEmpty()){

TreeNode node = queue.poll();

if (node == null){

isEnd = true;

}else{

if (isEnd){

return false;

}

queue.offer(node.left);

queue.offer(node.right);

}

}

return true;

}

}

959\_Regions.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a N x N grid composed of 1 x 1 squares, each 1 x 1 square consists of a /, \, or blank space. These characters divide the

square into contiguous regions.

(Note that backslash characters are escaped, so a \ is represented as "\\".)

Return the number of regions.

Example 1:

Input:

[

" /",

"/ "

]

Output: 2

Explanation: The 2x2 grid is as follows:

Example 2:

Input:

[

" /",

" "

]

Output: 1

Explanation: The 2x2 grid is as follows:

Example 3:

Input:

[

"\\/",

"/\\"

]

Output: 4

Explanation: (Recall that because \ characters are escaped, "\\/" refers to \/, and "/\\" refers to /\.)

The 2x2 grid is as follows:

Example 4:

Input:

[

"/\\",

"\\/"

]

Output: 5

Explanation: (Recall that because \ characters are escaped, "/\\" refers to /\, and "\\/" refers to \/.)

The 2x2 grid is as follows:

Example 5:

Input:

[

"//",

"/ "

]

Output: 3

Explanation: The 2x2 grid is as follows:

Note:

1 <= grid.length == grid[0].length <= 30

grid[i][j] is either '/', '\', or ' '.

https://leetcode.com/problems/regions-cut-by-slashes/discuss/205674/C%2B%2B-with-picture-DFS-on-upscaled-grid/209661

Why transfer factor must be greater than 2?

Consider ["//", "/ "]. If you only scale it 2x, it looks like this:

0 1 0 1

1 0 1 0

0 1 0 0

1 0 0 0

Notice the italicized zeroes - if you try to search for zeroes in that space, you'll end up with 3 separate islands.

That's no good; we want it to count as one region. So, try scaling it 3x:

0 0 1 0 0 1

0 1 0 0 1 0

1 0 0 1 0 0

0 0 1 0 0 0

0 1 0 0 0 0

1 0 0 0 0 0

Now the zeroes identify as one island, not 3 - perfect!

Similar as Number of Island after transformation

class Solution {

public int regionsBySlashes(String[] grid) {

int n = grid.length;

int f = 3; //factor must be greater than 2

int[][] matrix = new int[n\*f][n\*f];

//build matrix

for (int i = 0; i < n; i++){

for (int j = 0 ; j < n; j++){

int startX = i \* f;

int startY = j \* f;

if (grid[i].charAt(j) == '/'){

for (int k = 0; k < f; k++){

for (int m = 0; m < f; m++){

if (k + m == f - 1){

matrix[startX+k][startY+m] = 1;

}

}

}

}else if (grid[i].charAt(j) == '\\'){

for (int k = 0; k < f; k++){

for (int m = 0; m < f; m++){

if (k == m){

matrix[startX+k][startY+m] = 1;

}

}

}

}

}

}

//BFS to find ZERO regions

int res = 0;

for (int i = 0; i < n \* f; i++){

for (int j = 0; j < n \* f; j++){

if (matrix[i][j] == 0){

bfs(matrix, i, j);

res++;

}

}

}

return res;

}

private void bfs(int[][] matrix, int x, int y){

int n = matrix.length;

Queue<int[]> queue = new LinkedList<>();

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

queue.offer(new int[]{x, y});

matrix[x][y] = 1;

while (!queue.isEmpty()){

int[] curr = queue.poll();

for (int[] dir : dirs){

int nx = curr[0] + dir[0];

int ny = curr[1] + dir[1];

if (nx >= 0 && nx < n && ny >= 0 && ny < n && matrix[nx][ny] == 0){

queue.offer(new int[]{nx, ny});

matrix[nx][ny] = 1;

}

}

}

}

}

96\_PartitionList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

Have you met this question in a real interview? Yes

Example

Given 1->4->3->2->5->2->null and x = 3,

return 1->2->2->4->3->5->null.

/\*\*

\* Definition for ListNode.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int val) {

\* this.val = val;

\* this.next = null;

\* }

\* }

\*/

public class Solution {

/\*

\* @param head: The first node of linked list

\* @param x: An integer

\* @return: A ListNode

\*/

public ListNode partition(ListNode head, int x) {

if (head == null){

return head;

}

ListNode smallDummy = new ListNode(-1);

ListNode largeDummy = new ListNode(-1);

ListNode small = smallDummy;

ListNode large = largeDummy;

while (head != null){

if (head.val < x){

small.next = head;

small = head;

}else{

large.next = head;

large = head;

}

head = head.next;

}

large.next = null;

small.next = largeDummy.next;

return smallDummy.next;

}

}

96\_Unique.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example,

Given n = 3, there are a total of 5 unique BST's.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

Catalan number:

https://leetcode.com/problems/unique-binary-search-trees/discuss/31666/DP-Solution-in-6-lines-with-explanation.-F(i-n)-G(i-1)-\*-G(n-i)

https://www.geeksforgeeks.org/program-nth-catalan-number/

G(n): the number of unique BST for a sequence of length n.

F(i, n), 1 <= i <= n: the number of unique BST, where the number i is the root of BST, and the sequence ranges from 1 to n.

F(i, n) = G(i-1) \* G(n-i) 1<= i <= n

G(n) = F(1, n) + F(2, n) + ... + F(n, n)

= G(0) \* G(n-1) + G(1) \* G(n-2) + ... + G(n-1) \* G(0)

G(0) = 1, G(1) = 1

Iterative:

class Solution {

public int numTrees(int n) {

int[] dp = new int[n+1];

dp[0] = 1;

dp[1] = 1;

for (int i = 2; i <= n; i++){

for (int j = 1; j <= i; j++){

dp[i] += dp[j-1] \* dp [i-j];

}

}

return dp[n];

}

}

Recursive: (TLE)

class Solution {

public int numTrees(int n) {

if (n <= 1){

return 1;

}

int ans = 0;

for (int i = 1; i <= n; i++){

ans += numTrees(i-1) \* numTrees(n-i);

}

return ans;

}

}

+memorization => accepted

class Solution {

Map<Integer, Integer> map = new HashMap<>();

public int numTrees(int n) {

if (map.containsKey(n)){

return map.get(n);

}

if (n <= 1){

map.put(n, 1);

return 1;

}

int ans = 0;

for (int i = 1; i <= n; i++){

ans += numTrees(i-1) \* numTrees(n-i);

}

map.put(n, ans);

return ans;

}

}

960\_Delete.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given an array A of N lowercase letter strings, all of the same length.

Now, we may choose any set of deletion indices, and for each string, we delete all the characters in those indices.

For example, if we have an array A = ["babca","bbazb"] and deletion indices {0, 1, 4}, then the final array after deletions is

["bc","az"].

Suppose we chose a set of deletion indices D such that after deletions, the final array has every element (row) in lexicographic

order.

For clarity, A[0] is in lexicographic order (ie. A[0][0] <= A[0][1] <= ... <= A[0][A[0].length - 1]), A[1] is in lexicographic

order (ie. A[1][0] <= A[1][1] <= ... <= A[1][A[1].length - 1]), and so on.

Return the minimum possible value of D.length.

Example 1:

Input: ["babca","bbazb"]

Output: 3

Explanation: After deleting columns 0, 1, and 4, the final array is A = ["bc", "az"].

Both these rows are individually in lexicographic order (ie. A[0][0] <= A[0][1] and A[1][0] <= A[1][1]).

Note that A[0] > A[1] - the array A isn't necessarily in lexicographic order.

Example 2:

Input: ["edcba"]

Output: 4

Explanation: If we delete less than 4 columns, the only row won't be lexicographically sorted.

Example 3:

Input: ["ghi","def","abc"]

Output: 0

Explanation: All rows are already lexicographically sorted.

Note:

1 <= A.length <= 100

1 <= A[i].length <= 100

Based onl Longest Increasing Subsequence

Time complexity: O(mn^2)

https://leetcode.com/problems/delete-columns-to-make-sorted-iii/discuss/205679/C++JavaPython-Maximum-Increasing-Subsequence

class Solution {

public int minDeletionSize(String[] A) {

int m = A.length;

int n = A[0].length();

int res = n;

int[] dp = new int[n];

int k = 0;

for (int j = 0; j < n; j++){

dp[j] = 1;

for (int i = 0; i < j; i++){

for (k = 0; k < m; k++){

if (A[k].charAt(i) > A[k].charAt(j)){

break;

}

}

if (k == m){// if all characters in column j are equal or greater than i then i is a valid column

dp[j] = Math.max(dp[j], dp[i]+1);

}

}

res = Math.min(res, n - dp[j]);

}

return res;

}

}

class Solution {

public int minDeletionSize(String[] A) {

int m = A.length;

int n = A[0].length();

int res = n;

int[] dp = new int[n];

for (int j = 0; j < n; j++){

dp[j] = 1;

for (int i = 0; i < j; i++){

boolean validColumn = true;

for (int k = 0; k < m; k++){

if (A[k].charAt(i) > A[k].charAt(j)){

validColumn = false;

break;

}

}

if (validColumn){// if all characters in column j are equal or greater than i then i is a valid column

dp[j] = Math.max(dp[j], dp[i]+1);

}

}

res = Math.min(res, n - dp[j]);

}

return res;

}

}

961\_Repeated.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a array A of size 2N, there are N+1 unique elements, and exactly one of these elements is repeated N times.

Return the element repeated N times.

Example 1:

Input: [1,2,3,3]

Output: 3

Example 2:

Input: [2,1,2,5,3,2]

Output: 2

Example 3:

Input: [5,1,5,2,5,3,5,4]

Output: 5

Note:

4 <= A.length <= 10000

0 <= A[i] < 10000

A.length is even

Method 1:

Time complexity: O(N)

Space complexity: O(N)

class Solution {

public int repeatedNTimes(int[] A) {

Set<Integer> set = new HashSet<>();

for (int i = 0; i < A.length; i++){

if (!set.add(A[i])){

return A[i];

}

}

return -1;

}

}

Method 2:

Time complexity: O(N)

Space complexity: O(1)

https://leetcode.com/problems/n-repeated-element-in-size-2n-array/discuss/208317/C++-2-lines-O(4)-or-O-(1)

The intuition here is that the repeated numbers have to appear either next to each other (A[i] and A[i + 1]), or

alternated (A[i] and A[i + 2]).

The only exception is sequences like [2, 1, 3, 2]. In this case, the result is the last number, so we just return it in the end.

class Solution {

public int repeatedNTimes(int[] A) {

int n = A.length;

if (A[0] == A[n-1]){

return A[0];

}

for (int i = 0; i < n-2; i++){

if (A[i] == A[i+1] || A[i] == A[i+2]){

return A[i];

}

}

return A[n-1];

}

}

962\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of integers, a ramp is a tuple (i, j) for which i < j and A[i] <= A[j]. The width of such a ramp is j - i.

Find the maximum width of a ramp in A. If one doesn't exist, return 0.

Example 1:

Input: [6,0,8,2,1,5]

Output: 4

Explanation:

The maximum width ramp is achieved at (i, j) = (1, 5): A[1] = 0 and A[5] = 5.

Example 2:

Input: [9,8,1,0,1,9,4,0,4,1]

Output: 7

Explanation:

The maximum width ramp is achieved at (i, j) = (2, 9): A[2] = 1 and A[9] = 1.

Note:

2 <= A.length <= 50000

0 <= A[i] <= 50000

Method 1: Brute force (TLE)

Time complexity: O(N^2)

Space complexity: O(1)

class Solution {

public int maxWidthRamp(int[] A) {

int n = A.length;

int res = -1;

for (int i = 0; i < n - 1; i++){

for (int j = n-1; j > i; j--){

if (A[i] <= A[j]){

res = Math.max(res, j - i);

}

}

}

return res == -1 ? 0 : res;

}

}

Method 2: sorting

Time complexity: O(NlogN)

Space complexity: O(N)

class Solution {

class Pair {

int index;

int val;

public Pair (int index, int val){

this.index = index;

this.val = val;

}

}

public int maxWidthRamp(int[] A) {

int N = A.length;

Pair[] pairs = new Pair[N];

for (int i = 0; i < N; i++){

pairs[i] = new Pair(i, A[i]);

}

Arrays.sort(pairs, new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

return p1.val - p2.val;

}

});

//below is the code: for given array, find the largest difference between current element and previous elements

int res = 0;

int min = N;

for (int i = 0; i < N; i++){

res = Math.max(res, pairs[i].index - min);

min = Math.min(min, pairs[i].index);

}

return res;

}

}

Method 3: best solution

class Solution {

public int maxWidthRamp(int[] A) {

int N = A.length;

int[] maxR = new int[N];

int[] minL = new int[N];

minL[0] = A[0];

for (int i = 1; i < N; i++){

minL[i] = Math.min(A[i], minL[i-1]);

}

maxR[N-1] = A[N-1];

for (int j = N-2; j >= 0; j--){

maxR[j] = Math.max(A[j], maxR[j+1]);

}

int i = 0;

int j = 0;

int res = 0;

while (i < N && j < N){

if (minL[i] <= maxR[j]){

res = Math.max(res, j - i);

j++;

}else{

i++;

}

}

return res;

}

}

https://leetcode.com/problems/maximum-width-ramp/discuss/208341/O(N)-JAVA

Actually, no need minL

class Solution {

public int maxWidthRamp(int[] A) {

int N = A.length;

int[] maxR = new int[N];

maxR[N-1] = A[N-1];

for (int j = N-2; j >= 0; j--){

maxR[j] = Math.max(A[j], maxR[j+1]);

}

int i = 0;

int j = 0;

int res = 0;

while (i < N && j < N){

if (A[i] <= maxR[j]){

res = Math.max(res, j - i);

j++;

}else{

i++;

}

}

return res;

}

}

Method 4: monotonic decreasing stack store index

https://leetcode.com/problems/maximum-width-ramp/discuss/208348/JavaC++Python-O(N)-Using-Stack

Time complexity: O(N)

class Solution {

public int maxWidthRamp(int[] A) {

Stack<Integer> stack = new Stack<>();

for (int i = 0; i < A.length; i++){

if(stack.isEmpty() || A[stack.peek()] >= A[i]){

stack.push(i);

}

}

int res = 0;

for (int i = A.length - 1; i > 0; i--){

while (!stack.isEmpty() && A[stack.peek()] <= A[i]){

res = Math.max(res, i - stack.pop());

}

if (stack.isEmpty()){

break;

}

}

return res;

}

}

963\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a set of points in the xy-plane, determine the minimum area of any rectangle formed from these points, with sides not

necessarily parallel to the x and y axes.

If there isn't any rectangle, return 0.

Method 1: Brute force

O(N^3)

class Solution {

public double minAreaFreeRect(int[][] points) {

Set<String> set = new HashSet<>();

for (int[] p : points){

set.add(p[0] + ":" + p[1]);

}

double res = Double.MAX\_VALUE;

for (int[] p1 : points){

for (int[] p2 : points){

if (p1[0] == p2[0] && p1[1] == p2[1]){

continue;

}

for (int[] p3 : points){

if (p1[0] == p3[0] && p1[1] == p3[1] || p2[0] == p3[0] && p2[1] == p3[1]){

continue;

}

//pythagorean theorem

if (dist(p1, p3) + dist(p2, p3) != dist(p1, p2)){

continue;

}

// x4 = x3 + (x2 - x1)

//y4 = y3 + (y2 - y1)

int x = p1[0] + p2[0] - p3[0];

int y = p1[1] + p2[1] - p3[1];

if (!set.contains(x + ":" + y)){

continue;

}

double area = Math.sqrt(dist(p1,p3)) \* Math.sqrt(dist(p2, p3));

res = Math.min(res, area);

}

}

}

return Double.compare(Double.MAX\_VALUE, res) == 0 ? 0 : res;

}

private int dist(int[] p1, int[] p2){

int x = p1[0] - p2[0];

int y = p1[1] - p2[1];

return x\*x + y\*y;

}

}

964\_Least.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a single positive integer x, we will write an expression of the form x (op1) x (op2) x (op3) x ... where each operator

op1, op2, etc. is either addition, subtraction, multiplication, or division (+, -, \*, or /). For example, with x = 3, we might

write 3 \* 3 / 3 + 3 - 3 which is a value of 3.

When writing such an expression, we adhere to the following conventions:

The division operator (/) returns rational numbers.

There are no parentheses placed anywhere.

We use the usual order of operations: multiplication and division happens before addition and subtraction.

It's not allowed to use the unary negation operator (-). For example, "x - x" is a valid expression as it only uses subtraction,

but "-x + x" is not because it uses negation.

We would like to write an expression with the least number of operators such that the expression equals the given target.

Return the least number of operators used.

Example 1:

Input: x = 3, target = 19

Output: 5

Explanation: 3 \* 3 + 3 \* 3 + 3 / 3. The expression contains 5 operations.

Example 2:

Input: x = 5, target = 501

Output: 8

Explanation: 5 \* 5 \* 5 \* 5 - 5 \* 5 \* 5 + 5 / 5. The expression contains 8 operations.

Example 3:

Input: x = 100, target = 100000000

Output: 3

Explanation: 100 \* 100 \* 100 \* 100. The expression contains 3 operations.

Note:

2 <= x <= 100

1 <= target <= 2 \* 10^8

Method: DFS + memo

https://leetcode.com/problems/least-operators-to-express-number/discuss/208445/c++-recursive-easy-to-understand

class Solution {

Map<Integer, Integer> memo = new HashMap<>();

public int leastOpsExpressTarget(int x, int target) {

if (memo.containsKey(target)){

return memo.get(target);

}

if (x > target){

return Math.min(target\*2 - 1, (x-target)\*2);

}

if (x == target){

return 0;

}

int times = 0;

long sum = x;

while (sum < target){

times++;

sum \*= x;

}

if (sum == target){

return times;

}

int minus = Integer.MAX\_VALUE;

if (sum - target < target){

minus = leastOpsExpressTarget(x, (int)(sum - target)) + times;

}

int add = leastOpsExpressTarget(x, (int)(target - (sum/x))) + times - 1;

int res = Math.min(minus, add) + 1;

memo.put(target, res);

return res;

}

}

965\_Univalued.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A binary tree is univalued if every node in the tree has the same value.

Return true if and only if the given tree is univalued.

Method 1: preorder traversal

class Solution {

public boolean isUnivalTree(TreeNode root) {

if (root == null){

return true;

}

Stack<TreeNode> stack = new Stack<>();

stack.push(root);

while (!stack.isEmpty()){

TreeNode node = stack.pop();

if (node.val != root.val){

return false;

}

if (node.right != null){

stack.push(node.right);

}

if (node.left != null){

stack.push(node.left);

}

}

return true;

}

}

Method 2: inorder traversal

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isUnivalTree(TreeNode root) {

Stack<TreeNode> stack = new Stack<>();

TreeNode curr = root;

while (curr != null || !stack.isEmpty()){

while (curr != null){

stack.push(curr);

curr = curr.left;

}

TreeNode node = stack.pop();

if (node.val != root.val){

return false;

}

curr = node.right;

}

return true;

}

}

Method 3: recursion

class Solution {

public boolean isUnivalTree(TreeNode root) {

if (root == null){

return true;

}

return isValid(root, root.val);

}

private boolean isValid(TreeNode node, int val){

if (node == null){

return true;

}

if (node.val != val){

return false;

}

return isValid(node.left, val) && isValid(node.right, val);

}

}

966\_Vowe.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a wordlist, we want to implement a spellchecker that converts a query word into a correct word.

For a given query word, the spell checker handles two categories of spelling mistakes:

Capitalization: If the query matches a word in the wordlist (case-insensitive), then the query word is returned with the same case

as the case in the wordlist.

Example: wordlist = ["yellow"], query = "YellOw": correct = "yellow"

Example: wordlist = ["Yellow"], query = "yellow": correct = "Yellow"

Example: wordlist = ["yellow"], query = "yellow": correct = "yellow"

Vowel Errors: If after replacing the vowels ('a', 'e', 'i', 'o', 'u') of the query word with any vowel individually, it matches a

word in the wordlist (case-insensitive), then the query word is returned with the same case as the match in the wordlist.

Example: wordlist = ["YellOw"], query = "yollow": correct = "YellOw"

Example: wordlist = ["YellOw"], query = "yeellow": correct = "" (no match)

Example: wordlist = ["YellOw"], query = "yllw": correct = "" (no match)

In addition, the spell checker operates under the following precedence rules:

When the query exactly matches a word in the wordlist (case-sensitive), you should return the same word back.

When the query matches a word up to capitlization, you should return the first such match in the wordlist.

When the query matches a word up to vowel errors, you should return the first such match in the wordlist.

If the query has no matches in the wordlist, you should return the empty string.

Given some queries, return a list of words answer, where answer[i] is the correct word for query = queries[i].

Example 1:

Input: wordlist = ["KiTe","kite","hare","Hare"], queries = ["kite","Kite","KiTe","Hare","HARE","Hear","hear","keti","keet","keto"]

Output: ["kite","KiTe","KiTe","Hare","hare","","","KiTe","","KiTe"]

Note:

1 <= wordlist.length <= 5000

1 <= queries.length <= 5000

1 <= wordlist[i].length <= 7

1 <= queries[i].length <= 7

All strings in wordlist and queries consist only of english letters.

https://leetcode.com/problems/vowel-spellchecker/discuss/211189/JavaC++Python-Two-HashMap

class Solution {

public String[] spellchecker(String[] wordlist, String[] queries) {

int n = queries.length;

String[] res = new String[n];

Set<String> set = new HashSet<>();

Map<String, String> lowerMap = new HashMap<>();

Map<String, String> devowelMap = new HashMap<>();

for (String w : wordlist){

set.add(w);

String lower = w.toLowerCase();

String devowel = lower.replaceAll("[aeiou]", "#");

if (!lowerMap.containsKey(lower)){

lowerMap.put(lower, w);

}

if (!devowelMap.containsKey(devowel)){

devowelMap.put(devowel, w);

}

}

for (int i = 0; i < n; i++){

if (set.contains(queries[i])){

res[i] = queries[i];

continue;

}

String lower = queries[i].toLowerCase();

String devowel = lower.replaceAll("[aeiou]", "#");

if (lowerMap.containsKey(lower)){

res[i] = lowerMap.get(lower);

}else if (devowelMap.containsKey(devowel)){

res[i] = devowelMap.get(devowel);

}else{

res[i] = "";

}

}

return res;

}

}

968\_Binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, we install cameras on the nodes of the tree.

Each camera at a node can monitor its parent, itself, and its immediate children.

Calculate the minimum number of cameras needed to monitor all nodes of the tree.

Example 1:

Input: [0,0,null,0,0]

Output: 1

Explanation: One camera is enough to monitor all nodes if placed as shown.

Example 2:

Input: [0,0,null,0,null,0,null,null,0]

Output: 2

Explanation: At least two cameras are needed to monitor all nodes of the tree. The above image shows one of the valid configurations

of camera placement.

Note:

The number of nodes in the given tree will be in the range [1, 1000].

Every node has value 0.

https://leetcode.com/problems/binary-tree-cameras/discuss/211246/C++-Greedy+DFS-O(n)-Time-4ms-with-Explanation

Greedy + post order

Time complexity: O(N)

Space complexity: O(h)

class Solution {

private int count = 0;

private static final int NOT\_MONITOR = 0;

private static final int MONITOR\_BY\_OTHER = 1;

private static final int CAMERA\_HERE = 2;

public int minCameraCover(TreeNode root) {

if (getState(root) == NOT\_MONITOR){

count++;

}

return count;

}

private int getState(TreeNode root){

if (root == null){

return MONITOR\_BY\_OTHER;

}

int left = getState(root.left);

int right = getState(root.right);

if (left == NOT\_MONITOR || right == NOT\_MONITOR){

count++;

return CAMERA\_HERE;

}

if (left == CAMERA\_HERE || right == CAMERA\_HERE){

return MONITOR\_BY\_OTHER;

}

return NOT\_MONITOR;

}

}

969\_Pancake.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A, we can perform a pancake flip: We choose some positive integer k <= A.length, then reverse the order of the first k

elements of A. We want to perform zero or more pancake flips (doing them one after another in succession) to sort the array A.

Return the k-values corresponding to a sequence of pancake flips that sort A. Any valid answer that sorts the array

within 10 \* A.length flips will be judged as correct.

Example 1:

Input: [3,2,4,1]

Output: [4,2,4,3]

Explanation:

We perform 4 pancake flips, with k values 4, 2, 4, and 3.

Starting state: A = [3, 2, 4, 1]

After 1st flip (k=4): A = [1, 4, 2, 3]

After 2nd flip (k=2): A = [4, 1, 2, 3]

After 3rd flip (k=4): A = [3, 2, 1, 4]

After 4th flip (k=3): A = [1, 2, 3, 4], which is sorted.

Example 2:

Input: [1,2,3]

Output: []

Explanation: The input is already sorted, so there is no need to flip anything.

Note that other answers, such as [3, 3], would also be accepted.

Note:

1 <= A.length <= 100

A[i] is a permutation of [1, 2, ..., A.length]

Find the largest number

Flip twice to the tail

Time complexity: O(N^2)

class Solution {

public List<Integer> pancakeSort(int[] A) {

List<Integer> res = new ArrayList<>();

int N = A.length;

for (int i = N; i > 0; i--){

for (int j = 0; j < i; j++){

if (A[j] == i && j != i - 1){

if (j != 0){

res.add(j+1);

flip(A, j);

}

res.add(i);

flip(A, i-1);

break;

}

}

}

return res;

}

private void flip(int[] A, int j){

int start = 0;

int end = j;

while (start < end){

int temp = A[start];

A[start] = A[end];

A[end] = temp;

start++;

end--;

}

}

}

97\_InterleavingString.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

For example,

Given:

s1 = "aabcc",

s2 = "dbbca",

When s3 = "aadbbcbcac", return true.

When s3 = "aadbbbaccc", return false.

https://leetcode.com/problems/interleaving-string/solution/

Time complexity: O(m\*n)

Space complexity: O(m\*n)

class Solution {

public boolean isInterleave(String s1, String s2, String s3) {

if (s3.length() != s1.length() + s2.length()){

return false;

}

int m = s1.length();

int n = s2.length();

boolean[][] dp = new boolean[m+1][n+1];

dp[0][0] = true;

for (int i = 1; i <=m; i++){

dp[i][0] = dp[i-1][0] && s1.charAt(i-1) == s3.charAt(i-1);

}

for (int j = 1;j <= n; j++){

dp[0][j] = dp[0][j-1] && s2.charAt(j-1) == s3.charAt(j-1);

}

for (int i = 1; i <= m; i++){

for (int j = 1; j <= n; j++){

dp[i][j] = dp[i-1][j] && s1.charAt(i-1) == s3.charAt(i+j-1) || dp[i][j-1] && s2.charAt(j-1) == s3.charAt(i+j-1);

}

}

return dp[m][n];

}

}

Rolling array:

time complexity: O(m\*n)

space complexity: O(n)

class Solution {

public boolean isInterleave(String s1, String s2, String s3) {

if (s3.length() != s1.length() + s2.length()){

return false;

}

int m = s1.length();

int n = s2.length();

boolean[][] dp = new boolean[2][n+1];

for (int i = 0; i <= m; i++){

for (int j = 0; j <= n; j++){

if (i == 0 && j == 0){

dp[i][j] = true;

}else if (j==0){

dp[i%2][j] = dp[(i-1)%2][0] &&s1.charAt(i-1) == s3.charAt(i-1);

}else if (i == 0){

dp[i%2][j] = dp[i%2][j-1] && s2.charAt(j-1) == s3.charAt(j-1);

}else{

dp[i%2][j] = dp[(i-1)%2][j] && s1.charAt(i-1) == s3.charAt(i+j-1) || dp[i%2][j-1] && s2.charAt(j-1) == s3.charAt(i+j-1);

}

}

}

return dp[m%2][n];

}

}

970\_Powerful.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two non-negative integers x and y, an integer is powerful if it is equal to x^i + y^j for some integers i >= 0 and j >= 0.

Return a list of all powerful integers that have value less than or equal to bound.

You may return the answer in any order. In your answer, each value should occur at most once.

Example 1:

Input: x = 2, y = 3, bound = 10

Output: [2,3,4,5,7,9,10]

Explanation:

2 = 2^0 + 3^0

3 = 2^1 + 3^0

4 = 2^0 + 3^1

5 = 2^1 + 3^1

7 = 2^2 + 3^1

9 = 2^3 + 3^0

10 = 2^0 + 3^2

Example 2:

Input: x = 3, y = 5, bound = 15

Output: [2,4,6,8,10,14]

Note:

1 <= x <= 100

1 <= y <= 100

0 <= bound <= 10^6

class Solution {

public List<Integer> powerfulIntegers(int x, int y, int bound) {

Set<Integer> res = new HashSet<>();

for (int i = 1; i < bound; i \*= x){

for (int j = 1; j < bound; j \*= y){

if (i + j <= bound){

res.add(i + j);

}

if (y == 1){

break;

}

}

if (x == 1){

break;

}

}

return new ArrayList<>(res);

}

}

971\_Flip.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree with N nodes, each node has a different value from {1, ..., N}.

A node in this binary tree can be flipped by swapping the left child and the right child of that node.

Consider the sequence of N values reported by a preorder traversal starting from the root. Call such a sequence of N values

the voyage of the tree.

(Recall that a preorder traversal of a node means we report the current node's value, then preorder-traverse the left child,

then preorder-traverse the right child.)

Our goal is to flip the least number of nodes in the tree so that the voyage of the tree matches the voyage we are given.

If we can do so, then return a list of the values of all nodes flipped. You may return the answer in any order.

If we cannot do so, then return the list [-1].

Example 1:

Input: root = [1,2], voyage = [2,1]

Output: [-1]

Example 2:

Input: root = [1,2,3], voyage = [1,3,2]

Output: [1]

Example 3:

Input: root = [1,2,3], voyage = [1,2,3]

Output: []

Note:

1 <= N <= 100

https://leetcode.com/problems/flip-binary-tree-to-match-preorder-traversal/discuss/214216/JavaC++Python-DFS-Solution

Method 1: dfs with global value

class Solution {

int i;

public List<Integer> flipMatchVoyage(TreeNode root, int[] voyage) {

List<Integer> res = new ArrayList<>();

return canMatch(root, voyage, res) ? res : Arrays.asList(-1);

}

private boolean canMatch(TreeNode root, int[] voyage, List<Integer> res){

if (root == null){

return true;

}

if (root.val != voyage[i]){

return false;

}

i++;

if (root.left != null && root.left.val != voyage[i]){

res.add(root.val);

return canMatch(root.right, voyage, res) && canMatch(root.left, voyage, res);

}

return canMatch(root.left, voyage, res) && canMatch(root.right, voyage, res);

}

}

class Solution {

int i;

public List<Integer> flipMatchVoyage(TreeNode root, int[] voyage) {

List<Integer> res = new ArrayList<>();

return canMatch(root, voyage, res) ? res : Arrays.asList(-1);

}

private boolean canMatch(TreeNode root, int[] voyage, List<Integer> res){

if (root == null){

return true;

}

if (root.val != voyage[i]){

return false;

}

i++;

if (root.left != null && root.left.val != voyage[i]){

if (root.right != null && root.right.val != voyage[i]){

return false;

}

res.add(root.val);

return canMatch(root.right, voyage, res) && canMatch(root.left, voyage, res);

}

return canMatch(root.left, voyage, res) && canMatch(root.right, voyage, res);

}

}

Method 2: dfs without global val

class Solution {

public List<Integer> flipMatchVoyage(TreeNode root, int[] voyage) {

List<Integer> res = new ArrayList<>();

int[] p = new int[1];

return canMatch(root, voyage, res, p) ? res : Arrays.asList(-1);

}

private boolean canMatch(TreeNode root, int[] voyage, List<Integer> res, int [] pointer){

if (root == null){

return true;

}

if (root.val != voyage[pointer[0]]){

return false;

}

pointer[0]++;

if (root.left != null && root.left.val != voyage[pointer[0]]){

res.add(root.val);

return canMatch(root.right, voyage, res, pointer) && canMatch(root.left, voyage, res, pointer);

}

return canMatch(root.left, voyage, res, pointer) && canMatch(root.right, voyage, res, pointer);

}

}

973\_Closest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have a list of points on the plane. Find the K closest points to the origin (0, 0).

(Here, the distance between two points on a plane is the Euclidean distance.)

You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in.)

Example 1:

Input: points = [[1,3],[-2,2]], K = 1

Output: [[-2,2]]

Explanation:

The distance between (1, 3) and the origin is sqrt(10).

The distance between (-2, 2) and the origin is sqrt(8).

Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

We only want the closest K = 1 points from the origin, so the answer is just [[-2,2]].

Example 2:

Input: points = [[3,3],[5,-1],[-2,4]], K = 2

Output: [[3,3],[-2,4]]

(The answer [[-2,4],[3,3]] would also be accepted.)

Note:

1 <= K <= points.length <= 10000

-10000 < points[i][0] < 10000

-10000 < points[i][1] < 10000

class Solution {

public int[][] kClosest(int[][] points, int K) {

Arrays.sort(points, new Comparator<int[]>(){

public int compare (int[] p1, int[] p2){

return (p1[0] \* p1[0] + p1[1] \* p1[1]) - (p2[0] \* p2[0] + p2[1] \* p2[1]);

}

});

int[][] res = new int[K][2];

for (int i = 0; i < K; i++){

res[i] = points[i];

}

return res;

}

}

Time complexity: O(nlogn)

Space complexity: O(K)

class Solution {

public int[][] kClosest(int[][] points, int K) {

int[][] res = new int[K][2];

Queue<int[]> pq = new PriorityQueue<int[]>(new Comparator<int[]>(){

public int compare (int[] p1, int[] p2){

return (p1[0] \* p1[0] + p1[1] \* p1[1]) - (p2[0] \* p2[0] + p2[1] \* p2[1]);

}

});

for (int[] p : points){

pq.offer(new int[]{p[0], p[1]});

}

int index = 0;

while (!pq.isEmpty() && index < K){

res[index++] = pq.poll();

}

return res;

}

}

974\_Subarray.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of integers, return the number of (contiguous, non-empty) subarrays that have a sum divisible by K.

Example 1:

Input: A = [4,5,0,-2,-3,1], K = 5

Output: 7

Explanation: There are 7 subarrays with a sum divisible by K = 5:

[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

Note:

1 <= A.length <= 30000

-10000 <= A[i] <= 10000

2 <= K <= 10000

Check 523. Continuous Subarray Sum

class Solution {

public int subarraysDivByK(int[] A, int K) {

Map<Integer, Integer> map = new HashMap<>();

map.put(0, 1);

int count = 0;

int sum = 0;

for (int i : A){

sum = (sum + i % K + K) % K;//convert negative remainder to positive by adding K. in Java: -1 % 2 = -1. In Python: -1 % 2 = 1

if (map.containsKey(sum)){

count += map.get(sum);

}

map.put(sum, map.getOrDefault(sum, 0) + 1);

}

return count;

}

}

https://leetcode.com/problems/subarray-sums-divisible-by-k/discuss/217980/Java-O(N)-with-HashMap-and-preSum

class Solution {

public int subarraysDivByK(int[] A, int K) {

Map<Integer, Integer> map = new HashMap<>();

map.put(0, 1);

int count = 0;

int sum = 0;

for (int i : A){

sum = (sum + i)% K;

if (sum < 0){

sum += K;

}

if (map.containsKey(sum)){

count += map.get(sum);

}

map.put(sum, map.getOrDefault(sum, 0) + 1);

}

return count;

}

}

975\_Odd.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

You are given an integer array A. From some starting index, you can make a series of jumps. The (1st, 3rd, 5th, ...) jumps in

the series are called odd numbered jumps, and the (2nd, 4th, 6th, ...) jumps in the series are called even numbered jumps.

You may from index i jump forward to index j (with i < j) in the following way:

During odd numbered jumps (ie. jumps 1, 3, 5, ...), you jump to the index j such that A[i] <= A[j] and A[j] is the smallest

possible value. If there are multiple such indexes j, you can only jump to the smallest such index j.

During even numbered jumps (ie. jumps 2, 4, 6, ...), you jump to the index j such that A[i] >= A[j] and A[j] is the largest

possible value. If there are multiple such indexes j, you can only jump to the smallest such index j.

(It may be the case that for some index i, there are no legal jumps.)

A starting index is good if, starting from that index, you can reach the end of the array (index A.length - 1) by jumping some

number of times (possibly 0 or more than once.)

Return the number of good starting indexes.

Example 1:

Input: [10,13,12,14,15]

Output: 2

Explanation:

From starting index i = 0, we can jump to i = 2 (since A[2] is the smallest among A[1], A[2], A[3], A[4] that is greater or

equal to A[0]), then we can't jump any more.

From starting index i = 1 and i = 2, we can jump to i = 3, then we can't jump any more.

From starting index i = 3, we can jump to i = 4, so we've reached the end.

From starting index i = 4, we've reached the end already.

In total, there are 2 different starting indexes (i = 3, i = 4) where we can reach the end with some number of jumps.

Example 2:

Input: [2,3,1,1,4]

Output: 3

Explanation:

From starting index i = 0, we make jumps to i = 1, i = 2, i = 3:

During our 1st jump (odd numbered), we first jump to i = 1 because A[1] is the smallest value in (A[1], A[2], A[3], A[4])

that is greater than or equal to A[0].

During our 2nd jump (even numbered), we jump from i = 1 to i = 2 because A[2] is the largest value in (A[2], A[3], A[4])

that is less than or equal to A[1]. A[3] is also the largest value, but 2 is a smaller index, so we can only jump to i = 2

and not i = 3.

During our 3rd jump (odd numbered), we jump from i = 2 to i = 3 because A[3] is the smallest value in (A[3], A[4]) that is

greater than or equal to A[2].

We can't jump from i = 3 to i = 4, so the starting index i = 0 is not good.

In a similar manner, we can deduce that:

From starting index i = 1, we jump to i = 4, so we reach the end.

From starting index i = 2, we jump to i = 3, and then we can't jump anymore.

From starting index i = 3, we jump to i = 4, so we reach the end.

From starting index i = 4, we are already at the end.

In total, there are 3 different starting indexes (i = 1, i = 3, i = 4) where we can reach the end with some number of jumps.

Example 3:

Input: [5,1,3,4,2]

Output: 3

Explanation:

We can reach the end from starting indexes 1, 2, and 4.

Note:

1 <= A.length <= 20000

0 <= A[i] < 100000

Method 1: DP

Time complexity: O(N^2)

Space complexity: O(N)

class Solution {

public int oddEvenJumps(int[] A) {

int N = A.length;

boolean[] odd = new boolean[N];

boolean[] even = new boolean[N];

odd[N-1] = true;

even[N-1] = true;

int[] smallMap = new int[N];

int[] largeMap= new int[N];

for (int i = 0; i < N; i++){

smallMap[i] = -1;

largeMap[i] = -1;

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;

for (int j = i+1; j < N;j++){

if (A[j] >= A[i] && A[j] < min){

smallMap[i] = j;

min = A[j];

}

if (A[j] <= A[i] && A[j] > max){

largeMap[i] = j;

max = A[j];

}

}

}

for (int i = N-2; i >= 0; i--){

if (smallMap[i] != -1 && even[smallMap[i]]){

odd[i] = true;

}

if (largeMap[i] != -1 && odd[largeMap[i]]){

even[i] = true;

}

}

int count = 0;

for (int i = 0; i < N; i++){

if (odd[i]){

count++;

}

}

return count;

}

}

Method 2: Treemap + DP Best solution

Time complexity: O(NlogN)

Space complexity: O(N)

Need find the next equal or greater number and index, so choose treemap

class Solution {

public int oddEvenJumps(int[] A) {

int N = A.length;

boolean[] odd = new boolean[N];

boolean[] even = new boolean[N];

odd[N-1] = true;

even[N-1] = true;

TreeMap<Integer, Integer> treemap = new TreeMap<>();

treemap.put(A[N-1], N-1);

int count = 1;

for (int i = N-2; i >= 0; i--){

Integer high = treemap.ceilingKey(A[i]);

Integer low = treemap.floorKey(A[i]);

if (high != null && even[treemap.get(high)]){

odd[i] = true;

}

if (low != null && odd[treemap.get(low)]){

even[i] = true;

}

if (odd[i]){

count++;

}

treemap.put(A[i], i);

}

return count;

}

}

976\_Largest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of positive lengths, return the largest perimeter of a triangle with non-zero area, formed from 3 of these lengths.

If it is impossible to form any triangle of non-zero area, return 0.

Example 1:

Input: [2,1,2]

Output: 5

Example 2:

Input: [1,2,1]

Output: 0

Example 3:

Input: [3,2,3,4]

Output: 10

Example 4:

Input: [3,6,2,3]

Output: 8

Note:

3 <= A.length <= 10000

1 <= A[i] <= 10^6

class Solution {

public int largestPerimeter(int[] A) {

Arrays.sort(A);

int N = A.length;

for (int i = N - 1; i >= 2; i--){

int one = A[i-1];

int two = A[i-2];

if (one + two > A[i]){

return one + two + A[i];

}

}

return 0;

}

}

977\_Squares.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array of integers A sorted in non-decreasing order, return an array of the squares of each number,

also in sorted non-decreasing order.

Example 1:

Input: [-4,-1,0,3,10]

Output: [0,1,9,16,100]

Example 2:

Input: [-7,-3,2,3,11]

Output: [4,9,9,49,121]

Note:

1 <= A.length <= 10000

-10000 <= A[i] <= 10000

A is sorted in non-decreasing order.

class Solution {

public int[] sortedSquares(int[] A) {

int n = A.length;

int index = n-1;

int[] res = new int[n];

int left = 0;

int right = n - 1;

while (left <= right){

if (A[left] \* A[left] > A[right] \* A[right]){

res[index--] = A[left] \* A[left];

left++;

}else{

res[index--] = A[right] \* A[right];

right--;

}

}

return res;

}

}

Best solution:

class Solution {

public int[] sortedSquares(int[] A) {

int n = A.length;

int index = n-1;

int[] res = new int[n];

int left = 0;

int right = n - 1;

for (int p = n-1; p >= 0; p--){

if (Math.abs(A[left]) > Math.abs(A[right])){

res[p] = A[left] \* A[left];

left++;

}else{

res[p] = A[right] \* A[right];

right--;

}

}

return res;

}

}

978\_Longest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A subarray A[i], A[i+1], ..., A[j] of A is said to be turbulent if and only if:

For i <= k < j, A[k] > A[k+1] when k is odd, and A[k] < A[k+1] when k is even;

OR, for i <= k < j, A[k] > A[k+1] when k is even, and A[k] < A[k+1] when k is odd.

That is, the subarray is turbulent if the comparison sign flips between each adjacent pair of elements in the subarray.

Return the length of a maximum size turbulent subarray of A.

Example 1:

Input: [9,4,2,10,7,8,8,1,9]

Output: 5

Explanation: (A[1] > A[2] < A[3] > A[4] < A[5])

Example 2:

Input: [4,8,12,16]

Output: 2

Example 3:

Input: [100]

Output: 1

Note:

1 <= A.length <= 40000

0 <= A[i] <= 10^9

Method 1:

Time complexity: O(N)

Space complexity: O(N)

class Solution {

public int maxTurbulenceSize(int[] A) {

int n = A.length;

if (n == 1){

return 1;

}

int[] dp = new int[n];

dp[1] = A[0] != A[1] ? 2 : 1;

int max = 1;

for (int i = 2; i < n; i++){

if (A[i-2] > A[i-1] && A[i-1] < A[i] || A[i-2] < A[i-1] && A[i-1] > A[i]){

dp[i] = dp[i-1] + 1;

}else{

if (A[i] != A[i-1]){

dp[i] = 2;

}

}

max = Math.max(max, dp[i]);

}

return max;

}

}

https://leetcode.com/problems/longest-turbulent-subarray/discuss/221935/Java-O(N)-time-O(1)-space

https://leetcode.com/problems/longest-turbulent-subarray/discuss/221980/Simplest-O(n)-Java-solution

Method 2:

Time complexity: O(N)

Space complexity: O(1)

class Solution {

public int maxTurbulenceSize(int[] A) {

int n = A.length;

if (n == 1){

return 1;

}

int curr = 2;

int max = 1;

for (int i = 2; i < n; i++){

if (A[i-2] > A[i-1] && A[i-1] < A[i] || A[i-2] < A[i-1] && A[i-1] > A[i]){

curr += 1;

}else{

if (A[i] != A[i-1]){

curr = 2;

}

}

max = Math.max(max, curr);

}

return max;

}

}

979\_Distribute.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the root of a binary tree with N nodes, each node in the tree has node.val coins, and there are N coins total.

In one move, we may choose two adjacent nodes and move one coin from one node to another. (The move may be from parent to child, or from child to parent.)

Return the number of moves required to make every node have exactly one coin.

Basically we are calculating how many times (a cumulative result)

each coin pass the edge (parent-child edge) in order to make the graph balanced.

class Solution {

int res = 0;

public int distributeCoins(TreeNode root) {

coinGivenToParent(root);

return res;

}

private int coinGivenToParent(TreeNode root){

if (root == null){

return 0;

}

int left = coinGivenToParent(root.left);

int right = coinGivenToParent(root.right);

res += Math.abs(left) + Math.abs(right);

return left + right + root.val - 1;

}

}

class Solution {

public int distributeCoins(TreeNode root) {

int[] d = new int[1];

coinGivenToParent(root,d );

return d[0];

}

private int coinGivenToParent(TreeNode root, int[] d){

if (root == null){

return 0;

}

int left = coinGivenToParent(root.left, d);

int right = coinGivenToParent(root.right, d);

d[0] += Math.abs(left) + Math.abs(right);

return left + right + root.val - 1;

}

}

98\_ValidBinarySearchTree.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

iven a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

Example 1:

2

/ \

1 3

Binary tree [2,1,3], return true.

Example 2:

1

/ \

2 3

Binary tree [1,2,3], return false.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

class ResultType{

boolean isValid;

int min;

int max;

public ResultType(boolean isValid, int min, int max){

this.isValid = isValid;

this.min = min;

this.max = max;

}

}

public boolean isValidBST(TreeNode root) {

if (root == null){

return true;

}

ResultType result = helper(root);

return result.isValid;

}

private ResultType helper(TreeNode root){

if (root == null){

return new ResultType(true, Integer.MAX\_VALUE, Integer.MIN\_VALUE);

}

ResultType left = helper(root.left);

ResultType right = helper(root.right);

if (!left.isValid || !right.isValid){

return new ResultType(false, 0, 0);

}

//note that must check root.left != null for the corner case which root.val = Integer.MAX\_VALUE or Integer.MIN\_VALUE

if (root.left != null && left.max >= root.val || root.right != null && right.min <= root.val){

return new ResultType(false, 0, 0);

}

return new ResultType(true, Math.min(left.min, root.val), Math.max(right.max, root.val));

}

}

Method 2: iterative template

https://leetcode.com/problems/validate-binary-search-tree/discuss/32112/Learn-one-iterative-inorder-traversal-apply-it-to-multiple-tree-questions-(Java-Solution)

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isValidBST(TreeNode root) {

Stack<TreeNode> stack = new Stack<>();

TreeNode node = root;

TreeNode prev = null;

while (node != null || !stack.isEmpty()){

while (node != null){

stack.push(node);

node = node.left;

}

TreeNode curr = stack.pop();

if (prev != null && prev.val >= curr.val){

return false;

}

prev = curr;

node = curr.right;

}

return true;

}

}

980\_Unique.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a 2-dimensional grid, there are 4 types of squares:

1 represents the starting square. There is exactly one starting square.

2 represents the ending square. There is exactly one ending square.

0 represents empty squares we can walk over.

-1 represents obstacles that we cannot walk over.

Return the number of 4-directional walks from the starting square to the ending square, that walk over every non-obstacle square exactly once.

Example 1:

Input: [[1,0,0,0],[0,0,0,0],[0,0,2,-1]]

Output: 2

Explanation: We have the following two paths:

1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2)

2. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2)

Example 2:

Input: [[1,0,0,0],[0,0,0,0],[0,0,0,2]]

Output: 4

Explanation: We have the following four paths:

1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2),(2,3)

2. (0,0),(0,1),(1,1),(1,0),(2,0),(2,1),(2,2),(1,2),(0,2),(0,3),(1,3),(2,3)

3. (0,0),(1,0),(2,0),(2,1),(2,2),(1,2),(1,1),(0,1),(0,2),(0,3),(1,3),(2,3)

4. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2),(2,3)

Example 3:

Input: [[0,1],[2,0]]

Output: 0

Explanation:

There is no path that walks over every empty square exactly once.

Note that the starting and ending square can be anywhere in the grid.

Note:

1 <= grid.length \* grid[0].length <= 20

class Solution {

public int uniquePathsIII(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

List<List<Integer>> res = new ArrayList<>();

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

boolean[][] visited = new boolean[m][n];

int start = 0;

int end = 0;

int count = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1){

start = i \* n + j;

visited[i][j] = true;

}

if (grid[i][j] == 2){

end = i \* n + j;

}

if (grid[i][j] == 0){

count++;

}

}

}

dfs(grid, res, new ArrayList<Integer>(), dirs, start, end, visited, count);

return res.size();

}

private void dfs (int[][] grid, List<List<Integer>> res, List<Integer> item, int[][] dirs, int start, int end, boolean[][] visited, int count){

if (start == end){

if (count == item.size() - 1){

List<Integer> cand = new ArrayList<>(item);

cand.remove(cand.size() - 1);// remove the last dest point

res.add(cand);

}

return;

}

int m = grid.length;

int n = grid[0].length;

int x = start / n;

int y = start % n;

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] != -1 && !visited[nx][ny]){

visited[nx][ny] = true;

item.add(nx \* n + ny);

dfs(grid, res, item, dirs, nx \* n + ny, end, visited, count);

visited[nx][ny] = false;

item.remove(item.size() - 1);

}

}

}

}

981\_Time.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Create a timebased key-value store class TimeMap, that supports two operations.

1. set(string key, string value, int timestamp)

Stores the key and value, along with the given timestamp.

2. get(string key, int timestamp)

Returns a value such that set(key, value, timestamp\_prev) was called previously, with timestamp\_prev <= timestamp.

If there are multiple such values, it returns the one with the largest timestamp\_prev.

If there are no values, it returns the empty string ("").

Example 1:

Input: inputs = ["TimeMap","set","get","get","set","get","get"], inputs = [[],["foo","bar",1],["foo",1],["foo",3],["foo","bar2",4],["foo",4],["foo",5]]

Output: [null,null,"bar","bar",null,"bar2","bar2"]

Explanation:

TimeMap kv;

kv.set("foo", "bar", 1); // store the key "foo" and value "bar" along with timestamp = 1

kv.get("foo", 1); // output "bar"

kv.get("foo", 3); // output "bar" since there is no value corresponding to foo at timestamp 3 and timestamp 2, then the only value is at timestamp 1 ie "bar"

kv.set("foo", "bar2", 4);

kv.get("foo", 4); // output "bar2"

kv.get("foo", 5); //output "bar2"

Example 2:

Input: inputs = ["TimeMap","set","set","get","get","get","get","get"], inputs = [[],["love","high",10],["love","low",20],["love",5],["love",10],["love",15],["love",20],["love",25]]

Output: [null,null,null,"","high","high","low","low"]

Note:

All key/value strings are lowercase.

All key/value strings have length in the range [1, 100]

The timestamps for all TimeMap.set operations are strictly increasing.

1 <= timestamp <= 10^7

TimeMap.set and TimeMap.get functions will be called a total of 120000 times (combined) per test case.

class TimeMap {

Map<String, TreeMap<Integer, String>> tm;

/\*\* Initialize your data structure here. \*/

public TimeMap() {

tm = new HashMap<>();

}

public void set(String key, String value, int timestamp) {

if (!tm.containsKey(key)){

tm.put(key, new TreeMap<>());

}

TreeMap<Integer, String> map = tm.get(key);

map.put(timestamp, value);

}

public String get(String key, int timestamp) {

if (!tm.containsKey(key)){

return "";

}

TreeMap<Integer, String> map = tm.get(key);

Integer floor = map.floorKey(timestamp);

if (floor == null){

return "";

}

return map.get(floor);

}

}

/\*\*

\* Your TimeMap object will be instantiated and called as such:

\* TimeMap obj = new TimeMap();

\* obj.set(key,value,timestamp);

\* String param\_2 = obj.get(key,timestamp);

\*/

983\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a country popular for train travel, you have planned some train travelling one year in advance. The days of the year that you will travel is given as an array days. Each day is an integer from 1 to 365.

Train tickets are sold in 3 different ways:

a 1-day pass is sold for costs[0] dollars;

a 7-day pass is sold for costs[1] dollars;

a 30-day pass is sold for costs[2] dollars.

The passes allow that many days of consecutive travel. For example, if we get a 7-day pass on day 2, then we can travel for 7 days: day 2, 3, 4, 5, 6, 7, and 8.

Return the minimum number of dollars you need to travel every day in the given list of days.

Example 1:

Input: days = [1,4,6,7,8,20], costs = [2,7,15]

Output: 11

Explanation:

For example, here is one way to buy passes that lets you travel your travel plan:

On day 1, you bought a 1-day pass for costs[0] = $2, which covered day 1.

On day 3, you bought a 7-day pass for costs[1] = $7, which covered days 3, 4, ..., 9.

On day 20, you bought a 1-day pass for costs[0] = $2, which covered day 20.

In total you spent $11 and covered all the days of your travel.

Example 2:

Input: days = [1,2,3,4,5,6,7,8,9,10,30,31], costs = [2,7,15]

Output: 17

Explanation:

For example, here is one way to buy passes that lets you travel your travel plan:

On day 1, you bought a 30-day pass for costs[2] = $15 which covered days 1, 2, ..., 30.

On day 31, you bought a 1-day pass for costs[0] = $2 which covered day 31.

In total you spent $17 and covered all the days of your travel.

Note:

1 <= days.length <= 365

1 <= days[i] <= 365

days is in strictly increasing order.

costs.length == 3

1 <= costs[i] <= 1000

https://leetcode.com/problems/minimum-cost-for-tickets/discuss/226670/Java-DP-Solution-with-explanation-O(n)

Similar as Coin Change

If no trip on day i, then minCost(i) = minCost(i-1).

minCost(i)=0 for all i ≤ 0.

Otherwise:

If a 1-day pass on day i. In this case, minCost(i) = minCost(i) + costs[0].

If a 7-day pass ending on day i. then : In this case, minCost(i) = min(minCost(i − 7), minCost(i − 6), …, minCost(i − 1)) + costs[1].

But since since minCost is increasing (adding a day never reduces the minCost) hence:

minCost(i) = minCost(i − 7) + costs[2]

Time complexity: O(M\*N)

class Solution {

public int mincostTickets(int[] days, int[] costs) {

int max = days[days.length-1];

boolean[] travelDay = new boolean[max+1];

for (int day : days){

travelDay[day] = true;

}

int[] costDay = {1, 7, 30};

int[] dp = new int[max+1];

for (int i = 1; i <= max; i++){

if (!travelDay[i]){

dp[i] = dp[i-1];

}else{

dp[i] = Integer.MAX\_VALUE;

for (int j = 0; j < costs.length; j++){

int cand = costs[j];

if (i >= costDay[j]){

cand += dp[i - costDay[j]];

}

dp[i] = Math.min(dp[i], cand);

}

}

}

return dp[max];

}

}

Better:

class Solution {

public int mincostTickets(int[] days, int[] costs) {

int n = days.length;

int max = days[n-1];

int[] dp = new int[max+1];

boolean[] travelDay = new boolean[max+1];

for (int day : days){

travelDay[day] = true;

}

int[] costDay = {1, 7, 30};

for (int i = 1; i <= max; i++){

if (!travelDay[i]){

dp[i] = dp[i-1];

}else{

dp[i] = Integer.MAX\_VALUE;

for (int j = 0; j < costDay.length; j++){

if (i >= costDay[j]){

dp[i] = Math.min(dp[i], dp[i-costDay[j]] + costs[j]);

}else{

dp[i] = Math.min(dp[i], costs[j]);

}

}

}

}

return dp[max];

}

}

984\_String.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two integers A and B, return any string S such that:

S has length A + B and contains exactly A 'a' letters, and exactly B 'b' letters;

The substring 'aaa' does not occur in S;

The substring 'bbb' does not occur in S.

Example 1:

Input: A = 1, B = 2

Output: "abb"

Explanation: "abb", "bab" and "bba" are all correct answers.

Example 2:

Input: A = 4, B = 1

Output: "aabaa"

Note:

0 <= A <= 100

0 <= B <= 100

It is guaranteed such an S exists for the given A and B.

class Solution {

public String strWithout3a3b(int A, int B) {

StringBuilder sb = new StringBuilder();

int countA = 0;

int countB = 0;

while (A > 0 && B > 0){

if (A > B){

if (countA < 2){

sb.append('a');

A--;

countA++;

countB = 0;

}else{

sb.append('b');

B--;

countB = 1;

countA = 0;

}

}else if (A < B){

if (countB < 2){

sb.append('b');

B--;

countB++;

countA = 0;

}else{

sb.append('a');

A--;

countA = 1;

countB = 0;

}

}else{

if (countA < 2){

sb.append('a');

countA++;

countB = 0;

A--;

}else {

sb.append('b');

countB++;

countA = 0;

B--;

}

}

}

while (A > 0){

sb.append('a');

A--;

}

while (B > 0){

sb.append('b');

B--;

}

return sb.toString();

}

}

985\_Sum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We have an array A of integers, and an array queries of queries.

For the i-th query val = queries[i][0], index = queries[i][1], we add val to A[index]. Then, the answer to the i-th query is the sum of the even values of A.

(Here, the given index = queries[i][1] is a 0-based index, and each query permanently modifies the array A.)

Return the answer to all queries. Your answer array should have answer[i] as the answer to the i-th query.

Example 1:

Input: A = [1,2,3,4], queries = [[1,0],[-3,1],[-4,0],[2,3]]

Output: [8,6,2,4]

Explanation:

At the beginning, the array is [1,2,3,4].

After adding 1 to A[0], the array is [2,2,3,4], and the sum of even values is 2 + 2 + 4 = 8.

After adding -3 to A[1], the array is [2,-1,3,4], and the sum of even values is 2 + 4 = 6.

After adding -4 to A[0], the array is [-2,-1,3,4], and the sum of even values is -2 + 4 = 2.

After adding 2 to A[3], the array is [-2,-1,3,6], and the sum of even values is -2 + 6 = 4.

Note:

1 <= A.length <= 10000

-10000 <= A[i] <= 10000

1 <= queries.length <= 10000

-10000 <= queries[i][0] <= 10000

0 <= queries[i][1] < A.length

Method 1:

Time complexity: O(N)

Space complexity: O(N)

class Solution {

public int[] sumEvenAfterQueries(int[] A, int[][] queries) {

int n = A.length;

int[] res = new int[queries.length];

int sum = 0;

Set<Integer> set = new HashSet<>();

for (int i = 0; i < n; i++){

if (A[i] % 2 == 0){

sum += A[i];

set.add(i);

}

}

int j = 0;

res[0] = sum;

System.out.println(sum);

for (int[] query : queries){

int index = query[1];

int val = query[0];

if ((A[index] + val) % 2 == 0){

if (set.contains(index)){

sum += val;

res[j++] = sum;

}else{

sum += A[index] + val;

res[j++] = sum;

set.add(index);

}

}else{

if (set.contains(index)){

sum -= A[index];

res[j++] = sum;

set.remove(index);

}else{

res[j++] = sum;

}

}

A[index] += val;

}

return res;

}

}

Method 2: Best

Time complexity: O(N)

Space complexity: O(1)

class Solution {

public int[] sumEvenAfterQueries(int[] A, int[][] queries) {

int n = A.length;

int[] res = new int[queries.length];

int sum = 0;

for (int i = 0; i < n; i++){

if (A[i] % 2 == 0){

sum += A[i];

}

}

int i = 0;

for (int[] query : queries){

int val = query[0];

int index = query[1];

if (A[index] % 2 == 0){

sum -= A[index];

}

A[index] += val;

if (A[index] % 2 == 0){

sum += A[index];

}

res[i++] = sum;

}

return res;

}

}

986\_Interval.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given two lists of closed intervals, each list of intervals is pairwise disjoint and in sorted order.

Return the intersection of these two interval lists.

(Formally, a closed interval [a, b] (with a <= b) denotes the set of real numbers x with a <= x <= b. The intersection of two

closed intervals is a set of real numbers that is either empty, or can be represented as a closed interval. For example, the

intersection of [1, 3] and [2, 4] is [2, 3].)

Method 1: O(((m+n)Log(m+n))

Sweep line similar as meeting room

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

class Pair {

int val;

int flag;

public Pair (int val, int flag){

this.val = val;

this.flag = flag;

}

}

public Interval[] intervalIntersection(Interval[] A, Interval[] B) {

List<Interval> list = new ArrayList<>();

List<Pair> pairs = new ArrayList<>();

for (Interval i : A){

pairs.add(new Pair(i.start, 1));

pairs.add(new Pair(i.end, -1));

}

for (Interval i : B){

pairs.add(new Pair(i.start, 1));

pairs.add(new Pair(i.end, -1));

}

Collections.sort(pairs, new Comparator<Pair>(){

public int compare (Pair p1, Pair p2){

if (p1.val == p2.val){

return p2.flag - p1.flag; //start must go ahead

}

return p1.val - p2.val;

}

});

int active = 0;

int prev = Integer.MIN\_VALUE;

for (Pair p : pairs){

if (p.flag == 1){

active++;

}else{

active--;

}

if (active == 2){

if (prev == Integer.MIN\_VALUE){

prev = p.val;

}

}else if (active == 1){

if (prev != Integer.MIN\_VALUE){

list.add(new Interval(prev, p.val));

prev = Integer.MIN\_VALUE;

}

}

}

Interval[] res = new Interval[list.size()];

for (int i = 0; i < res.length; i++){

res[i] = list.get(i);

}

return res;

}

}

Method 2: best solution

O(m+n)

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

class Solution {

public Interval[] intervalIntersection(Interval[] A, Interval[] B) {

int m = A.length;

int n = B.length;

int i = 0;

int j = 0;

List<Interval> list = new ArrayList<>();

while (i < m && j < n){

Interval a = A[i];

Interval b = B[j];

int startMax = Math.max(a.start, b.start);

int endMin = Math.min(a.end, b.end);

if (startMax <= endMin){

list.add(new Interval(startMax, endMin));

}

if (endMin == a.end){

i++;

}

if (endMin == b.end){

j++;

}

}

Interval[] res = new Interval[list.size()];

for (int k = 0; k < list.size(); k++){

res[k] = list.get(k);

}

return res;

}

}

987\_Vertical.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a binary tree, return the vertical order traversal of its nodes values.

For each node at position (X, Y), its left and right children respectively will be at positions (X-1, Y-1) and (X+1, Y-1).

Running a vertical line from X = -infinity to X = +infinity, whenever the vertical line touches some nodes, we report the values of the nodes in order from top to bottom (decreasing Y coordinates).

If two nodes have the same position, then the value of the node that is reported first is the value that is smaller.

Return an list of non-empty reports in order of X coordinate. Every report will have a list of values of nodes.

Method 1: based on vertical order traversal

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<Integer>> verticalTraversal(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

if (root == null){

return res;

}

Map<Integer, List<Integer>> map = new HashMap<>();

Queue<TreeNode> nodeQ = new LinkedList<>();

Queue<Integer> indexQ = new LinkedList<>();

nodeQ.offer(root);

indexQ.offer(0);

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;

while (!nodeQ.isEmpty()){

int size = nodeQ.size();

Map<Integer, Integer> collision = new HashMap<>(); //not hashset because more than two points could collision

for (int i = 0; i < size; i++){

TreeNode node = nodeQ.poll();

int index = indexQ.poll();

if (!map.containsKey(index)){

map.put(index, new ArrayList<>());

}

if (!collision.containsKey(index)){

map.get(index).add(node.val);

collision.put(index, 1);

}else{

int points = collision.get(index);

List<Integer> list = map.get(index);

Stack<Integer> stack = new Stack<>();

while (points > 0){

int val = list.get(list.size() - 1);

if (node.val >= val){

list.add(node.val);

while (!stack.isEmpty()){

list.add(stack.pop());

}

break;

}else{

stack.push(list.remove(list.size() - 1));

}

points--;

}

if (points == 0){

list.add(node.val);

while (!stack.isEmpty()){

list.add(stack.pop());

}

}

collision.put(index, collision.get(index) + 1);

}

min = Math.min(min, index);

max = Math.max(max, index);

if (node.left != null){

nodeQ.offer(node.left);

indexQ.offer(index-1);

}

if (node.right != null){

nodeQ.offer(node.right);

indexQ.offer(index+1);

}

}

}

for (int i = min; i<= max; i++){

res.add(map.get(i));

}

return res;

}

}

Method 2: DFS Better solution

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<List<Integer>> verticalTraversal(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

if (root == null){

return res;

}

TreeMap<Integer, TreeMap<Integer, TreeSet<Integer>>> mapX = new TreeMap<>();

dfs(root, mapX, 0, 0);

for (int x : mapX.keySet()){

List<Integer> list = new ArrayList<>();

TreeMap<Integer, TreeSet<Integer>> mapY = mapX.get(x);

for (int y : mapY.keySet()){

TreeSet<Integer> set = mapY.get(y);

for (int val : set){

list.add(val);

}

}

res.add(list);

}

return res;

}

private void dfs (TreeNode root, TreeMap<Integer, TreeMap<Integer, TreeSet<Integer>>> mapX, int x, int y){

if (root == null){

return;

}

if (!mapX.containsKey(x)){

mapX.put(x, new TreeMap<>());

}

TreeMap<Integer, TreeSet<Integer>> mapY = mapX.get(x);

if (!mapY.containsKey(y)){

mapY.put(y, new TreeSet<>());

}

TreeSet<Integer> set = mapY.get(y);

set.add(root.val);

dfs(root.left, mapX, x-1, y+1);

dfs(root.right, mapX, x+1, y+1);

}

}

988\_Smallest.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given the root of a binary tree, each node has a value from 0 to 25 representing the letters 'a' to 'z': a value of 0 represents 'a', a value of 1 represents 'b', and so on.

Find the lexicographically smallest string that starts at a leaf of this tree and ends at the root.

(As a reminder, any shorter prefix of a string is lexicographically smaller: for example, "ab" is lexicographically smaller than "aba". A leaf of a node is a node that has no children.)

Method 1: Top Down

class Solution {

String res = "";

public String smallestFromLeaf(TreeNode root) {

if (root == null){

return "";

}

dfs(root, "");

return res;

}

private void dfs(TreeNode root, String str){

if (root == null){

return;

}

if (root.left == null && root.right == null){

str = (char)(root.val + 'a') + str;

if (res.equals("") || str.compareTo(res) < 0){

res = str;// or res = new String(str);

}

return;

}

str = (char)(root.val + 'a') + str;

dfs(root.left, str);

dfs(root.right, str);

}

}

best solution

class Solution {

String res = "";

public String smallestFromLeaf(TreeNode root) {

dfs(root, "");

return res;

}

private void dfs(TreeNode root, String prev){

if (root == null){

return;

}

String curr = (char)('a' + root.val) + prev;

if (root.left == null && root.right == null){//check if it is leaf

if (res.equals("") || curr.compareTo(res) < 0){

res = curr;

}

}

dfs(root.left, curr);

dfs(root.right, curr);

}

}

Method 2: without global

Top Down

class Solution {

public String smallestFromLeaf(TreeNode root) {

if (root == null){

return "";

}

String[] res = new String[1];

dfs(root, "", res);

return res[0];

}

private void dfs(TreeNode root, String str, String[] res){

if (root == null){

return;

}

if (root.left == null && root.right == null){

str = (char)(root.val + 'a') + str;

if (res[0] == null || str.compareTo(res[0]) < 0){

res[0] = str;

}

return;

}

str = (char)(root.val + 'a') + str;

dfs(root.left, str, res);

dfs(root.right, str, res);

}

}

class Solution {

public String smallestFromLeaf(TreeNode root) {

String[] res = new String[1];

res[0] = "";

dfs(root, "", res);

return res[0];

}

private void dfs(TreeNode root, String prev, String[] res){

if (root == null){

return;

}

String curr = (char)('a' + root.val) + prev;

if (root.left == null && root.right == null){//check if it is leaf

if (res[0].equals("") || curr.compareTo(res[0]) < 0){

res[0] = curr;

}

}

dfs(root.left, curr, res);

dfs(root.right, curr, res);

}

}

Methdo 3: Bottom up

class Solution {

public String smallestFromLeaf(TreeNode root) {

return dfs(root);

}

private String dfs(TreeNode root){

if (root == null){

return null;

}

String left = dfs(root.left);

String right =dfs(root.right);

char c = (char)(root.val + 'a');

if (left == null && right == null){

return "" + c;

}else if (left != null && right == null){

return left + c;

}else if (left == null && right != null){

return right + c;

}

if (left.compareTo(right) < 0){

return left + c;

}

return right + c;

}

}

989\_Add.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

For a non-negative integer X, the array-form of X is an array of its digits in left to right order. For example, if X = 1231, then the array form is [1,2,3,1].

Given the array-form A of a non-negative integer X, return the array-form of the integer X+K.

Example 1:

Input: A = [1,2,0,0], K = 34

Output: [1,2,3,4]

Explanation: 1200 + 34 = 1234

Example 2:

Input: A = [2,7,4], K = 181

Output: [4,5,5]

Explanation: 274 + 181 = 455

Example 3:

Input: A = [2,1,5], K = 806

Output: [1,0,2,1]

Explanation: 215 + 806 = 1021

Example 4:

Input: A = [9,9,9,9,9,9,9,9,9,9], K = 1

Output: [1,0,0,0,0,0,0,0,0,0,0]

Explanation: 9999999999 + 1 = 10000000000

Note：

1 <= A.length <= 10000

0 <= A[i] <= 9

0 <= K <= 10000

If A.length > 1, then A[0] != 0

Best solution:

class Solution {

public List<Integer> addToArrayForm(int[] A, int K) {

List<Integer> res = new ArrayList<>();

for (int i = A.length - 1; i >= 0; i--){

res.add(0, (A[i] + K) % 10);

K = (A[i] + K) / 10;

}

while (K > 0){

res.add(0, K%10);

K /= 10;

}

return res;

}

}

class Solution {

public List<Integer> addToArrayForm(int[] A, int K) {

List<Integer> res = new ArrayList<>();

List<Integer> klist = new ArrayList<>();

while (K > 0){

klist.add(0, K%10);

K /= 10;

}

int i = A.length - 1;

int j = klist.size() - 1;

int carry = 0;

while (i >= 0 || j >= 0){

int a = i >= 0 ? A[i] : 0;

int b = j >= 0 ? klist.get(j) : 0;

int val = a + b + carry;

res.add(val%10);

carry = val/10;

i--;

j--;

}

List<Integer> result = new ArrayList<>();

if (carry > 0){

result.add(carry);

}

int n = res.size();

for (int k = 0; k < n; k++){

result.add(res.get(n-1-k));

}

return result;

}

}

99\_Recover.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:

A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

TreeNode lastNode = new TreeNode(Integer.MIN\_VALUE);

TreeNode wrong1 = null;

TreeNode wrong2 = null;

public void recoverTree(TreeNode root) {

// TreeNode wrong1 = new TreeNode(Integer.MIN\_VALUE);

// TreeNode wrong2 = new TreeNode(Integer.MIN\_VALUE); //pass by value is not working

// TreeNode lastNode = new TreeNode(Integer.MIN\_VALUE);

dfs(root);

int temp = wrong1.val;

wrong1.val = wrong2.val;

wrong2.val = temp;

}

private void dfs(TreeNode root){

if (root == null){

return;

}

dfs(root.left);

if (wrong1 == null && root.val < lastNode.val){

wrong1 = lastNode;

}

if (wrong1 != null && root.val < lastNode.val){

wrong2 = root;

}

lastNode = root;

dfs(root.right);

}

}

99\_ReorderList.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a singly linked list L: L0 → L1 → … → Ln-1 → Ln

reorder it to: L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → …

Have you met this question in a real interview? Yes

Example

Given 1->2->3->4->null, reorder it to 1->4->2->3->null.

Challenge

Can you do this in-place without altering the nodes' values?

/\*\*

\* Definition for ListNode.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int val) {

\* this.val = val;

\* this.next = null;

\* }

\* }

\*/

public class Solution {

/\*

\* @param head: The head of linked list.

\* @return: nothing

\*/

public void reorderList(ListNode head) {

if (head == null){

return;

}

ListNode mid = findMiddle(head);

ListNode tail = reverse(mid.next);

mid.next = null;

merge(head, tail);

}

private ListNode findMiddle(ListNode head){

ListNode slow = head;

ListNode fast = head.next;

while (fast != null && fast.next != null){

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

private ListNode reverse(ListNode head){

ListNode prev = null;

ListNode curt = head;

while (curt != null){

ListNode temp = curt.next;

curt.next = prev;

prev = curt;

curt = temp;

}

return prev;

}

private void merge(ListNode head1, ListNode head2){

ListNode lastNode = new ListNode(0);

int index = 0;

while (head1 != null && head2 != null){

if (index %2 == 0){

lastNode.next = head1;

lastNode = head1;

head1 = head1.next;

}else{

lastNode.next = head2;

lastNode = head2;

head2 = head2.next;

}

index++;

}

if (head1 == null){

lastNode.next = head2;

}else{

lastNode.next = head1;

}

}

}

990\_Satisfiability.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array equations of strings that represent relationships between variables, each string equations[i] has length 4 and

takes one of two different forms: "a==b" or "a!=b". Here, a and b are lowercase letters (not necessarily different) that

represent one-letter variable names.

Return true if and only if it is possible to assign integers to variable names so as to satisfy all the given equations.

Example 1:

Input: ["a==b","b!=a"]

Output: false

Explanation: If we assign say, a = 1 and b = 1, then the first equation is satisfied, but not the second. There is no way to assign the variables to satisfy both equations.

Example 2:

Input: ["b==a","a==b"]

Output: true

Explanation: We could assign a = 1 and b = 1 to satisfy both equations.

Example 3:

Input: ["a==b","b==c","a==c"]

Output: true

Example 4:

Input: ["a==b","b!=c","c==a"]

Output: false

Example 5:

Input: ["c==c","b==d","x!=z"]

Output: true

Note:

1 <= equations.length <= 500

equations[i].length == 4

equations[i][0] and equations[i][3] are lowercase letters

equations[i][1] is either '=' or '!'

equations[i][2] is '='

Method 1: Union Find

class Solution {

class UF{

int[] parent;

int[] size;

int count;

public UF(int N){

parent = new int[N];

size = new int[N];

count = N;

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (parent[x] == x){

return x;

}

return parent[x] = find(parent[x]);// path compression

//return find(parent[x]); //not path compression

}

public void union(int a, int b){

int root\_a = find(a);

int root\_b = find(b);

if (root\_a != root\_b){

parent[root\_a] = root\_b;

size[root\_b] += size[root\_a];

count--;

}

}

public boolean connect(int a, int b){

return find(a) == find(b);

}

public int size(){

return count;

}

}

public boolean equationsPossible(String[] equations) {

UF uf = new UF(26);

for (String equation : equations){

int first = (int)(equation.charAt(0) - 'a');

int second = (int)(equation.charAt(3) - 'a');

char equal = equation.charAt(1);

if (equal == '='){

uf.union(first, second);

}

}

for (String equation : equations){

int first = (int)(equation.charAt(0) - 'a');

int second = (int)(equation.charAt(3) - 'a');

char equal = equation.charAt(1);

if (equal == '!'){

if (uf.connect(first, second)){

return false;

}

}

}

return true;

}

}

991\_Broken.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On a broken calculator that has a number showing on its display, we can perform two operations:

Double: Multiply the number on the display by 2, or;

Decrement: Subtract 1 from the number on the display.

Initially, the calculator is displaying the number X.

Return the minimum number of operations needed to display the number Y.

Example 1:

Input: X = 2, Y = 3

Output: 2

Explanation: Use double operation and then decrement operation {2 -> 4 -> 3}.

Example 2:

Input: X = 5, Y = 8

Output: 2

Explanation: Use decrement and then double {5 -> 4 -> 8}.

Example 3:

Input: X = 3, Y = 10

Output: 3

Explanation: Use double, decrement and double {3 -> 6 -> 5 -> 10}.

Example 4:

Input: X = 1024, Y = 1

Output: 1023

Explanation: Use decrement operations 1023 times.

Note:

1 <= X <= 10^9

1 <= Y <= 10^9

https://leetcode.com/problems/broken-calculator/solution/

Think backward

class Solution {

public int brokenCalc(int X, int Y) {

if (X >= Y){

return X - Y;

}

if (Y % 2 == 1){

return brokenCalc(X, Y + 1) + 1;

}

return brokenCalc(X, Y/2) + 1;

}

}

class Solution {

public int brokenCalc(int X, int Y) {

int count = 0;

while (Y > X){

count++;

if (Y % 2 == 1){

Y++;

}else{

Y /= 2;

}

}

return count + X - Y;

}

}

992\_Subarrays.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of positive integers, call a (contiguous, not necessarily distinct) subarray of A good if the number of different

integers in that subarray is exactly K.

(For example, [1,2,3,1,2] has 3 different integers: 1, 2, and 3.)

Return the number of good subarrays of A.

Example 1:

Input: A = [1,2,1,2,3], K = 2

Output: 7

Explanation: Subarrays formed with exactly 2 different integers: [1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2].

Example 2:

Input: A = [1,2,1,3,4], K = 3

Output: 3

Explanation: Subarrays formed with exactly 3 different integers: [1,2,1,3], [2,1,3], [1,3,4].

Note:

1 <= A.length <= 20000

1 <= A[i] <= A.length

1 <= K <= A.length

Intuition:

Write a helper using sliding window,

to get the number of subarrays with at most K distinct elements.

Then f(exactly K) = f(atMost K) - f(atMost K-1).

Time Complexity:

O(N)

class Solution {

public int subarraysWithKDistinct(int[] A, int K) {

return helper(A, K) - helper(A, K-1);

}

private int helper(int[] A, int K){

Map<Integer, Integer> map = new HashMap<>();

int start = 0;

int end = 0;

int count = 0;

int res = 0;

while (end < A.length){

map.put(A[end], map.getOrDefault(A[end], 0) + 1);

if (map.get(A[end]) == 1){

count++;

}

end++;

while (count > K){

if (map.get(A[start]) == 1){

count--;

}

map.put(A[start], map.get(A[start]) - 1);

start++;

}

res += end - start;

}

return res;

}

}

993\_Cousins.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a binary tree, the root node is at depth 0, and children of each depth k node are at depth k+1.

Two nodes of a binary tree are cousins if they have the same depth, but have different parents.

We are given the root of a binary tree with unique values, and the values x and y of two different nodes in the tree.

Return true if and only if the nodes corresponding to the values x and y are cousins.

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isCousins(TreeNode root, int x, int y) {

if (root == null){

return false;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

int size = queue.size();

boolean foundX = false;

boolean foundY = false;

int parentX = -1;

int parentY = -1;

for (int i = 0; i < size; i++){

TreeNode node = queue.poll();

if (node.left != null){

queue.offer(node.left);

if (node.left.val == x){

parentX = node.val;

foundX = true;

}

if (node.left.val == y){

parentY = node.val;

foundY = true;

}

}

if (node.right != null){

queue.offer(node.right);

if (node.right.val == x){

parentX = node.val;

foundX = true;

}

if (node.right.val == y){

parentY = node.val;

foundY = true;

}

}

}

if (foundX && foundY){

if (parentX == parentY){

return false;

}

return true;

}else if (foundX && !foundY || !foundX && foundY){

return false;

}

}

return false;

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isCousins(TreeNode root, int x, int y) {

if (root == null){

return false;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()){

int size = queue.size();

boolean foundX = false;

boolean foundY = false;

TreeNode parentX = null;

TreeNode parentY = null;

for (int i = 0; i < size; i++){

TreeNode curr = queue.poll();

if (curr.left != null){

queue.offer(curr.left);

if (curr.left.val == x){

foundX = true;

parentX = curr;

}

if (curr.left.val == y){

foundY = true;

parentY = curr;

}

}

if (curr.right != null){

queue.offer(curr.right);

if (curr.right.val == x){

foundX = true;

parentX = curr;

}

if (curr.right.val == y){

foundY = true;

parentY = curr;

}

}

if (foundX && foundY){

if (parentX != parentY){

return true;

}else{

return false;

}

}

}

}

return false;

}

}

994\_Rotting.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a given grid, each cell can have one of three values:

the value 0 representing an empty cell;

the value 1 representing a fresh orange;

the value 2 representing a rotten orange.

Every minute, any fresh orange that is adjacent (4-directionally) to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1 instead.

class Solution {

public int orangesRotting(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

Queue<int[]> queue = new LinkedList<>();

boolean hasFresh = false;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 2){

queue.offer(new int[]{i, j});

}

if (grid[i][j] == 1){

hasFresh = true;

}

}

}

if (queue.isEmpty()){

if (hasFresh){

return -1;

}

return 0;

}

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

int count = -1;

while (!queue.isEmpty()){

int size = queue.size();

count++;

for (int i = 0; i < size; i++){

int[] curr = queue.poll();

int x = curr[0];

int y = curr[1];

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] == 1){

queue.offer(new int[]{nx, ny});

grid[nx][ny] = 2;

}

}

}

}

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 1){

return -1;

}

}

}

return count;

}

}

Better version:

class Solution {

public int orangesRotting(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

Queue<int[]> queue = new LinkedList<>();

int freshNum = 0;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (grid[i][j] == 2){

queue.offer(new int[]{i, j});

}

if (grid[i][j] == 1){

freshNum++;

}

}

}

if (freshNum == 0){

return 0;

}

int[][] dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

int count = -1;

while (!queue.isEmpty()){

int size = queue.size();

count++;

for (int i = 0; i < size; i++){

int[] curr = queue.poll();

int x = curr[0];

int y = curr[1];

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

if (nx >= 0 && nx < m && ny >= 0 && ny < n && grid[nx][ny] == 1){

queue.offer(new int[]{nx, ny});

grid[nx][ny] = 2;

freshNum--;

}

}

}

}

return freshNum == 0 ? count : -1;

}

}

995\_Minimum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In an array A containing only 0s and 1s, a K-bit flip consists of choosing a (contiguous) subarray of length K and simultaneously

changing every 0 in the subarray to 1, and every 1 in the subarray to 0.

Return the minimum number of K-bit flips required so that there is no 0 in the array. If it is not possible, return -1.

Example 1:

Input: A = [0,1,0], K = 1

Output: 2

Explanation: Flip A[0], then flip A[2].

Example 2:

Input: A = [1,1,0], K = 2

Output: -1

Explanation: No matter how we flip subarrays of size 2, we can't make the array become [1,1,1].

Example 3:

Input: A = [0,0,0,1,0,1,1,0], K = 3

Output: 3

Explanation:

Flip A[0],A[1],A[2]: A becomes [1,1,1,1,0,1,1,0]

Flip A[4],A[5],A[6]: A becomes [1,1,1,1,1,0,0,0]

Flip A[5],A[6],A[7]: A becomes [1,1,1,1,1,1,1,1]

Note:

1 <= A.length <= 30000

1 <= K <= A.length

Method 1: Greedy

Time complexity: O(N\*K)

Space complexity: O(1)

class Solution {

public int minKBitFlips(int[] A, int K) {

int count = 0;

for (int i = 0; i < A.length; i++){

if (A[i] == 0){

if (i + K > A.length){

return -1;

}

flip(A, i, K);

count++;

}

}

return count;

}

private void flip(int[] A, int start, int K){

for (int i = start; i < start + K; i++){

A[i] ^= 1;

}

}

}

Method 2: Best solution

Time complexity: O(n)

Space complexity: O(1)

Key:

Maintain the flip number (i.e. cur) within the sliding window K

if cur at i is even and A[i] is 0, we need flip

if cur at i is odd and A[i] is 1, we need flip

https://leetcode.com/problems/minimum-number-of-k-consecutive-bit-flips/discuss/238609/JavaC++Python-One-Pass-and-O(1)-Space

class Solution {

public int minKBitFlips(int[] A, int K) {

int res = 0;

int cur = 0;

for (int i = 0; i < A.length; i++){

if (i >= K && A[i-K] == 2){

cur--;

}

if (cur % 2 == A[i]){

if (i + K > A.length){

return -1;

}

A[i] = 2;

res++;

cur++;

}

}

return res;

}

}

996\_Number.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an array A of non-negative integers, the array is squareful if for every pair of adjacent elements, their sum is a perfect square.

Return the number of permutations of A that are squareful. Two permutations A1 and A2 differ if and only if there is some index i such that A1[i] != A2[i].

Example 1:

Input: [1,17,8]

Output: 2

Explanation:

[1,8,17] and [17,8,1] are the valid permutations.

Example 2:

Input: [2,2,2]

Output: 1

Note:

1 <= A.length <= 12

0 <= A[i] <= 1e9

class Solution {

public int numSquarefulPerms(int[] A) {

List<List<Long>> res = new ArrayList<>();

long[] B = new long[A.length];

for (int i = 0; i < A.length; i++){

B[i] = (long)A[i];

}

boolean[] visited = new boolean[A.length];

Arrays.sort(B);

backtrack(res, new ArrayList<>(), B, visited);

return res.size();

}

private void backtrack(List<List<Long>> res, List<Long> item, long[] A, boolean[] visited){

if (item.size() == A.length){

res.add(new ArrayList<>(item));

return;

}

for (int i = 0; i < A.length; i++){

if (i > 0 && A[i] == A[i-1] && !visited[i-1]){

continue;

}

if (visited[i]){

continue;

}

if (item.size() == 0){

visited[i] = true;

item.add(A[i]);

backtrack(res, item, A, visited);

item.remove(item.size() - 1);

visited[i] = false;

}else{

long prev = item.get(item.size() - 1);

long cand = A[i] + prev ;

if (isPerfectSquare(cand)){

visited[i] = true;

item.add(A[i]);

backtrack(res, item, A, visited);

item.remove(item.size() - 1);

visited[i] = false;

}

}

}

}

private boolean isPerfectSquare(long num) {

long start = 0;

long end = num;

long target = num;

while (start <= end){

long mid = start + (end - start) / 2;

if (mid \* mid == target){

return true;

}else if (mid \* mid < target){

start = mid+1;

}else{

end = mid-1;

}

}

return false;

}

}

Shorter version isPerfectSquare

private boolean isPerfectSquare(long num) {

long sqrt = (long)Math.sqrt(num);

return sqrt \* sqrt == num;

}

Method 2: use global variable

class Solution {

int count = 0;

public int numSquarefulPerms(int[] A) {

boolean[] visited = new boolean[A.length];

Arrays.sort(A);

backtrack(0, A, -1, visited);

return count;

}

private void backtrack(int total, int[] A, int lastNum, boolean[] visited){

if (total == A.length){

count++;

return;

}

for (int i = 0; i < A.length; i++){

if (i > 0 && A[i] == A[i-1] && !visited[i-1]){

continue;

}

if (visited[i]){

continue;

}

if (lastNum == -1){

visited[i] = true;

backtrack(total+1, A, A[i], visited);

visited[i] = false;

}else{

int cand = A[i] + lastNum ;

if (isPerfectSquare(cand)){

visited[i] = true;

backtrack(total+1, A, A[i], visited);

visited[i] = false;

}

}

}

}

private boolean isPerfectSquare(int num) {

int sqrt = (int)Math.sqrt(num);

return sqrt \* sqrt == num;

}

}

997\_Find.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In a town, there are N people labelled from 1 to N. There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

The town judge trusts nobody.

Everybody (except for the town judge) trusts the town judge.

There is exactly one person that satisfies properties 1 and 2.

You are given trust, an array of pairs trust[i] = [a, b] representing that the person labelled a trusts the person labelled b.

If the town judge exists and can be identified, return the label of the town judge. Otherwise, return -1.

Example 1:

Input: N = 2, trust = [[1,2]]

Output: 2

Example 2:

Input: N = 3, trust = [[1,3],[2,3]]

Output: 3

Example 3:

Input: N = 3, trust = [[1,3],[2,3],[3,1]]

Output: -1

Example 4:

Input: N = 3, trust = [[1,2],[2,3]]

Output: -1

Example 5:

Input: N = 4, trust = [[1,3],[1,4],[2,3],[2,4],[4,3]]

Output: 3

Note:

1 <= N <= 1000

trust.length <= 10000

trust[i] are all different

trust[i][0] != trust[i][1]

1 <= trust[i][0], trust[i][1] <= N

class Solution {

public int findJudge(int N, int[][] trust) {

Map<Integer, Set<Integer>> map = new HashMap<>();

Set<Integer> set = new HashSet<>();

for (int[] t : trust){

set.add(t[0]);

if (!map.containsKey(t[0])){

map.put(t[0], new HashSet<>());

}

if (!map.containsKey(t[1])){

map.put(t[1], new HashSet<>());

}

map.get(t[1]).add(t[0]);

}

if (map.isEmpty()){

return 1;

}

for (int i = 1; i <= N; i++){

if (set.contains(i)){

continue;

}

if (map.get(i).size() == N-1){

return i;

}

}

return -1;

}

}

Method 2: Similar as find celebrity

https://github.com/optimisea/Leetcode/blob/master/Java/645\_IdentifyCelebrity.java

https://leetcode.com/problems/find-the-town-judge/discuss/242938/JavaC++Python-Directed-Graph

Best solution:

class Solution {

public int findJudge(int N, int[][] trust) {

int[] count = new int[N+1];

for (int[] t : trust){

count[t[0]]--;

count[t[1]]++;

}

for (int i = 1; i <= N; i++){

if (count[i] == N-1){

return i;

}

}

return -1;

}

}

998\_Maximum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We are given the root node of a maximum tree: a tree where every node has a value greater than any other value in its subtree.

Just as in the previous problem, the given tree was constructed from an list A (root = Construct(A)) recursively with the following

Construct(A) routine:

If A is empty, return null.

Otherwise, let A[i] be the largest element of A. Create a root node with value A[i].

The left child of root will be Construct([A[0], A[1], ..., A[i-1]])

The right child of root will be Construct([A[i+1], A[i+2], ..., A[A.length - 1]])

Return root.

Note that we were not given A directly, only a root node root = Construct(A).

Suppose B is a copy of A with the value val appended to it. It is guaranteed that B has unique values.

Return Construct(B).

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode insertIntoMaxTree(TreeNode root, int val) {

if (root == null){

return new TreeNode(val);

}

return dfs(root, null, val);

}

private TreeNode dfs(TreeNode root, TreeNode parent, int val){

if (root == null){

return new TreeNode(val);

}

if (val > root.val){

TreeNode newRoot = new TreeNode(val);

if (parent == null){

newRoot.left = root;

return newRoot;

}

if (root == parent.left){

newRoot.left = root;

parent.left = newRoot;

}else {

newRoot.left = root;

parent.right = newRoot;

}

return newRoot;

}

root.right = insertIntoMaxTree(root.right, val);

return root;

}

}

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode insertIntoMaxTree(TreeNode root, int val) {

if (root == null){

return new TreeNode(val);

}

if (root.val < val){

TreeNode node = new TreeNode(val);

node.left = root;

return node;

}

root.right = insertIntoMaxTree(root.right, val);

return root;

}

}

999\_Available.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On an 8 x 8 chessboard, there is one white rook. There also may be empty squares, white bishops, and black pawns. These are given

as characters 'R', '.', 'B', and 'p' respectively. Uppercase characters represent white pieces, and lowercase characters represent

black pieces.

The rook moves as in the rules of Chess: it chooses one of four cardinal directions (north, east, west, and south), then moves in

that direction until it chooses to stop, reaches the edge of the board, or captures an opposite colored pawn by moving to the

same square it occupies. Also, rooks cannot move into the same square as other friendly bishops.

Return the number of pawns the rook can capture in one move.

class Solution {

public int numRookCaptures(char[][] board) {

int m = board.length;

int n = board[0].length;

int x = -1;

int y = -1;

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

if (board[i][j] == 'R'){

x = i;

y = j;

break;

}

}

}

int count = 0;

int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};

for (int[] dir : dirs){

int nx = x + dir[0];

int ny = y + dir[1];

while (nx >= 0 && nx < m && ny >= 0 && ny < n && board[nx][ny] != 'B'){

if (board[nx][ny] == 'p'){

count++;

break;

}

nx += dir[0];

ny += dir[1];

}

}

return count;

}

}

binary.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

https://leetcode.com/articles/introduction-to-binary-search/

Template for binary search and question similar as 719

https://leetcode.com/problems/find-k-th-smallest-pair-distance/discuss/109082/Approach-the-problem-using-the-%22trial-and-error%22-algorithm

Binary Search related question

786. K-th Smallest Prime Fraction

774 Minimize Max Distance to Gas Station

719. Find K-th Smallest Pair Distance

668. Kth Smallest Number in Multiplication Table

644. Maximum Average Subarray II

378. Kth Smallest Element in a Sorted Matrix

35. Search Insert Position

Must see https://github.com/optimisea/Leetcode/blob/master/Java/35\_Search.java

Template to find the first smallest one:

public int binarySearch(int[] nums, int target) {

int n = nums.length;

int low = 0;

int high = n - 1;

while (low <= high){

int mid = low + (high - low) / 2;

if (nums[mid] <= target){

low = mid + 1;

else{

high = mid - 1;

}

}

return low;

}

35. Search Insert Position

Must see https://github.com/optimisea/Leetcode/blob/master/Java/35\_Search.java

Note that for this template, in most cases target should be between low (inclusive) and high (inclusive)

But there are two corner cases

case 1: target less than low, e.g. [1,2,3] target = 0 == > low will be 1, high will be 2

case 2: target greater than high, e.g. [1,2,3] target = 4 == > low will be 2, high will be 3

public int binarySearch(int[] nums, int target) {

int n = nums.length;

int low = 0;

int high = n - 1;

while (low + 1 < high){

int mid = low + (high - low) / 2;

if (nums[mid] <= target){

low = mid;

else{

high = mid;

}

}

if (nums[low] == target){

return low;

}

return high;

}

Question 719:

class Solution {

public int smallestDistancePair(int[] nums, int k) {

Arrays.sort(nums);

int n = nums.length;

int low = 0;

int high = nums[n-1] - nums[0];

while (low <= high){

int mid = low + (high - low) / 2;

int count = getLessEqual(nums, mid);

if (count <= k - 1){

low = mid + 1;

}else{

high = mid - 1;

}

}

return low;

}

private int getLessEqual(int[] nums, int val){

int res = 0;

for (int i = 0; i < nums.length; i++){

int j = i + 1;

while (j < nums.length && nums[j] - nums[i] <= val){

j++;

}

res += j - i - 1;

}

return res;

}

}

Question 378

class Solution {

public int kthSmallest(int[][] matrix, int k) {

int m = matrix.length;

int n = matrix[0].length;

int low = matrix[0][0];

int high = matrix[m-1][n-1];

while (low + 1 < high){

int mid = low + (high - low) / 2;

int count = getLessEqual(matrix, mid);

if (count <= k - 1){

low = mid;

}else{

high = mid;

}

}

if (getLessEqual(matrix, low) <= k - 1){

return high;

}

return low;

}

private int getLessEqual(int[][] matrix, int val){

int res = 0;

int m = matrix.length;

int n = matrix[0].length;

int i = m - 1;

int j = 0;

while (i >= 0 && j < n){

if (matrix[i][j] > val){

i--;

}else{

res += i + 1;

j++;

}

}

return res;

}

}

combSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// package whatever; // don't place package name!

/\*

题目不难，给你一个menu，是一个map，key是菜名，value是价格，比如

"apple": 3.25,

"chicken": 4.55,

"cake":10.85,

然后给你一个budget，比如7.80.

要你给出所有菜名的combination，总价要正好符合budget，次序不重要，但不能有重复。

比如，如果budget是7.80，他就要求结果是[["apple", "chicken"]]，不能是[["apple", "chicken"],["chicken","apple"]]

比如，如果budget是6.50，他就要求结果是[["apple", "apple"]]

\*/

import java.io.\*;

import java.util.\*;

class MyCode {

public static List<List<String>> menu (Map<String, Double> map, double target){

List<List<String>> res = new ArrayList<>();

String[] names = new String[map.size()];

int index = 0;

for (String key : map.keySet()){

names[index++] = key;

}

backtrack(map, target, res, new ArrayList<>(), names, 0);

return res;

}

private static void backtrack(Map<String, Double> map, double target, List<List<String>> res, List<String> item, String[] names, int start){

if (target <= 0){

if (target == 0)

res.add(new ArrayList<>(item));

return;

}

for (int i = start; i < names.length; i++){

item.add(names[i]);

backtrack(map, target - map.get(names[i]), res, item, names, i);

item.remove(item.size() - 1);

}

}

public static void main (String[] args) {

Map<String, Double> map = new HashMap<>();

map.put("apple", 3.25);

map.put("chickin", 4.55);

map.put("cake", 10.85);

map.put("banana", 3.25);

List<List<String>> res = menu(map, 6.50);

for (List<String> list : res){

System.out.println("list:");

for (String str : list){

System.out.println(str);

}

}

}

}

compareStr.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

https://www.1point3acres.com/bbs/forum.php?mod=viewthread&tid=504403&extra=page%3D1%26filter%3Dsortid%26sortid%3D311%26searchoption%5B3046%5D%5Bvalue%5D%3D2%26searchoption%5B3046%5D%5Btype%5D%3Dradio%26searchoption%5B3109%5D%5Bvalue%5D%3D2%26searchoption%5B3109%5D%5Btype%5D%3Dradio%26sortid%3D311%26orderby%3Ddateline

题目：

写一个compare 方法， 比较两个字符串的大小。字符串可以包含任何的ASCII字符。要求，把字符串中的数字当成一个整体比较， 比如：

s1： abc0123ac

s2: abc5ac

返回 1， s1 > s2. 因为 0123 > 5.

s1: abc555

s2: abcd

返回 1，s1 > s2. 这时候需要比较555和d的ASCII number的大小。. check 1point3acres for more.

s1: abc0123

s2: abc123

返回 0， s1 = s2. 因为0123 = 123

s1: abc097

s2: abca

返回 0， s1 = s2. 因为 a的ASCII number = 97

s1: abc

s2: abcd

返回 -1， s1 < s2. 字典顺序

int compare(String s1, String s2) ;

import java.io.\*;

class MyCode {

public static int compareStr(String S1, String S2){

int i = 0;

int j = 0;

while (i < S1.length() && j < S2.length()){

char c1 = S1.charAt(i);

char c2 = S2.charAt(j);

if (Character.isLetter(c1) && Character.isLetter(c2)){

if (c1 < c2){

return -1;

}else if (c1 > c2){

return 1;

}

i++;

j++;

}else if (Character.isDigit(c1) && Character.isLetter(c2)){

int num1 = 0;

while (i < S1.length() && Character.isDigit(S1.charAt(i))){

num1 = num1 \* 10 + (int)(S1.charAt(i) - '0');

i++;

}

int num2 = (int)(c2);

if (num1 < num2){

return -1;

}else if (num1 > num2){

return 1;

}

j++;

}else if (Character.isLetter(c1) && Character.isDigit(c2)){

int num2 = 0;

while (j < S2.length() && Character.isDigit(S2.charAt(j))){

num2 = num2 \* 10 + (int)(S2.charAt(j) - '0');

j++;

}

int num1 = (int)(c1);

if (num1 < num2){

return -1;

}else if (num1 > num2){

return 1;

}

i++;

}else{

int num1 = 0;

while (i < S1.length() && Character.isDigit(S1.charAt(i))){

num1 = num1 \* 10 + (int)(S1.charAt(i) - '0');

i++;

}

int num2 = 0;

while (j < S2.length() && Character.isDigit(S2.charAt(j))){

num2 = num2 \* 10 + (int)(S2.charAt(j) - '0');

j++;

}

if (num1 < num2){

return -1;

}else if (num1 > num2){

return 1;

}

}

}

if (i == S1.length() && j == S2.length()){

return 0;

}else if (i == S1.length()){

return -1;

}

return 1;

}

public static void main (String[] args) {

String str1 = "abc0123";

String str2 = "abc123";

System.out.println(compareStr(str1, str2));

}

}

Copy (1)1\_TwoSum.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[Error] - File could not be written...

distanceBetweenTwoNodesBST.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find distance between two nodes of BinarySearchTree

Given a list of unique integers, construct the binary tree by given order without rebalancing,

then find out the distance between two nodes..

public class old\_bstDistance {

class TreeNode{

int val;

TreeNode left;

TreeNode right;

public TreeNode(int val){

this.val = val;

}

}

public int bstDistance(int[] values, int node1, int node2){

if (values == null || values.length == 0){

return 0;

}

boolean node1In = false;

boolean node2In = false;

for (int i = 0; i < values.length; i++){

if (values[i] == node1){

node1In = true;

}

if (values[i] == node2){

node2In = true;

}

}

if (!node1In || !node2In){

return 0;

}

TreeNode root = new TreeNode(values[0]);

for (int i = 1; i < values.length; i++){

createBST(root, values[i]);

}

return shortestDist(root, node1, node2);

}

public void createBST(TreeNode root, int value){

if (value < root.val){

if (root.left == null){

root.left = new TreeNode(value);

}else{

createBST(root.left, value);

}

}else{

if (root.right == null){

root.right = new TreeNode(value);

}else{

createBST(root.right, value);

}

}

}

public TreeNode bstLCA(TreeNode root, TreeNode p, TreeNode q){

if (root == null || root == p || root == q){

return root;

}

if (p.val < root.val && root.val < q.val || q.val < root.val && root.val < q.val){

return root;

}else if (root.val > p.val && root.val > q.val){

return bstLCA(root.left, p, q);

}else if (root.val < p.val && root.val < q.val){

return bstLCA(root.right, p, q);

}

return null;

}

public int shortestDist(TreeNode root, int n1, int n2){

if (root == null){

return 0;

}

if (root.val > n1 && root.val > n2){

return shortestDist(root.left, n1, n2);

}else if (root.val < n1 && root.val < n2){

return shortestDist(root.right, n1, n2);

}else{// note that root now becomes the LCA of n1 and n2

return distFromRoot(root, n1) + distFromRoot(root, n2);

}

}

public int distFromRoot(TreeNode root, int n){

if (root == null || root.val == n){

return 0;

}

if (root.val > n){

return 1 + distFromRoot(root.left, n);

}

return 1 + distFromRoot(root.right, n);

}

public static void main(String[] args){

int[] values = {20, 10, 5, 15, 30, 25, 55};

old\_bstDistance result = new old\_bstDistance();

int ans = result.bstDistance(values, 5, 35);

System.out.println(ans);

}

/\*

We have discussed distance between two nodes in binary tree. The time complexity of this solution is O(n)

In case of BST, we can find distance faster. We start from root and for every node, we do following.

If both keys are greater than current node, we move to right child of current node.

If both keys are smaller than current node, we move to left child of current node.

If one keys is smaller and other key is greater, current node is Lowest Common Ancestor (LCA) of two nodes.

We find distances of current node from two keys and return sum of the distances.

Time complexity: O(h) where h is the height of bst

\*/

}

LICS\_2D.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Find the longest increasing continuous subsequence in 2D.

jiuzhang advanced algorithm

memorization search(dynamic programming)

dp[x][y] denotes the longest increasing continous subseuqnce at x, y

public class Solution{

int[][] dp;

int[][] flag;

int n, m;

public in LICSII(int[][] A){

if (A == null || A.length == 0){

return 0;

}

n = A.length;

m = A[0].length;

int ans = 0;

dp = new int[n][m];

flag = new int[n][m];

for (int i = 0; i < n; i++){

for (int j = 0; j < m; j++){

dp[i][j] = search(i, j, A);

ans = Math.max(ans, dp[i][j]);

}

}

return ans;

}

int[] dx = {1, 0, -1, 0};

int[] dy = {0, 1, 0, -1};

private int search(int x, int y, int[][] A){

if (flag[x][y] != 0){

return dp[x][y];

}

int ans = 1;

for (int i = 0; i < dx.length;; i++){

int nx = x + dx[i];

int ny = y + dy[i];

if (nx < A.length && nx >= 0 && 0 <= ny && ny < A[0].length){

if (A[x][y] > A[nx][ny]){

ans = Math.max(ans, search(nx, ny, A) + 1);

}

}

}

flag[x][y] = 1;

dp[x][y] = ans;

return ans;

}

}

linkedInBS.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Binary search 的应用，找比自己大的字母，arr = [a, b, c, d, f, v, z] key = z, ret = a; arr = [a, b, d,

f, v, z], key = b, ret = d

35. Search Insert Position

Must see https://github.com/optimisea/Leetcode/blob/master/Java/35\_Search.java

Method 1:

Note that for this template, in most cases target should be between low (inclusive) and high (inclusive)

But there are two corner cases

case 1: target less than low, e.g. [1,2,3] target = 0 == > low will be 1, high will be 2

case 2: target greater than high, e.g. [1,2,3] target = 4 == > low will be 2, high will be 3

class MyCode {

public static int bs (char[] arr, char key){

int n = arr.length;

int start = 0;

int end = n - 1;

while (start + 1 < end){

int mid = start + (end - start) / 2;

if (arr[mid] == key){

if (mid == n-1){

return arr[0];

}else{

return arr[mid+1];

}

}else if (arr[mid] > key){

end = mid;

}else{

start = mid;

}

}

if (arr[start] > key){

return arr[start];

}else if (arr[end] > key){

return arr[end];

}

return arr[0];

}

public static void main (String[] args) {

char[] arr = {'b', 'd', 's'};

System.out.println((char)bs(arr, 'a'));

}

}

Method 2:

// package whatever; // don't place package name!

import java.io.\*;

class MyCode {

public static int bs (char[] arr, char key){

int n = arr.length;

int start = 0;

int end = n - 1;

while (start <= end){

int mid = start + (end - start) / 2;

if (arr[mid] == key){

if (mid == n-1){

return arr[0];

}else{

return arr[mid+1];

}

}else if (arr[mid] > key){

end = mid-1;

}else{

start = mid+1;

}

}

//start will be the position to be inserted to keep sorted (check leetcode 35)

if (start == n){

return arr[0];

}

if (arr[start] > key){

return arr[start];

}else if (start < n-1){//implicitily arr[start] == key

return arr[start+1];

}

return arr[0];

}

public static void main (String[] args) {

char[] arr = {'b', 'd', 's'};

System.out.println((char)bs(arr, 'b'));

}

}

mergeSort.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Merge Sort Template

493. Reverse Pairs

https://github.com/optimisea/Leetcode/blob/master/Java/493\_Reverse.java

315. Count of Smaller Numbers After Self

https://github.com/optimisea/Leetcode/blob/master/Java/315\_Count.java

Template

class Solution {

int count = 0;

public int reversePairs(int[] nums) {

int[] temp = new int[nums.length];

mergeSort(nums, temp, 0, nums.length - 1);

return count;

}

private void mergeSort(int[] nums, int[] temp, int start, int end){

if (start >= end){

return;

}

int mid = (start + end) / 2;

mergeSort(nums, temp, start, mid);

mergeSort(nums, temp, mid + 1, end);

merge(nums, temp, start, mid, end);

}

private void merge(int[] nums, int[] temp, int start, int mid, int end){

int left = start;

int right = mid + 1;

int index = start;

while (left <= mid && right <= end){

if (nums[left] < nums[right]){

temp[index++] = nums[left++];

}else {

temp[index++] = nums[right++];

}

}

while (left <= mid){

temp[index++] = nums[left++];

}

while (right <= end){

temp[index++] = nums[right++];

}

for (int i = start; i <= end; i++){

nums[i] = temp[i];

}

}

}

montonic.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

https://leetcode.com/problems/sum-of-subarray-minimums/discuss/178876/stack-solution-with-very-detailed-explanation-step-by-step

Some applications of monotone (increase/decrease) stack in leetcode:

Next Greater Element II (a very basic one)

Largest Rectangle in Histogram(almost the same as this problem)

Maximal Rectangle(please do this problem after you solve the above one)

Trapping Rain Water (challenge)

Remove Duplicate Letters(challenge)

Remove K Digits

Create Maximum Number 132 Pattern(challenge, instead of focusing on the elements in the stack, this problem focuses on

the elements poped from the monotone stack)

sliding window maximum(challenge, monotone queue)

Max Chunks To Make Sorted II

Check Leetcode 907 Sum of Subarray Minimums

Better version:

https://leetcode.com/problems/sum-of-subarray-minimums/discuss/170750/C%2B%2BJavaPython-Stack-Solution

class Solution {

public int sumSubarrayMins(int[] A) {

int res = 0, n = A.length, mod = (int)1e9 + 7;

int[] left = new int[n], right = new int[n];

Stack<int[]> s1 = new Stack<>(), s2 = new Stack<>();

for (int i = 0; i < n; ++i) {

int count = 1;

while (!s1.isEmpty() && s1.peek()[0] > A[i])

count += s1.pop()[1];

s1.push(new int[] {A[i], count});

left[i] = count;

}

for (int i = n - 1; i >= 0; --i) {

int count = 1;

while (!s2.isEmpty() && s2.peek()[0] >= A[i]) // use == to handle duplicate elements, only count one time either in left or right

count += s2.pop()[1];

s2.push(new int[] {A[i], count});

right[i] = count;

}

for (int i = 0; i < n; ++i)

res = (res + A[i] \* left[i] \* right[i]) % mod;

return res;

}

}

Best of Best:

Similar as Leetcode 901 Online Stock Span

class Solution {

public int sumSubarrayMins(int[] A) {

int res = 0, n = A.length, mod = (int)1e9 + 7;

int[] left = new int[n], right = new int[n];

Stack<int[]> s1 = new Stack<>(), s2 = new Stack<>();

for (int i = 0; i < n; ++i) {

int count = 1;

while (!s1.isEmpty() && s1.peek()[0] > A[i])

count += s1.pop()[1];

s1.push(new int[] {A[i], count});

left[i] = count;

}

for (int i = n - 1; i >= 0; --i) {

int count = 1;

while (!s2.isEmpty() && s2.peek()[0] >= A[i]) // use == to handle duplicate elements, only count one time either in left or right

count += s2.pop()[1];

s2.push(new int[] {A[i], count});

right[i] = count;

}

for (int i = 0; i < n; ++i)

res = (res + A[i] \* left[i] \* right[i]) % mod;

return res;

}

}

slidingwindow.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

https://leetcode.com/problems/find-all-anagrams-in-a-string/discuss/92007

https://github.com/optimisea/Leetcode/edit/master/Java/159\_Longest.java

class Solution {

public int lengthOfLongestSubstringTwoDistinct(String s) {

if (s == null){

return 0;

}

Map<Character, Integer> map = new HashMap<>();

int start = 0;

int end = 0;

int max = 0;

int count = 0;

int k = 2;

while (end < s.length()){

char charEnd = s.charAt(end);

map.put(charEnd, map.getOrDefault(charEnd, 0) + 1);

if (map.get(charEnd) == 1){

count++;

}

end++;

while (count > k){

char charStart = s.charAt(start);

if (map.get(charStart) == 1){

count--;

}

map.put(charStart, map.get(charStart) - 1);

start++;

}

max = Math.max(max, end - start); // if the question is to ask about longest or max, put check condition here.

}

return max;

}

}

https://github.com/optimisea/Leetcode/blob/master/Java/76\_MinimumWindowSubstring.java

Minimum Window Substring (not like Minimum Window Subsequence which cares about the order of characters)

class Solution {

public String minWindow(String s, String t) {

Map<Character, Integer> map = new HashMap<>();

for (char c : t.toCharArray()){

map.put(c, map.getOrDefault(c, 0) + 1);

}

int count = map.size();

int start = 0;

int end = 0;

int head = 0;

int minLen = Integer.MAX\_VALUE;

while (end < s.length()){

char cEnd = s.charAt(end);

if (map.containsKey(cEnd)){

map.put(cEnd, map.get(cEnd) - 1);

if (map.get(cEnd) == 0){

count--;

}

}

end++;

while (count == 0){

if (end - start < minLen){ //if the question is to ask about the min, put check condition here

minLen = end - start;

head = start;

}

char cStart = s.charAt(start);

if (map.containsKey(cStart)){

if (map.get(cStart) == 0){

count++;

}

map.put(cStart, map.get(cStart) + 1);

}

start++;

}

}

return minLen == Integer.MAX\_VALUE ? "" : s.substring(head, head + minLen);

}

}

trieTemplate.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Trie {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode (){

children = new TrieNode[26];

isEnd = false;

}

}

private TrieNode root;

public Trie (){

root = new TrieNode();

}

public void insert(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

if (node.children[word.charAt(i) - 'a'] == null){

node.children[word.charAt(i) - 'a'] = new TrieNode();

}

node = node.children[word.charAt(i) - 'a'];

}

node.isEnd = true;

}

public boolean startWith(String prefix){

TrieNode node = searchPrefix(prefix);

return node != null;

}

public boolean search(String word){

TrieNode node = searchPrefix(word);

return node != null && node.isEnd;

}

public TrieNode searchPrefix(String prefix){

TrieNode node = root;

for (int i = 0; i < prefix.length(); i++){

if (node.children[prefix.charAt(i) - 'a'] == null){

return null;

}

node = node.children[prefix.charAt(i) - 'a'];

}

return node;

}

}

Refer to Leetcode 212 word search II for Trie + backtracking

Better version:

class Trie {

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode () {

children = new TrieNode[26];

isEnd = false;

}

}

TrieNode root;

public Trie () {

root = new TrieNode();

}

public void insert(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

node.children[c - 'a'] = new TrieNode();

}

node = node.children[c - 'a'];

}

node.isEnd = true;

}

public boolean startWith(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return true;

}

public boolean search(String word){

TrieNode node = root;

for (int i = 0; i < word.length(); i++){

char c = word.charAt(i);

if (node.children[c - 'a'] == null){

return false;

}

node = node.children[c - 'a'];

}

return node.isEnd;

}

}

UnionFindTemplate.java

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Time complexity with path compression: Find: O(1), Union: O(1)

class UF{

int[] parent;

int[] size;

int count;

public UF(int N){

parent = new int[N];

size = new int[N];

count = N;

for (int i = 0; i < N; i++){

parent[i] = i;

size[i] = 1;

}

}

public int find(int x){

if (parent[x] == x){

return x;

}

return parent[x] = find(parent[x]);// path compression

//return find(parent[x]); //not path compression

}

public void union(int a, int b){

int root\_a = find(a);

int root\_b = find(b);

if (root\_a != root\_b){

parent[root\_a] = root\_b;

size[root\_b] += size[root\_a];

count--;

}

}

public boolean connect(int a, int b){

return find(a) == find(b);

}

public int size(){

return count;

}

}