Contents

[1. Array 14](#_Toc38359838)

[1. Find duplicate in the array 14](#_Toc38359839)

[Problem 14](#_Toc38359840)

[Reference 14](#_Toc38359841)

[Approach 14](#_Toc38359842)

[Solution 14](#_Toc38359843)

[Time and space complexity 14](#_Toc38359844)

[2. Find Single element in the array 15](#_Toc38359845)

[Problem 15](#_Toc38359846)

[Reference 15](#_Toc38359847)

[Approach 15](#_Toc38359848)

[Solution 15](#_Toc38359849)

[Time and space complexity 15](#_Toc38359850)

[3. Find Numbers with Even Number of Digits 15](#_Toc38359851)

[Problem 15](#_Toc38359852)

[Reference 16](#_Toc38359853)

[Approach 16](#_Toc38359854)

[Solution 16](#_Toc38359855)

[Time and space complexity 16](#_Toc38359856)

[4. Largest Number 16](#_Toc38359857)

[Problem 16](#_Toc38359858)

[Reference 17](#_Toc38359859)

[Approach 17](#_Toc38359860)

[Solution 17](#_Toc38359861)

[Time and space complexity 18](#_Toc38359862)

[5. Search Insert Position (Binary Search) 18](#_Toc38359863)

[Problem 18](#_Toc38359864)

[Reference 18](#_Toc38359865)

[Approach 18](#_Toc38359866)

[Solution 18](#_Toc38359867)

[Time and space complexity 19](#_Toc38359868)

[6. Count Negative Numbers in a Sorted Matrix 19](#_Toc38359869)

[Problem 19](#_Toc38359870)

[Reference 19](#_Toc38359871)

[Approach 19](#_Toc38359872)

[Solution 19](#_Toc38359873)

[Time and space complexity 20](#_Toc38359874)

[7. Move Zeroes to end of array 20](#_Toc38359875)

[Problem 20](#_Toc38359876)

[Reference 20](#_Toc38359877)

[Approach 20](#_Toc38359878)

[Solution 21](#_Toc38359879)

[Time and space complexity 21](#_Toc38359880)

[8. Two Sum 21](#_Toc38359881)

[Problem 21](#_Toc38359882)

[Reference 21](#_Toc38359883)

[Approach 21](#_Toc38359884)

[Solution 22](#_Toc38359885)

[Time and space complexity 22](#_Toc38359886)

[9. Merge two sorted array 22](#_Toc38359887)

[Problem 22](#_Toc38359888)

[Reference 22](#_Toc38359889)

[Approach 22](#_Toc38359890)

[Solution 22](#_Toc38359891)

[Time and space complexity 23](#_Toc38359892)

[10. Best Time to Buy & Sell Stock 23](#_Toc38359893)

[Problem 23](#_Toc38359894)

[Reference 24](#_Toc38359895)

[Approach 24](#_Toc38359896)

[Solution 24](#_Toc38359897)

[Time and space complexity 24](#_Toc38359898)

[11. Best Time to Buy & Sell Stock II 24](#_Toc38359899)

[Problem 24](#_Toc38359900)

[Reference 25](#_Toc38359901)

[Approach 25](#_Toc38359902)

[Solution 25](#_Toc38359903)

[Time and space complexity 25](#_Toc38359904)

[12. Length of longest contiguous subarray with equal number of 0 and 1. 25](#_Toc38359905)

[Problem 25](#_Toc38359906)

[Reference 26](#_Toc38359907)

[Approach 26](#_Toc38359908)

[Solution 26](#_Toc38359909)

[Time and space complexity 26](#_Toc38359910)

[13. Product of Array Except Self. 27](#_Toc38359911)

[Problem 27](#_Toc38359912)

[Reference 27](#_Toc38359913)

[Approach 27](#_Toc38359914)

[Solution 27](#_Toc38359915)

[Time and space complexity 27](#_Toc38359916)

[14. Minimum Path sum in 2D-Array. 28](#_Toc38359917)

[Problem 28](#_Toc38359918)

[Reference 28](#_Toc38359919)

[Approach 28](#_Toc38359920)

[Solution 28](#_Toc38359921)

[Time and space complexity 28](#_Toc38359922)

[15. Number of Islands in 2-D Array. 29](#_Toc38359923)

[Problem 29](#_Toc38359924)

[Reference 29](#_Toc38359925)

[Approach 29](#_Toc38359926)

[Solution 29](#_Toc38359927)

[Time and space complexity 30](#_Toc38359928)

[2. String 30](#_Toc38359929)

[1. Check if a string is substring of source. (Rabin Karp Algorithm) 30](#_Toc38359930)

[Problem 30](#_Toc38359931)

[Reference 30](#_Toc38359932)

[Approach 30](#_Toc38359933)

[Solution 31](#_Toc38359934)

[Time and space complexity 32](#_Toc38359935)

[2. Anagram 32](#_Toc38359936)

[Problem 32](#_Toc38359937)

[Reference 32](#_Toc38359938)

[Approach 32](#_Toc38359939)

[Solution 32](#_Toc38359940)

[Time and space complexity 33](#_Toc38359941)

[3. Group Anagrams 33](#_Toc38359942)

[Problem 33](#_Toc38359943)

[Reference 33](#_Toc38359944)

[Approach 33](#_Toc38359945)

[Solution 33](#_Toc38359946)

[Time and space complexity 34](#_Toc38359947)

[4. Backspace String Compare 34](#_Toc38359948)

[Problem 34](#_Toc38359949)

[Reference 35](#_Toc38359950)

[Approach 35](#_Toc38359951)

[Solution 35](#_Toc38359952)

[Time and space complexity 36](#_Toc38359953)

[5. Perform String Shifts 36](#_Toc38359954)

[Problem 36](#_Toc38359955)

[Reference 36](#_Toc38359956)

[Approach 36](#_Toc38359957)

[Solution 37](#_Toc38359958)

[Time and space complexity 37](#_Toc38359959)

[6. Check Valid Parenthesis String 37](#_Toc38359960)

[Problem 37](#_Toc38359961)

[Reference 38](#_Toc38359962)

[Approach 38](#_Toc38359963)

[Solution 38](#_Toc38359964)

[Time and space complexity 39](#_Toc38359965)

[3. Math 39](#_Toc38359966)

[1. Palindrome Number 39](#_Toc38359967)

[Problem 39](#_Toc38359968)

[Reference 39](#_Toc38359969)

[Approach 39](#_Toc38359970)

[Solution 39](#_Toc38359971)

[Time and space complexity 39](#_Toc38359972)

[2. Happy Number 40](#_Toc38359973)

[Problem 40](#_Toc38359974)

[Reference 40](#_Toc38359975)

[Approach 40](#_Toc38359976)

[Solution 40](#_Toc38359977)

[Time and space complexity 41](#_Toc38359978)

[4. Linked List 41](#_Toc38359979)

[1. Reverse single linked list (Iterative + recursive) 41](#_Toc38359980)

[Problem 41](#_Toc38359981)

[Reference 41](#_Toc38359982)

[Approach 41](#_Toc38359983)

[Solution 41](#_Toc38359984)

[Time and space complexity 42](#_Toc38359985)

[2. Reverse single linked list recursive 42](#_Toc38359986)

[Problem 42](#_Toc38359987)

[Reference 42](#_Toc38359988)

[Approach 42](#_Toc38359989)

[Solution 42](#_Toc38359990)

[Time and space complexity 42](#_Toc38359991)

[3. Check Palindrome single linked list 42](#_Toc38359992)

[Problem 42](#_Toc38359993)

[Reference 42](#_Toc38359994)

[Approach 42](#_Toc38359995)

[Solution 43](#_Toc38359996)

[Time and space complexity 45](#_Toc38359997)

[4. Find middle element of single linked list 45](#_Toc38359998)

[Problem 45](#_Toc38359999)

[Reference 45](#_Toc38360000)

[Approach 45](#_Toc38360001)

[Solution 45](#_Toc38360002)

[Time and space complexity 46](#_Toc38360003)

[5. Detect cycle in linked list 46](#_Toc38360004)

[Problem 46](#_Toc38360005)

[Reference 46](#_Toc38360006)

[Approach 46](#_Toc38360007)

[Solution 46](#_Toc38360008)

[Time and space complexity 47](#_Toc38360009)

[6. Detect cycle in linked list and return starting node of loop 47](#_Toc38360010)

[Problem 47](#_Toc38360011)

[Reference 47](#_Toc38360012)

[Approach 47](#_Toc38360013)

[Solution 47](#_Toc38360014)

[Time and space complexity 48](#_Toc38360015)

[7. Intersection of Two Linked Lists 48](#_Toc38360016)

[Problem 48](#_Toc38360017)

[Reference 48](#_Toc38360018)

[Approach 48](#_Toc38360019)

[Solution 48](#_Toc38360020)

[Time and space complexity 49](#_Toc38360021)

[8. Delete Linked List Elements 49](#_Toc38360022)

[Problem 49](#_Toc38360023)

[Reference 50](#_Toc38360024)

[Approach 50](#_Toc38360025)

[Solution 50](#_Toc38360026)

[Time and space complexity 50](#_Toc38360027)

[9. Separate odd even nodes 51](#_Toc38360028)

[Problem 51](#_Toc38360029)

[Reference 51](#_Toc38360030)

[Approach 51](#_Toc38360031)

[Solution 51](#_Toc38360032)

[Time and space complexity 52](#_Toc38360033)

[10. Remove Nth Node From End of List 52](#_Toc38360034)

[Problem 52](#_Toc38360035)

[Reference 52](#_Toc38360036)

[Approach 52](#_Toc38360037)

[Solution 52](#_Toc38360038)

[Time and space complexity 53](#_Toc38360039)

[11. Rotate List 53](#_Toc38360040)

[Problem 53](#_Toc38360041)

[Reference 53](#_Toc38360042)

[Approach 54](#_Toc38360043)

[Solution 54](#_Toc38360044)

[Time and space complexity 55](#_Toc38360045)

[12. Swap Nodes in Pairs Recursive 55](#_Toc38360046)

[Problem 55](#_Toc38360047)

[Reference 55](#_Toc38360048)

[Approach 55](#_Toc38360049)

[Solution 55](#_Toc38360050)

[Time and space complexity 55](#_Toc38360051)

[5. Stack 55](#_Toc38360052)

[1. Next greatest element in array 55](#_Toc38360053)

[Problem 55](#_Toc38360054)

[Reference 56](#_Toc38360055)

[Approach 56](#_Toc38360056)

[Solution 56](#_Toc38360057)

[Time and space complexity 57](#_Toc38360058)

[2. Valid Parentheses 57](#_Toc38360059)

[Problem 57](#_Toc38360060)

[Reference 58](#_Toc38360061)

[Approach 58](#_Toc38360062)

[Solution 58](#_Toc38360063)

[Time and space complexity 58](#_Toc38360064)

[3. Implement Queue using Stack 58](#_Toc38360065)

[Problem 58](#_Toc38360066)

[Reference 59](#_Toc38360067)

[Approach 59](#_Toc38360068)

[Solution 59](#_Toc38360069)

[Time and space complexity 60](#_Toc38360070)

[4. Current Maximum element in stack 60](#_Toc38360071)

[Problem 60](#_Toc38360072)

[Reference 60](#_Toc38360073)

[Approach 60](#_Toc38360074)

[Solution 61](#_Toc38360075)

[Time and space complexity 63](#_Toc38360076)

[6. Queue 63](#_Toc38360077)

[1. Implement Stack using queue 63](#_Toc38360078)

[Problem 63](#_Toc38360079)

[Reference 64](#_Toc38360080)

[Approach 64](#_Toc38360081)

[Solution 64](#_Toc38360082)

[Time and space complexity 64](#_Toc38360083)

[7. Sorting 65](#_Toc38360084)

[1. Insertion Sort in array 65](#_Toc38360085)

[Problem 65](#_Toc38360086)

[Reference 65](#_Toc38360087)

[Approach 65](#_Toc38360088)

[Solution 65](#_Toc38360089)

[Time and space complexity 66](#_Toc38360090)

[2. Insertion Sort in Single Linked list 66](#_Toc38360091)

[Problem 66](#_Toc38360092)

[Reference 66](#_Toc38360093)

[Approach 66](#_Toc38360094)

[Solution 66](#_Toc38360095)

[Time and space complexity 67](#_Toc38360096)

[3. Selection Sort in array 67](#_Toc38360097)

[Problem 67](#_Toc38360098)

[Reference 67](#_Toc38360099)

[Approach 67](#_Toc38360100)

[Solution 67](#_Toc38360101)

[Time and space complexity 68](#_Toc38360102)

[4. Selection Sort in Single linked list 68](#_Toc38360103)

[Problem 68](#_Toc38360104)

[Reference 68](#_Toc38360105)

[Approach 68](#_Toc38360106)

[Solution 68](#_Toc38360107)

[Time and space complexity 69](#_Toc38360108)

[5. Bubble Sort in array 69](#_Toc38360109)

[Problem 69](#_Toc38360110)

[Reference 69](#_Toc38360111)

[Approach 69](#_Toc38360112)

[Solution 69](#_Toc38360113)

[Time and space complexity 70](#_Toc38360114)

[6. Bubble Sort in Single Linked list 70](#_Toc38360115)

[Problem 70](#_Toc38360116)

[Reference 70](#_Toc38360117)

[Approach 70](#_Toc38360118)

[Solution 70](#_Toc38360119)

[Time and space complexity 70](#_Toc38360120)

[7. Merge Sort in array 71](#_Toc38360121)

[Problem 71](#_Toc38360122)

[Reference 71](#_Toc38360123)

[Approach 71](#_Toc38360124)

[Solution 72](#_Toc38360125)

[Time and space complexity 73](#_Toc38360126)

[8. Merge Sort in linked list 73](#_Toc38360127)

[Problem 73](#_Toc38360128)

[Reference 73](#_Toc38360129)

[Approach 73](#_Toc38360130)

[Solution 73](#_Toc38360131)

[Time and space complexity 75](#_Toc38360132)

[9. Quick Sort in array 75](#_Toc38360133)

[Problem 75](#_Toc38360134)

[Reference 75](#_Toc38360135)

[Approach 75](#_Toc38360136)

[Solution 76](#_Toc38360137)

[Time and space complexity 77](#_Toc38360138)

[8. Selection problem 77](#_Toc38360139)

[1. Find kth smallest element in array (Quick Select) 77](#_Toc38360140)

[Problem 77](#_Toc38360141)

[Reference 77](#_Toc38360142)

[Approach 77](#_Toc38360143)

[Solution 78](#_Toc38360144)

[Time and space complexity 79](#_Toc38360145)

[9 Bitwise operator 79](#_Toc38360146)

[1. Convert Binary Number in a Linked List to Integer 80](#_Toc38360147)

[Problem 80](#_Toc38360148)

[Reference 81](#_Toc38360149)

[Approach 81](#_Toc38360150)

[Solution 81](#_Toc38360151)

[Time and space complexity 81](#_Toc38360152)

[10. Binary Tree 81](#_Toc38360153)

[1. Pre-Order Traversal Recursive 83](#_Toc38360154)

[Problem 83](#_Toc38360155)

[Reference 83](#_Toc38360156)

[Approach 83](#_Toc38360157)

[Solution 83](#_Toc38360158)

[Time and space complexity 83](#_Toc38360159)

[2. Pre-Order Traversal Iterative 83](#_Toc38360160)

[Problem 83](#_Toc38360161)

[Reference 83](#_Toc38360162)

[Approach 84](#_Toc38360163)

[Solution 84](#_Toc38360164)

[Time and space complexity 84](#_Toc38360165)

[3. In-Order Traversal Recursive 84](#_Toc38360166)

[Problem 84](#_Toc38360167)

[Reference 84](#_Toc38360168)

[Approach 84](#_Toc38360169)

[Solution 85](#_Toc38360170)

[Time and space complexity 85](#_Toc38360171)

[4. In-Order Traversal Iterative 85](#_Toc38360172)

[Problem 85](#_Toc38360173)

[Reference 85](#_Toc38360174)

[Approach 85](#_Toc38360175)

[Solution 85](#_Toc38360176)

[Time and space complexity 85](#_Toc38360177)

[5. Post-Order Traversal Recursive 85](#_Toc38360178)

[Problem 85](#_Toc38360179)

[Reference 86](#_Toc38360180)

[Approach 86](#_Toc38360181)

[Solution 86](#_Toc38360182)

[Time and space complexity 86](#_Toc38360183)

[10. Post-Order Traversal Iterative 86](#_Toc38360184)

[Problem 86](#_Toc38360185)

[Reference 86](#_Toc38360186)

[Approach 86](#_Toc38360187)

[Solution 87](#_Toc38360188)

[Time and space complexity 87](#_Toc38360189)

[11. Level-Order Traversal 87](#_Toc38360190)

[Problem 87](#_Toc38360191)

[Reference 87](#_Toc38360192)

[Approach 87](#_Toc38360193)

[Solution 87](#_Toc38360194)

[Time and space complexity 88](#_Toc38360195)

[12. Zig-Zag Order Traversal Iterative 88](#_Toc38360196)

[Problem 88](#_Toc38360197)

[Reference 88](#_Toc38360198)

[Approach 88](#_Toc38360199)

[Solution 88](#_Toc38360200)

[Time and space complexity 89](#_Toc38360201)

[9. Binary Tree Tilt 89](#_Toc38360202)

[Problem 89](#_Toc38360203)

[Reference 90](#_Toc38360204)

[Approach 90](#_Toc38360205)

[Solution 90](#_Toc38360206)

[Time and space complexity 90](#_Toc38360207)

[11. Number of nodes in binary tree 90](#_Toc38360208)

[Problem 90](#_Toc38360209)

[Reference 91](#_Toc38360210)

[Approach 91](#_Toc38360211)

[Solution 91](#_Toc38360212)

[Time and space complexity 91](#_Toc38360213)

[12. Number of leaves in binary tree 91](#_Toc38360214)

[Problem 91](#_Toc38360215)

[Reference 91](#_Toc38360216)

[Approach 91](#_Toc38360217)

[Solution 91](#_Toc38360218)

[Time and space complexity 92](#_Toc38360219)

[13. Number of Full nodes in binary tree 92](#_Toc38360220)

[Problem 92](#_Toc38360221)

[Reference 92](#_Toc38360222)

[Approach 92](#_Toc38360223)

[Solution 92](#_Toc38360224)

[Time and space complexity 92](#_Toc38360225)

[14. Maximum Depth/height in binary tree 92](#_Toc38360226)

[Problem 92](#_Toc38360227)

[Reference 92](#_Toc38360228)

[Approach 92](#_Toc38360229)

[Solution 93](#_Toc38360230)

[Time and space complexity 94](#_Toc38360231)

[15. Minimum Depth in binary tree iteratively 94](#_Toc38360232)

[Problem 94](#_Toc38360233)

[Reference 94](#_Toc38360234)

[Approach 94](#_Toc38360235)

[Solution 94](#_Toc38360236)

[Time and space complexity 95](#_Toc38360237)

[16. Maximum element in binary tree 95](#_Toc38360238)

[Problem 95](#_Toc38360239)

[Reference 95](#_Toc38360240)

[Approach 95](#_Toc38360241)

[Solution 95](#_Toc38360242)

[Time and space complexity 96](#_Toc38360243)

[17. Find Node in a binary tree 96](#_Toc38360244)

[Problem 96](#_Toc38360245)

[Reference 96](#_Toc38360246)

[Approach 96](#_Toc38360247)

[Solution 96](#_Toc38360248)

[Time and space complexity 96](#_Toc38360249)

[18. Insert item in a binary tree 96](#_Toc38360250)

[Problem 96](#_Toc38360251)

[Reference 96](#_Toc38360252)

[Approach 96](#_Toc38360253)

[Solution 97](#_Toc38360254)

[Time and space complexity 97](#_Toc38360255)

[19. Level-Order Traversal Reverse Order 97](#_Toc38360256)

[Problem 97](#_Toc38360257)

[Reference 97](#_Toc38360258)

[Approach 97](#_Toc38360259)

[Solution 97](#_Toc38360260)

[Time and space complexity 98](#_Toc38360261)

[20. Diameter of Binary tree 98](#_Toc38360262)

[Problem 98](#_Toc38360263)

[Reference 98](#_Toc38360264)

[Approach 98](#_Toc38360265)

[Solution 99](#_Toc38360266)

[Time and space complexity 100](#_Toc38360267)

[11. Binary Search Tree 100](#_Toc38360268)

[1. Search in Binary Search Tree 100](#_Toc38360269)

[Problem 100](#_Toc38360270)

[Reference 101](#_Toc38360271)

[Approach 101](#_Toc38360272)

[Solution 101](#_Toc38360273)

[Time and space complexity 101](#_Toc38360274)

[2. Insert in Binary Search Tree 101](#_Toc38360275)

[Problem 101](#_Toc38360276)

[Reference 102](#_Toc38360277)

[Approach 102](#_Toc38360278)

[Solution 102](#_Toc38360279)

[Time and space complexity 102](#_Toc38360280)

[3. Construct Binary Search Tree from given pre-order and in-order 102](#_Toc38360281)

[Problem 102](#_Toc38360282)

[Reference 102](#_Toc38360283)

[Approach 102](#_Toc38360284)

[Solution 103](#_Toc38360285)

[Time and space complexity 104](#_Toc38360286)

[4. Construct Binary Search Tree from given pre-order 104](#_Toc38360287)

[Problem 104](#_Toc38360288)

[Reference 104](#_Toc38360289)

[Approach 104](#_Toc38360290)

[Solution 105](#_Toc38360291)

[Time and space complexity 105](#_Toc38360292)

[12. Dynamic Programming 105](#_Toc38360293)

[1. Maximum sum in Contiguous Sub-Array 105](#_Toc38360294)

[Problem 105](#_Toc38360295)

[Reference 105](#_Toc38360296)

[Approach 105](#_Toc38360297)

[Solution 106](#_Toc38360298)

[Time and space complexity 106](#_Toc38360299)

[13. Design data structure 106](#_Toc38360300)

[1. Design LRU cache with put and get in o(1) time 106](#_Toc38360301)

[Problem 106](#_Toc38360302)

[Reference 107](#_Toc38360303)

[Approach 107](#_Toc38360304)

[Solution 107](#_Toc38360305)

[Time and space complexity 108](#_Toc38360306)

[13. Heap 108](#_Toc38360307)

[1. Add and delete element from Max heap. 110](#_Toc38360308)

[Problem 110](#_Toc38360309)

[Reference 111](#_Toc38360310)

[Approach 111](#_Toc38360311)

[Solution 111](#_Toc38360312)

[Time and space complexity 112](#_Toc38360313)

[2. Last Stone Weight 113](#_Toc38360314)

[Problem 113](#_Toc38360315)

[Reference 113](#_Toc38360316)

[Approach 113](#_Toc38360317)

[Solution 113](#_Toc38360318)

[Time and space complexity 114](#_Toc38360319)

# 1. Array

## Find duplicate in the array

### Problem

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.

Input:

[4,3,2,7,8,2,3,1]

Output:

[2,3]

### Reference

LEETCODE

### Approach

Since content is with-in array index range. We will use itself array as hashTable and when we found item first time we change sign to negative and if second time we get same negative it is duplicate. If data was not in-range we will use hashmap or set.

### Solution

public List<Integer> findDuplicates(int[] nums) {

List<Integer> list = new ArrayList<>();

for (int i = 0; i < nums.length; i++) {

int n=Math.abs(nums[i])-1;

if (nums[n] < 0) {

list.add(n+1);

} else {

nums[n] = -nums[n];

}

}

return list;

}

### Time and space complexity

Time - O(n)

Space – O(1).In space as list needed only for this question

## Find Single element in the array

### Problem

Given a **non-empty** 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?

**Example 1:**

**Input:** [2,2,1]

**Output:** 1

**Example 2:**

**Input:** [14,1,2,1,2]

**Output:** 14

### Reference

LEETCODE

### Approach

We can use xor. Xor of two same elements is 0. So we loop array and do xor after end of loop. We have single element in result.

### Solution

public int singleNumber(int[] nums) {

int res = nums[0];

for(int i=1; i<nums.length; i++){

res = res^nums[i];

}

return res;

}

### Time and space complexity

Time - O(n)

Space – O(1)

## Find Numbers with Even Number of Digits

### Problem

Given an array nums of integers, return how many of them contain an even number of digits.

Example 1:

Input: nums = [12,345,2,6,7896]

Output: 2

Explanation:

12 contains 2 digits (even number of digits).

345 contains 3 digits (odd number of digits).

2 contains 1 digit (odd number of digits).

6 contains 1 digit (odd number of digits).

7896 contains 4 digits (even number of digits).

Therefore only 12 and 7896 contain an even number of digits.

### Reference

LEETCODE, MATH

### Approach

Approach 1 - can be to iterate over loop and convert each number to String and then check length is even or odd.

Approach 2 - can be to iterate over loop and use Math.log10 method and then check result%2==0. if it is true it is ODD else EVEN.

### Solution

public int findNumbers(int[] nums) {

int c=0;

for(int i=0;i<nums.length;i++){

int result = (int)Math.log10(nums[i]);

if(result %2!=0){

c++;

}

}

return c;

}

### Time and space complexity

Time - O(n)

Space – O(1)

## Largest Number

### Problem

Given a list of non-negative integers, arrange them such that they form the largest number.

Example 1:

Input: [10,2]

Output: "210"

Example 2:

Input: [3,30,34,5,9]

Output: "9534330"

### Reference

Leetcode, sort, array

### Approach

We need to sort data smartly i.e. write comparator smartly. So, for two string like 3 and 34 to check which one should come first just contact both combo like – 334 and 343. Now we know number larger can be made if 34 comes first and 3 after that.

So, we use above logic and sort the array.

### Solution

public String largestNumber(int[] nums) {

String[] str = new String[nums.length];

for(int i=0;i<nums.length;i++){

str[i]=String.valueOf(nums[i]);

}

Arrays.sort(str,(o1,o2)->{

String s1=o1+o2;

String s2=o2+o1;

return s2.compareTo(s1);

});

if("0".equals(str[0])){

return str[0];

}

StringBuilder sb=new StringBuilder(nums.length);

for(String s:str){

sb.append(s);

}

return sb.toString();

}

### Time and space complexity

Time - O(nlogn)

## Search Insert Position (Binary Search)

### Problem

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

### Reference

LEETCODE, BINARY-SEARCH, ARRAY

### Approach

Binary search is best algorithm to search in a sorted array. It takes o(logn) time.

1. set start=0 and end=length-1

2. Iterate till start<=end

3. get mid of (start+end)/2 and check if target is in left or right or in the middle.

4. If target<arr[mid] it means target is present in left. So update end=mid-1.

5. So, by this approach we are dividing the items to be searched to half every time.

In this particular problem if element does not exist. In such case start will tell the position of element where it should supposed to be. In classic binary search if item does not found we return -1

### Solution

public int searchInsert(int[] nums, int target) {

int start = 0;

int end = nums.length - 1;

int mid = 0;

while (start <= end) {

mid = (start + end) / 2;

if (nums[mid] == target) {

return mid;

} else if (nums[mid] < target) {

start = mid + 1;

} else {

end = mid - 1;

}

}

return start;

}

### Time and space complexity

O(logn)

O(1)

## Count Negative Numbers in a Sorted Matrix

### Problem

Given m \* n matrix grid which is sorted in non-increasing order both row-wise and column-wise.

Return the number of negative numbers in grid.

Example 1:

Input: grid = [[4,3,2,-1],[3,2,1,-1],[1,1,-1,-2],[-1,-1,-2,-3]]

Output: 8

Explanation: There are 8 negatives number in the matrix.

### Reference

LEETCODE, ARRAY-2D, BINARY-SEARCH

### Approach

We use binary search algorithm row by row and find center if it’s negative update end=center-1 else start=center+1

When loop terminates start will be the index of first negative. So, total negative in that row is row.length – start.

Since we also know that column is also decreasing so, for second row we update end to start-1.so, that we will apply binary search to only 0 to last positive number in previous row.

And we keep it doing till last row.

### Solution

public int countNegatives(int[][] grid)

{

int c = 0;

for (int i = 0, end = grid[i].length - 1; i < grid.length; i++) {

int start = 0;

while (start <= end) {

int mid = (start + end) / 2;

if (grid[i][mid] < 0) {

end = mid - 1;

} else {

start = mid + 1;

}

}

c = c + grid[i].length - start;

end = start - 1;

}

return c;

}

### Time and space complexity

Time - O(n+m)

Space - O(1)

## Move Zeroes to end of array

### Problem

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.

Example:

Input: [0,1,0,3,12]

Output: [1,3,12,0,0]

### Reference

LEETCODE, ARRAY

### Approach

Here to make code generic we move val to the end of array.

We will keep count of val in c. and if c> 0 means we have at least one val. We move current element to i-c location. And update arr[i] to val.

It works because we make sure we are shifting non zero element to next available index on left. which will be i-c.

If we does not have any zero we will not shift.

e.g. – 010004 -> in this case 1 will be shift to zero index .

i.e. 100004. Now c=1 and i=1. So, c keep on incrementing to 4. For i=5,arr[5-4]=arr[5] .

so,op will be 140000.

### Solution

public void searchAndShift(int[] arr, int val) {

int c = 0;

for (int i = 0; i < arr.length; i++) {

if (arr[i] == val) {

c++;

} else if (c > 0) {

arr[i - c] = arr[i];

arr[i] = val;

}

}

}

### Time and space complexity

Time - O(n)

Space – O(1)

## Two Sum

### Problem

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].

### Reference

LEETCODE, ARRAY, HASHMAP

### Approach

Take hashmap and check if current item is in map if yes return else put (target-current item) in a loop.

### Solution

public int[] twoSum(int[] numbers, int target) {

Map<Integer, Integer> map = new HashMap<>();

for(int i = 0; i < numbers.length; i++) {

if(map.containsKey(numbers[i])) {

return new int[] {map.get(numbers[i]), i};

}

map.put(target - numbers[i], i);

}

return null;

}

### Time and space complexity

Time - O(n)

Space – O(n) (hashmap)

## Merge two sorted array

### Problem

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

Note:

The number of elements initialized in nums1 and nums2 are m and n respectively.

You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2.

Example:

Input:

nums1 = [1,2,3,0,0,0], m = 3

nums2 = [2,5,6], n = 3

Output: [1,2,2,3,5,6]

### Reference

LEETCODE, ARRAY

### Approach

Since we have empty places at end of nums1. We will start comparing from end i.e. from m-1 and n-1 and start filling larger element among two to the end of nums1. E.g. for 6 and 3 -> Output would for nums1=[1,2,3,0,0,6] and we decrement n only and we keep on doing it till the length of nums1 (not m).

### Solution

public void merge(int[] nums1, int m, int[] nums2, int n) {

int i = nums1.length - 1;

n--;

m--;

while (m >= 0 && n >= 0) {

if (nums1[m] > nums2[n]) {

nums1[i--] = nums1[m--];

} else {

nums1[i--] = nums2[n--];

}

}

while (n >= 0) {

nums1[i--] = nums2[n--];

}

}

### Time and space complexity

Time - O(nums1.length)

Space – O(1)

## Best Time to Buy & Sell Stock

### Problem

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 (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one.

**Example 1:**

**Input:** [7,1,5,3,6,4]

**Output:** 5

**Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 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

**Explanation:** In this case, no transaction is done, i.e. max profit = 0.

### Reference

LEETCODE, ARRAY

### Approach

1. Assign min=prices[0] and maxProfit as 0.
2. Iterate over entire array and We will keep track of min first and also update maxProfit if prices[i] - min > maxProfit.
3. After loop finishes we will have maxProfit.

### Solution

public int maxProfit(int[] prices) {

if(prices==null || prices.length<2){

return 0;

}

int min = prices[0];

int maxProfit = 0;

for(int i=1;i<prices.length;i++){

if(min>prices[i]){

min=prices[i];

}

if((prices[i]-min) > maxProfit){

maxProfit = prices[i]-min;

}

}

return maxProfit;

}

### Time and space complexity

Time - O(n)

Space – O(1)

## Best Time to Buy & Sell Stock II

### Problem

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 (i.e., buy one and sell one share of the stock multiple times).

**Note:** You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** [7,1,5,3,6,4]

**Output:** 7

**Explanation:** Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

  Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

**Example 2:**

**Input:** [1,2,3,4,5]

**Output:** 4

**Explanation:** Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

  Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are

  engaging multiple transactions at the same time. You must sell before buying again.

### Reference

LEETCODE, ARRAY

### Approach

1. We will consider buying and purchasing if i-1 element is smaller than i. and add it to total profit.
2. Take maxProfit=0.
3. Iterate through each element and update max with max += arr[i]-arr[i-1]. If arr[i]>arr[i-1].

### Solution

**public** **int** maxProfit2(**int**[] prices) {

**if** (prices == **null** || prices.length < 2) {

**return** 0;

}

**int** maxProfit = 0;

**for** (**int** i = 1; i < prices.length; i++) {

**if** (prices[i - 1] < prices[i]) {

maxProfit += prices[i] - prices[i - 1];

}

}

**return** maxProfit;

}

### Time and space complexity

Time - O(n)

Space – O(1)

## Length of longest contiguous subarray with equal number of 0 and 1.

### Problem

Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1.

**Example 1:**

**Input:** [1,0,1,1,0]

**Output:** 4

**Explanation:** [0,1,1,0] is a longest contiguous subarray with equal number of 0 and 1

### Reference

LEETCODE, ARRAY, GITHUB

### Approach

1. We will keep track of sum in sum and maximum length in max variable.
2. To make it simple we will store -1 in place of 0.
3. We take map and store sum as key and index as value. Add <0,-1> in map. As by default 0 sum has -1 length.
4. While iterating if sum is not present in map we add sum as key and index as value.
5. If at any point map already contains that key it means that numbers present between starting of that index to the current index has equal number of -1 and 1 (this is because adding 0 to any number is number itself)

Then We update max if it’s less than current\_length(i.e. i-map.get(sum)).

### Solution

public int findMaxLength(int[] nums) {

int sum = 0;

int max = 0;

Map<Integer,Integer> map = new HashMap<>();

map.put(0,-1);

for(int i=0;i<nums.length;i++) {

sum+=(nums[i]==0 ? -1 : 1);

if(map.containsKey(sum)) {

if(max < i-map.get(sum)) {

max = i-map.get(sum);

}

} else {

map.put(sum,i);

}

}

return max;

}

### Time and space complexity

Time - O(n)

Space – O(n)

## Product of Array Except Self.

### Problem

Given an array nums of n integers where n > 1,  return an array output such that output[i] is equal to the product of all the elements of nums except nums[i].

**Example:**

**Input:** [1,2,3,4]

**Output:** [24,12,8,6]

**Note:**Please solve it **without division** and in O(n).

### Reference

LEETCODE, ARRAY, GITHUB

### Approach

1. Since we can create result array we will use it to store multiplication result.
2. First we go from left to right starting from i=1 and keep storing product at I index as = res[i-1]\*nums[i-1]
3. After loop finishes at every index we have multiplication result of all left elements. so, now we just need to multiply right elements product also.
4. Since last index already has desired result. We start from second last index.
5. Take var m to hold current multiplication result.
6. Keep on multiplying m with output array index and then update m with m\*nums[i]

### Solution

**public** **int**[] productExceptSelf(**int**[] nums) {

**int**[] output = **new** **int**[nums.length];

output[0] = 1;

// count multiplication result from left to right in every index.

**for** (**int** i = 1; i < nums.length; i++) {

output[i] = nums[i - 1] \* output[i - 1];

}

//in above example output[]=[1,1,2,6]

// now every index has multiplication result till left.

//so, go from right to left and multiply remaining element for every index from

// right.

**int** m = nums[nums.length - 1];//m will keep multiplication from right till i index

**for** (**int** i = nums.length - 2; i >= 0; i--) {

output[i] = output[i] \* m;

m = m \* nums[i];

}

**return** output;

}

### Time and space complexity

Time - O(n)

Space – O(1). If we ignore output array

## Minimum Path sum in 2D-Array.

### Problem

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:**

**Input:**

[

  [1,3,1],

[1,5,1],

[4,2,1]

]

**Output:** 7

**Explanation:** Because the path 1→3→1→1→1 minimizes the sum.

### Reference

LEETCODE, ARRAY, GITHUB, GRAPH

### Approach

1. For any node present at index i,j. the only possible way is to come from either left or from top.
2. For any index I,j we check out of left or top whose value is minimum -> grid[i-1][j],grid[i][j-1] and add it to the current node path.
3. So, we keep on updating distance of grid[i][j]= grid[i][j] + Min of grid[i-1][j],grid[i][j-1]
4. Once we reach at last index we have the desired minimum path.

### Solution

**public** **int** minPathSum(**int**[][] grid) {

**for** (**int** i = 0; i < grid.length; i++) {

**for** (**int** j = 0; j < grid[i].length; j++) {

**int** left = i > 0 ? grid[i - 1][j] : Integer.***MAX\_VALUE***;

**int** top = j > 0 ? grid[i][j - 1] : Integer.***MAX\_VALUE***;

**if** (i == 0 && j == 0) {

**continue**;

}

grid[i][j] += MathUtil.*min*(left, top);

}

}

**return** grid[grid.length - 1][grid[0].length - 1];

}

### Time and space complexity

Time - O(n\*m)

Space – O(1).

## Number of Islands in 2-D Array.

### Problem

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

### Reference

LEETCODE, ARRAY, GITHUB, GRAPH

### Approach

1. All 1 together makes one island.
2. So we start with first 1 and mark it as visited and will keep on visiting all neighbor 1 till possible.
3. After that increment island count.
4. Now repeat step 2 and 3 again for next 1.

### Solution

**public** **int** numIslands(**char**[][] grid) {

**int** c = 0;

**for** (**int** i = 0; i < grid.length; i++) {

**for** (**int** j = 0; j < grid[i].length; j++) {

**if** (grid[i][j] == '1') {

*visitIsland*(grid, i, j);

c++;

}

}

}

**return** c;

}

**private** **void** visitIsland(**char**[][] grid, **int** i, **int** j) {

**if** (grid[i][j] == '1') {

grid[i][j] = 'V';

**if** (i - 1 >= 0) {

*visitIsland*(grid, i - 1, j);

}

**if** (i + 1 < grid.length) {

*visitIsland*(grid, i + 1, j);

}

**if** (j - 1 >= 0) {

*visitIsland*(grid, i, j - 1);

}

**if** (j + 1 < grid[i].length) {

*visitIsland*(grid, i, j + 1);

}

}

}

### Time and space complexity

Time - O(n\*m) – extra

Space – O(1).

# 2. String

## Check if a string is substring of source. (Rabin Karp Algorithm)

### Problem

Check whether a given string is substring of source.

Example 1-

Input –

helloji,loj

Output –

True

### Reference

STRING, RABIN-KARP, ABDUL BARI

### Approach

Naive approach is to check character by character starting from i=0 and if not matched go back and now check for i=1.

Better approach -

It uses hashcode of a string and instead of matching character one by one.

\* We just match hashcode and once hashcode matched we check the content.

\* If not matched we subtract hashcode of first character and add hashcode of new character

\* It saves time of un-necessary comparison all the time.

\* But in worst case it might be possible that we might get hashcode same on every check.

\* To calculate hashcode again we just subtract hashcode of first character and add hashcode of next character in previous value.

\* For better performance make hash code function better to avoid un-necessary collision.

### Solution

int hSource = 0;

int hStr = 0;

//calculate hashcode of both source and string for first comparison

for (int i = 0; i < str.length(); i++) {

hSource = hSource + hashFunction(source.charAt(i));

hStr = hStr + hashFunction(str.charAt(i));

}

// we compare hash first and if matched return true.

// calculate hash again except for last value of i as we are generating hash in

// advanced.

for (int i = 0; i <= source.length() - str.length(); i++) {

if (hStr == hSource) {

int j = 0;

for (j = 0; j < str.length(); j++) {

if (source.charAt(j + i) != str.charAt(j)) {

break;

}

}

if (j == str.length()) {

return true;

}

}

//to avoid calculation after last index

if (i < source.length() - str.length()) {

hSource = hSource - hashFunction(source.charAt(i)) + hashFunction(source.charAt(i + str.length()));

}

}

return false;

### Time and space complexity

\* worst case - o(n\*m)

\* Best case - o(n+m)

## Anagram

### Problem

An anagram is a word formed by rearranging the letters of a different word. typically using all the original letters exactly once.

Given two strings s and t, write a function to determine if t is an anagram of s.

You may assume the string contains only lowercase alphabets.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

### Reference

LEETCODE, ARRAY

### Approach

Take array with 26 size and from first string increment counter and for second decrement counter.

After loop finished iterate over table array and check if any non-zero value exists it’s not anagram.

### Solution

public boolean isAnagram(String s, String t) {

int[] table=new int[26];

if(s.length()!=t.length()){

return false;

}

for(int i=0;i<s.length();i++){

table[s.charAt(i)-'a']+=1;

table[t.charAt(i)-'a']-=1;

}

for(int i=0;i<table.length;i++){

if(table[i]!=0){

return false;

}

}

return true;

}

### Time and space complexity

Time - O(n+26)

Space – O(26) means constant

## Group Anagrams

### Problem

Given an array of strings, group anagrams together.

**Example:**

**Input:** ["eat", "tea", "tan", "ate", "nat", "bat"],

**Output:**[["ate","eat","tea"],["nat","tan"],["bat"]]

**Note:**

* All inputs will be in lowercase.
* The order of your output does not matter.

### Reference

LEETCODE, STRING, ANAGRAM, HASHMAP

### Approach

1. It is asked in question to group the anagram together. So we can make map<String,List<String>> to hold the data.

2. For every anagram we need to generate same Map key. So, that we don’t need to check anything extra.

3. We will write hash method which will generate same hash value for every anagram.

4. For that we will create table array and count occurrences of each character of passed String.

5. Take String h and Now just update h with count first followed by character. For e.g. for eat and tea same hash will be returned which is 1a1e1t. but for tab it will be 1a1b1t.

### Solution

public List<List<String>> groupAnagrams(String[] strs) {

Map<String,List<String>> map = new HashMap<>();

for(int i=0;i<strs.length;i++){

String k= hash(strs[i]);

if(!map.containsKey(k)) {

map.put(k,new LinkedList<>());

}

map.get(k).add(strs[i]);

}

return new ArrayList<>(map.values());

}

private String hash(String str) {

int[] ch = new int[26];

StringBuilder h = new StringBuilder();

for(int i=0;i<str.length();i++){

ch[str.charAt(i)-'a']+=1;

}

for(int i=0;i<ch.length;i++){

if(ch[i]!=0){

h.append(ch[i]);

h.append((char)('a'+i));

}

}

return h.toString();

}

### Time and space complexity

Time - O(nm)

Space – map space and character array space.

## Backspace String Compare

### Problem

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".

### Reference

LEETCODE, STRING

### Approach

Logic is that we will prepare final first String and final second String after applying backspace. And after that compare both strings if they are same. Perform below operation on both Strings one by one.

1. start iterating String from backward and check if # meets. If exits count b.
2. for next item if # does not comes and b is positive it means it should be removed so we does not append to StringBuilder.
3. But if items is not # and also b<=0 . it means this item needed to be considered and we hence add it to StringBuilder.

### Solution

**public** **boolean** backspaceCompare(String s, String t) {

**return** *removeBackSpace*(s).equals(*removeBackSpace*(t));

}

**private** **static** String removeBackSpace(String s) {

StringBuilder s1 = **new** StringBuilder();

**int** b = 0;

**for** (**int** i = s.length() - 1; i >= 0; i--) {

**if** (s.charAt(i) == '#') {

b++;

} **else** {

**if** (b > 0) {

b--;

} **else** {

s1.append(s.charAt(i));

}

}

}

**return** s1.toString();

}

### Time and space complexity

Time - O(n)

Space – o(1).

## Perform String Shifts

### Problem

You are given a string s containing lowercase English letters, and a matrix shift, where shift[i] = [direction, amount]:

* direction can be 0 (for left shift) or 1 (for right shift).
* amount is the amount by which string s is to be shifted.
* A left shift by 1 means remove the first character of s and append it to the end.
* Similarly, a right shift by 1 means remove the last character of s and add it to the beginning.

Return the final string after all operations.

**Example 1:**

**Input:** s = "abc", shift = [[0,1],[1,2]]

**Output:** "cab"

**Explanation:**

[0,1] means shift to left by 1. "abc" -> "bca"

[1,2] means shift to right by 2. "bca" -> "cab"

**Example 2:**

**Input:** s = "abcdefg", shift = [[1,1],[1,1],[0,2],[1,3]]

**Output:** "efgabcd"

**Explanation:**

[1,1] means shift to right by 1. "abcdefg" -> "gabcdef"

[1,1] means shift to right by 1. "gabcdef" -> "fgabcde"

[0,2] means shift to left by 2. "fgabcde" -> "abcdefg"

[1,3] means shift to right by 3. "abcdefg" -> "efgabcd"

### Reference

LEETCODE, STRING

### Approach

You may notice that left shift cancels the right shift, so count the total left shift times (may be negative if the final result is right shift), and perform it once.

### Solution

**public** String stringShift(String s, **int**[][] shift) {

**int** c = 0;

**for** (**int** i = 0; i < shift.length; i++) {

**if** (shift[i][0] == 0) {

c += shift[i][1];

} **else** {

c -= shift[i][1];

}

}

**if** (c > 0) {

**return** *shiftLeft*(c % s.length(), s);

}

**return** *shiftRight*((-c) % s.length(), s);

}

/\*\*

\* **@return** String after left rotation n

\*/

**private** String shiftLeft(**int** n, String s) {

String pre = s.substring(0, n);

s = s.substring(pre.length(), s.length());

**return** s + pre;

}

/\*\*

\* **@return** String after right rotation n

\*/

**private** String shiftRight(**int** n, String s) {

String suf = s.substring(s.length() - n, s.length());

s = s.substring(0, s.length() - suf.length());

**return** suf + s;

}

### Time and space complexity

Time - O(n)

Space – o(1)

## Check Valid Parenthesis String

### Problem

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:

1. Any left parenthesis '(' must have a corresponding right parenthesis ')'.
2. Any right parenthesis ')' must have a corresponding left parenthesis '('.
3. Left parenthesis '(' must go before the corresponding right parenthesis ')'.
4. '\*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string.
5. An empty string is also valid.

**Example 1:**

**Input:** "()"

**Output:** True

**Example 2:**

**Input:** "(\*)"

**Output:** True

**Example 3:**

**Input:** "(\*))"

**Output:** True

### Reference

LEETCODE, STRING,GITHUB

### Approach

First consider \* as part of ‘(‘ and iterate from left to right. Secondly consider \* part of ‘)’ and check from right to left.

We follow algo – if ‘(‘ encounter c++ else c--. At any point if c is negative it means invalid string. And after loop finishes if c=0 . it means valid. Here we do that algo for both directions one by one.

### Solution

**public** **static** **boolean** checkValidString(String s) {

**int** bal = 0;

**for** (**int** i = 0; i < s.length(); i++) {

**if** (s.charAt(i) == '(' || s.charAt(i) == '\*') {

bal++;

} **else** **if** (--bal < 0) {

**return** **false;**//it means even ‘(‘ + ’\*’ is not sufficient

}

}

// terminate if brackets are balanced already.

**if** (bal == 0) {

**return** **true**;

}

//we know that now combo of ‘(‘ + ’\*’ > ‘)’. So verify from right we can make

//valid substring.

//repeat again by going right to left

bal = 0;

**for** (**int** i = s.length() - 1; i >= 0; i--) {

**if** (s.charAt(i) == ')' || s.charAt(i) == '\*') {

bal++;

} **else** **if** (--bal < 0) {

**return** **false**;

}

}

// if we reach here it means valid as it passes both loops without failing.

//either bal=0 or we have some extra stars left.

**return** **true**;

}

### Time and space complexity

Time - O(n)

Space – o(1)

# 3. Math

## 1. Palindrome Number

### Problem

Determine whether an integer is a palindrome. An integer is a palindrome when it reads the same backward as forward.

Example 1:

Input: 121

Output: true

Example 2:

Input: -121

Output: false

Explanation: From left to right, it reads -121. From right to left, it becomes 121-

### Reference

LEETCODE, MATH, MOD

### Approach

Reverse the original number by adding remainder to the original number – res\*10+(num%10);

### Solution

public boolean isPalindrome(int num) {

int res = 0;

int num1 = num;

while (num > 0) {

res = res \* 10 + (num % 10);

num = num / 10;

}

return num1 == res;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 2. Happy Number

### Problem

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. Those numbers for which this process ends in 1 are happy numbers.

**Example:**

**Input:** 19

**Output:** true

**Explanation:**

12 + 92 = 82

82 + 22 = 68

62 + 82 = 100

12 + 02 + 02 = 1

### Reference

LEETCODE, MATH

### Approach

1. Loop infinite and find squareAndSum of n.

2. If n reach to 7 or 1. It means it is happy number as 7 will also resolve to 1.

3. Any other single number digit will never resolve to 1 or 7. and hence we return false in such case

### Solution

**public** **boolean** isHappy(**int** n) {

**if** (n < 1) {

**return** **false**;

}

**while** (**true**) {

n = squareAndSum(n);

**if** (n == 1 || n == 7)

**return** **true**;

**if** (n < 10)

**return** **false**;

}

}

**private** **int** squareAndSum(**int** n) {

**int** sum = 0;

**while** (n != 0) {

**int** rem = n % 10;

n = n / 10;

sum = sum + (rem \* rem);

}

**return** sum;

}

### Time and space complexity

Time -

Space – o(1)

# 4. Linked List

## 1. Reverse single linked list (Iterative + recursive)

### Problem

Reverse single linked list.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. It can be done using three pointers. One to hold the current node,one with prev and one next node
2. Start by setting prev=null and current=head.
3. Iterate till current is not null and for every iteration set next = current.next and now set current.next=prev
4. And after that update prev to current and current to next.
5. After loop terminates current is the head of revsered list

### Solution

public void reverse() {

Node<T> prev = null;

while (head != null) {

Node<T> next = head.getNext();

head.setNext(prev);

prev = head;

head = next;

}

head = prev;

}

**public** Node<Integer> reverseListRecursive(Node<Integer> head) {

**return** reverseRecursive(head,**null**);

}

**private** Node<Integer> reverseRecursive(Node<Integer> head,Node<Integer> prev){

**if**(head==**null**){

**return** prev;

}

Node<Integer> next=head.getNext();

head.setNext(prev);

**return** reverseRecursive(next,head);

}

### Time and space complexity

Time - o(n)

Space – o(1) for iterative and o(n) for recursive

## Reverse single linked list recursive

### Problem

Reverse single linked list.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. Call method till we reach last node. It will return last node
2. After that the head is second last node.
3. So, set last node next as second last.
4. Set second last node next as null. Otherwise it will form cycle.
5. Since in first call we will last node which will be new head. So, we return same node p from every call.

### Solution

**public** Node<Integer> reverseListRecursive(Node<Integer> head) {

// for last node return head.

**if** (head == **null** || head.getNext() == **null**) {

**return** head;

}

// call method till last node

Node<Integer> p = *reverseListRecursive*(head.getNext());

// after above call stack end we will receive last node in p

// and at present head is second last node

// set last node next as second last

head.getNext().setNext(head);

// set head next null

head.setNext(**null**);

// return last node, As we are returning same p again and again. we will receive last node only.

**return** p;

}

### Time and space complexity

Time - o(n)

Space – o(n)

## Check Palindrome single linked list

### Problem

Check if given single linked list is palindrome.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. It can be done iteratively and recursively.
2. In **Iterative solution**->
   1. First find the middle of the linked list. After that set middle.next as null. So that we have now two linked list.
   2. Reverse one of the linked list and now compare one by one both linked list and if item different list is not palindrome.
3. In **Recursive solution->**
   1. take global variable left to hold the start side of the list and then call method recursively to reach the last element of the list and now compare if left item is equal to last item if yes return true else return false.
   2. Break the recursion if false is received or left reached end.

### Solution

**Iterative** –

public static boolean checkPalindromeIterative(Node<Integer> head) {

if (head == null || head.getNext() == null) {

return true;

}

// step1 find middle of the linked list

Node<Integer> middle = findMiddle(head);

// step2 partition the list into two halves- right and head

Node<Integer> right = middle.getNext();

middle.setNext(null);

// reverse one half

right = reverse(right);

//iterate with smaller half not null. in this way we ignore middle odd element if present.

while (right != null) {

if (head.getData() != right.getData()) {

return false;

}

head = head.getNext();

right = right.getNext();

}

return true;

}

private static Node<Integer> reverse(Node<Integer> head) {

Node<Integer> prev = null;

while (head != null) {

Node<Integer> next = head.getNext();

head.setNext(prev);

prev = head;

head = next;

}

return prev;

}

public static Node<Integer> findMiddle(Node<Integer> head) {

if (head == null) {

return head;

}

Node<Integer> fast = head;

Node<Integer> slow = head;

while (fast.getNext() != null && fast.getNext().getNext() != null) {

fast = fast.getNext().getNext();

slow = slow.getNext();

}

return slow;

}

**Recursive** –

private Node<T> left=head;

private boolean checkPalindrome(Node<T> node) {

if (left == null || node == null) {

return true;

}

boolean res = checkPalindrome(node.getNext());

if (res && left.getData().equals(node.getData())) {

left = left.getNext();

return true;

}

return false;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 4. Find middle element of single linked list

### Problem

Middle element of single linked list.

Input: [1,2,3,4,5,6]

Output: 3

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. Take two pointers one run at double speed and other single.
2. Once fast pointer reaches null. At that point slow will be in the middle.
3. If we want to return 4 in above code use below code –

Node<Integer> fast = head;

Node<Integer> slow = head;

while (fast!= null && fast.getNext()!= null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

}

return slow;

### Solution

public static Node<Integer> findMiddleOfLinkedList(Node<Integer> head) {

if (head == null) {

return head;

}

Node<Integer> fast = head;

Node<Integer> slow = head;

while (fast.getNext() != null && fast.getNext().getNext() != null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

}

return slow;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 5. Detect cycle in linked list

### Problem

Check if cycle exists in single linked list.

Input:



Output:

true

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. Take two pointers one run at double speed and other single.
2. If at any point they meet-> there is a cycle. And if node reaches null value. No cycle.

### Solution

public boolean hasCycle(Node<Integer> head) {

Node<Integer> slow = head;

Node<Integer> fast = head;

while (fast != null && fast.getNext() != null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

if (slow == fast) {

return true;

}

}

return false;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 6. Detect cycle in linked list and return starting node of loop

### Problem

Check if cycle exists in single linked list. If loop is present then it returns point to first node of loop. Else it returns NULL.



### Reference

SINGLE LINKED LIST, LEETCODE, GEEKSFORGEEKS

### Approach

1. Take two pointers one run at double speed and other single.
2. If at any point they meet-> there is a cycle. And if node reaches null value. No cycle.
3. If cycle exists set any of the two pointer to head and now loop till both of them meets again. That meeting point is cycle starting point.

### Solution

public static Node<Integer> detectCycle(Node<Integer> head) {

Node<Integer> slow = head;

Node<Integer> fast = head.getNext();

while (fast != null && fast.getNext() != null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

if (slow == fast) {

slow=head;

while(slow!=fast) {

slow=slow.getNext();

fast=fast.getNext();

}

return fast;

}

}

return null;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 7. Intersection of Two Linked Lists

### Problem

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:



Begin to intersect at node 8.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

* Get count of the nodes in the first list, let count be c1.
* Get count of the nodes in the second list, let count be c2.
* Get the difference of counts d = abs(c1 – c2)
* Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes.
* Then we can traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

### Solution

public Node<Integer> getIntersectionNode(Node<Integer> headA, Node<Integer> headB) {

int lenA = getLength(headA);

int lenB = getLength(headB);

Node<Integer> first = null;

Node<Integer> second = null;

int diff = 0;

if (lenA > lenB) {

first = headA;

second = headB;

diff = lenA - lenB;

} else {

first = headB;

second = headA;

diff = lenB - lenA;

}

for (int i = 0; i < diff; i++) {

first = first.getNext();

}

while (first != null && second != null) {

if (first == second) {

return first;

}

first = first.getNext();

second = second.getNext();

}

return null;

}

public int getLength(Node<Integer> temp) {

int len = 0;

while (temp != null) {

temp = temp.getNext();

len++;

}

return len;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 8. Delete Linked List Elements

### Problem

Write a program to remove all elements from a linked list of integers that have value ‘val’.

Example:

Input: 1->2->6->3->4->5->6, val = 6

Output: 1->2->3->4->5

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. First delete the head till val=head.val

2. Then take two pointer prev and curr.

3. If item match set prev.next=curr.next

### Solution

public ListNode removeElements(ListNode head, int val) {

while(head!=null && head.val==val){

head=head.next;

}

if(head==null){

return null;

}

ListNode prev = head;

ListNode temp = head.next;

while(temp!=null){

if(temp.val==val){

prev.next=temp.next;

}

else {

prev=prev.next;

}

temp=temp.next;

}

return head;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 9. Separate odd even nodes

### Problem

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.

Input: 2->1->3->5->6->4->7->NULL

Output: 2->3->6->7->1->5->4->NULL

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1.hold odd and even node, now in loop first update odd node

odd.next=even.next and odd=even.next;

2.now if odd.next is null. that means we dont have anything to be append to even node. so set even.next=null

3.else there is element. so , set even.next=odd.next and even=even.next

4.once loop finishes we have two different nodes -> odd will be at end element of itself and headEven which we already stored earlier will be starting of even

5.so, set odd.next=headEven;

### Solution

public static Node<Integer> oddEvenList(Node<Integer> head) {

if (head == null || head.getNext() == null || head.getNext().getNext() == null) {

return head;

}

Node<Integer> odd = head;

Node<Integer> even = head.getNext();

Node<Integer> evenHead = even;

while(even!=null && even.getNext()!=null) {

odd.setNext(even.getNext());

odd=odd.getNext();

if(odd.getNext()==null) {

even.setNext(null);

} else {

even.setNext(odd.getNext());

even=even.getNext();

}

}

odd.setNext(evenHead);

return head;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 10. Remove Nth Node From End of List

### Problem

Given a linked list, remove the n-th node from the end of list and return its head.

**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.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

Instead of calculating length first and then perform len-n. we can do it in one pass by first moving till n. now start another pointer from head and move it till first pointer reaches end. At that point second pointer will be at nth position.

1. move curr to the nth node.
2. Curr2=head
3. Now start again till curr become null. and keep on incrementing curr2.
4. Now curr2.next is the node which we want to delete.

### Solution

public Node<Integer> removeNthFromEnd(Node<Integer> head, int n) {

Node<Integer> curr = head;

for (int i = 0; i < n; i++) {

curr = curr.getNext();

}

if (curr == null) {

return head.getNext();

}

Node<Integer> curr2 = head;

while (curr.getNext() != null) {

curr2 = curr2.getNext();

curr = curr.getNext();

}

curr2.setNext(curr2.getNext().getNext());

return head;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 11. Rotate List

### Problem

Given a linked list, rotate the list to the right by k places, where k is non-negative.

**Example 1:**

**Input:** 1->2->3->4->5->NULL, k = 2

**Output:** 4->5->1->2->3->NULL

**Explanation:**

rotate 1 steps to the right: 5->1->2->3->4->NULL

rotate 2 steps to the right: 4->5->1->2->3->NULL

**Example 2:**

**Input:** 0->1->2->NULL, k = 4

**Output:** 2->0->1->NULL

**Explanation:**

rotate 1 steps to the right: 2->0->1->NULL

rotate 2 steps to the right: 1->2->0->NULL

rotate 3 steps to the right: 0->1->2->NULL

rotate 4 steps to the right: 2->0->1->NULL

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. Calculate length of list and in same loop store last node also

2. To avoid cycle use k=k%length. If k=0 it means nothing to do just return head.

3. Just go to till length-k and now we know we need to break from here and store newHead. make last.next=newHead;

4.return newHead.

### Solution

public Node<Integer> rotateRight(Node<Integer> head, int k) {

if (head == null || head.getNext() == null || k == 0) {

return head;

}

//find length

int len = 1;

Node<Integer> last = head;

while (last.getNext() != null) {

last = last.getNext();

len++;

}

k = k % len;

if (k == 0) {

return head;

}

//update k = len - k

k = len - k;

Node<Integer> temp = head;

//goto till k. i.e. break point

for (int i = 1; i < k; i++) {

temp = temp.getNext();

}

Node<Integer> newHead = temp.getNext();

temp.setNext(null);

last.setNext(head);

return newHead;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 12. Swap Nodes in Pairs Recursive

### Problem

Given a linked list, swap every two adjacent nodes and return its head.

You may **not** modify the values in the list's nodes, only nodes itself may be changed.

**Example:**

Given 1->2->3->4, you should return the list as 2->1->4->3.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. We will take head, head.next and head.next.next in head, second, third variable.
2. Set second.next as head. And we will set head.next as second of next recursive call.
3. Basically we will return second of next recursive call which will take third as argument.

### Solution

**public** **static** Node<Integer> swapPairsRecursive(Node<Integer> head) {

**if** (head == **null** || head.getNext() == **null**) {

**return** head;

}

Node<Integer> second = head.getNext();

Node<Integer> third = second.getNext();

second.setNext(head);

head.setNext(*swapPairsRecursive*(third));

**return** second;

}

### Time and space complexity

Time - o(n)

Space – o(n)

# 5. Stack

Follow LIFO order and push and pop operation take o(1) time.

## 1. Next greatest element in array

### Problem

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

For input array ->

{4, 5, 2, 25}

Element NGE

4 --> 5

5 --> 25

2 --> 25

25 --> -1

### Reference

SINGLE LINKED LIST, GEEKSFORGEEKS

### Approach

1. It can be done using two loops where we first find the element and after that next greatest element.

but it can be done in better way by using stack.

Algorithm -

1.if stack is empty push item into stack

2. else check if top element from stack < current array element.

3.if yes pop element. that current array element is the next greatest element of the popped element.

4.Keep popping from the stack while the popped element is smaller than next.

next becomes the next greater element for all such popped elements

5.Finally, push current element to the stack.

6.after the array is traversed completely the elements remained in the stack has -1 as next greatest element.

Since we wanted to return output array,we can store index of the element instead of actual element in stack.

and when we need to pop we uses it as a index of output to store the current element.

### Solution

public int[] nextGreatestElement(int[] arr) {

int n = arr.length;

int[] res = new int[n];

Deque<Integer> stack = new LinkedList<>();

//fill array with -1

for (int i = 0; i < n; i++) {

res[i] = -1;

}

for (int i = 0; i < n; i++) {

while (!stack.isEmpty() && arr[stack.peek()] < arr[i]) {

res[stack.pop()] = arr[i];

}

stack.push(i);

}

return res;

}

### Time and space complexity

Time - o(n)

Space – o(n)

If array is circular here then in above code loop till n\*2 and use i%n instead of i

for (int i = 0; i < n\*2; i++) {

while (!stack.isEmpty() && arr[stack.peek()] < arr[i%n]) {

res[stack.pop()] = arr[i%n];

}

stack.push(i%n);

}

## 2. Valid Parentheses

### Problem

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Note that an empty string is also considered valid.

**Example 1:**

**Input:** "()"

**Output:** true

**Example 2:**

**Input:** "()[]{}"

**Output:** true

**Example 3:**

**Input:** "([)]"

**Output:** false

**Example 4:**

**Input:** "{[]}"

**Output:** true

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. It can be done using stack. we push ‘)‘ if current element is ‘(‘ and for ‘[‘ we push ‘]’ and for ‘{‘ we push ‘}’
2. If we have current item from ‘),},]’ we pop the element. if popped element is same as current item. It means we can continue else we return false.
3. During iteration if we know that stack is empty and we have received closed bracket as current element it means invalid also.
4. After loop finishes if stack is empty return true.

### Solution

public boolean isValid(String s) {

Deque<Character> stack = new LinkedList<>();

for(int i=0;i<s.length();i++) {

char ch=s.charAt(i);

if(ch=='('){

stack.push(')');

} else if(ch=='[') {

stack.push(']');

} else if(ch=='{') {

stack.push('}');

} else if(stack.isEmpty() || stack.pop()!=ch) {

return false;

}

}

return stack.isEmpty();

}

### Time and space complexity

Time - o(n)

Space – o(n)

## 3. Implement Queue using Stack

### Problem

Implement Queue using Stacks

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. It can be done using two stack.stack1 for addition and stack2 for removal

When we add element we push into stack1.

1. On removal we check if stack2 is empty add all elements from stack1 to stack2.

Return stack2.pop()

### Solution

private Deque<Integer> addS = new LinkedList<>();

private Deque<Integer> remS = new LinkedList<>();

/\*\* Push element x to the back of queue. \*/

public void push(int x) {

addS.push(x);

}

/\*\* Removes the element from in front of queue and returns that element. \*/

public int pop() {

if(remS.isEmpty()){

while(!addS.isEmpty()){

remS.push(addS.pop());

}

}

return remS.pop();

}

/\*\* Get the front element. \*/

public int peek() {

if(remS.isEmpty()){

while(!addS.isEmpty()){

remS.push(addS.pop());

}

}

return remS.peek();

}

/\*\* Returns whether the queue is empty. \*/

public boolean empty() {

return addS.isEmpty() && remS.isEmpty();

}

### Time and space complexity

Time - NA

Space – NA

## 4. Current Maximum element in stack

### Problem

Find maximum in a stack in O(1) time and O(1) extra space.

Given a Stack, keep track of the maximum value in it. The maximum value may be the top

element of the stack, but once a new element is pushed or an element is pop from the

stack, the maximum element will be now from the rest of the elements.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

If two stacks are allowed – on add operation item will be added in both stacks. Stack1 will have original element and on stack2 we will current max.

For e.g –

Add 11

Stack1- 11

stack2 – 11

Add 2

Stack1- 11,2

stack2 – 11,11

add 3

Stack1- 11,2,3

stack2 – 11,11,11

add 40

Stack1- 11,2,3,40

stack2 – 11,11,11,40

So, on pop stack2 will give current max. and on pop we will remove from both the stacks.

**Without using extra stack i.e. O(1) –**

We will keep int max to store max element of the stack and instead of pushing item directly onto stack we follow –

**PUSH**

1. check if stack is empty. if yes add element and set max = newItem.

2. else

2.1. check if newItem x>max .if yes, push x+max to stack and update max=x. else push x in stack.

**POP**

Pop element x. if it is greater than max then return x-max else return max

**PEEK**

Peek from stack x. If it is > max return max else return x.

### Solution

public class StackMaxElementImproved {

private final Deque<Integer> stack = new Deque<>();

private int max = -1;

// o(1)

/\*\*

\*

\* @f:off

\* 1. check if stack is empty. if yes add element and set max = newItem.

\* 2. else

\* 2.1. check if newItem x>max .if yes, push x+max to stack and update max=x. else push x in stack.

\* @f:on

\*

\* @param data

\*/

public void push(int data) {

if (stack.isEmpty()) {

stack.push(data);

max = data;

} else {

if (max < data) {

stack.push(max + data);

max = data;

} else {

stack.push(data);

}

}

}

// o(1)

/\*\*

\*

\* @return get max element from stack

\*/

public int getMax() {

return max;

}

/\*\*

\* @f:off

\* 1. peek from stack. if y is greater than max -> return max.

\* else return y

\* @f:on

\* @return top element from stack

\*/

public int peek() {

if (stack.isEmpty()) {

return -1;

}

int item = stack.peek();

if (item > max) {

item = max;

}

return item;

}

// o(1)

/\*\*

\* 1. pop from stack. if y is greater than max -> set max=y-max and return old max.

\* else return y

\*

\* @return popped item

\*/

public int pop() {

if (stack.isEmpty()) {

return -1;

}

int item = stack.pop();

int res = item;

if (item > max) {

res = max;

max = item - max;

}

return res;

}

}

### Time and space complexity

Time – o(1) time for max

Space – o(1)

# 6. Queue

FIFO order, add and poll, peek methods take o(1) time

## 1. Implement Stack using queue

### Problem

Implement Stack using Queue.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. It can be implemented using single queue.
2. logic is in push method only. In push method just move all the elements back to the queue except the newly

Inserted one. In this way pop and peek will return newly inserted element just like stack.

### Solution

public class StackUsingQueue {

private Queue<Integer> qa = new LinkedList<>();

/\*\* Push element x onto stack. \*/

public void push(int x) {

int n = qa.size();

qa.add(x);

while (n-- > 0) {

qa.add(qa.poll());

}

}

/\*\* Removes the element on top of the stack and returns that element. \*/

public int pop() {

return qa.poll();

}

/\*\* Get the top element. \*/

public int top() {

return qa.peek();

}

/\*\* Returns whether the stack is empty. \*/

public boolean empty() {

return qa.isEmpty();

}

}

### Time and space complexity

Time - o(n) for push, o(1) for pop and peek.

Space – one queue needed only.

# 7. Sorting

**Stable algorithm** - Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

**In-place algorithm** is an algorithm which transforms input using no auxiliary data structure. However a small amount of extra storage space is allowed for auxiliary variables.

An **adaptive algorithm** is an algorithm that changes its behavior at the time it is run, based on information available and on a priori defined reward mechanism. E.g. shell sort. It takes into consideration some part of data already sorted. and hence work faster on such cases where such type of chunk exists.

**HYBRID ALGORITHM** - it is combination of two or more sorting algorithm to take advantage of both Algorithm.

E.g. 1. *TIMSORT* - *insertion sort + merge sort.*

\* As Insertion sort is faster than both merge and quick sort if elements are small e.g. <10.

\* For large data set it uses merge sort. Collections.sort uses it as in case of linked list no extra memeory needed in merge operation.

E.g. 2. *INTROSORT* - *Quicksort + Heapsort.*

\* As bad pivot selection can lead to O(n^2) in quick sort worst case. it uses hybrid of both 1. Insertion sort in array

## 1. Insertion Sort in array

### Problem

Sort the given 1 dimension array using insertion sort.

### Reference

ARRAY, GEEKSFORGEEKS

### Approach

1. On small data set insertion sort, selection sort works better than merge or quick sort.
2. It is just like arranging cards.
3. Insertion sort requires more swap as compared to selection sort and hence is not preferred where write operation is costly. On other hand insertion sort is more efficient than bubble or selection as it speeds up when array is already partially sorted.
4. Insert ith element in between 0 to i-1 position where it needed to be present. In general Select an item and on the left of that item keep sorted data and on right keep on considering and place it in correct place on left.
5. In-place, Iterative, adaptive, stable.

### Solution

for (int i = 1; i < array.length; i++) {

for (int j = 0; j < i; j++) {

if (array[i]<array[j]) {

swap(array, i, j);

}

}

}

### Time and space complexity

Time - o(n^2)

Space – o(1)

## 2. Insertion Sort in Single Linked list

### Problem

Sort the given Single linked list using insertion sort.

### Reference

SINGLE LINKED LIST, LEETCODE, GEEKSFORGEEKS

### Approach

Here we will create a dummy list and start adding element to in it in sorted order.

1. curr node will point to current iteration element.

2. prev will point to previous node after which element will be needed to insert

3. dummy - use to hold the result modified sorted list.

Steps -

1. Start curr from head and set dummy as least integer and prev also.

2. If item to be inserted is greater then prev node. we need to start searching from start i.e. from dummy. so, set prev=dummy

3. Now loop till we find position of new item. i.e. where it is less then prev.next or prev.next is null.

In this way we now know after prev we can add that item as after that other elements are greater.

4. Now just add element between prev and prev.next

5. Update curr node to next element of the loop.

### Solution

public static Node<Integer> insertionSort(Node<Integer> head) {

Node<Integer> dummy = new Node<Integer>(Integer.MIN\_VALUE);

Node<Integer> curr = head;

Node<Integer> prev = dummy;

while (curr != null) {

// Store nextNode for next iteration

Node<Integer> nextNode = curr.getNext();

// to save checking from start- below condition is used

if (prev.getData() > curr.getData()) {

prev = dummy;

}

// go to a point where we need to insert new item starting from prev.

while (prev.getNext() != null && prev.getNext().getData() < curr.getData()) {

prev = prev.getNext();

}

// insert current node between prev and prev.next

curr.setNext(prev.getNext());

prev.setNext(curr);

curr = nextNode;

}

return dummy.getNext();

}

### Time and space complexity

Time - o(n^2)

Space – o(1)

## Selection Sort in array

### Problem

On a given array apply selection sort to sort the data.

### Reference

ARRAY, LEETCODE, GEEKSFORGEEKS

### Approach

1. In it we find min in array and swap it with 0 index and then start searching min again from 1 to n and now swap min with 1 and so on.

2. It is an in-place algorithm

### Solution

for (int i = 0; i < array.length; i++) {

int minIndex = i;

for (int j = i + 1; j < array.length; j++) {

if (array[j].compareTo(array[minIndex]) < 0) {

minIndex = j;

}

}

swap(array, i, minIndex);

}

### Time and space complexity

Time - O(n^2)

Space – O(1)

## Selection Sort in Single linked list

### Problem

On a given single linked list apply selection sort to sort the data.

### Reference

SINGLE LINKED LIST, GEEKSFORGEEKS

### Approach

1. In it we find min in Linked list and swap it with 0 index and then start searching min again from 1 to n and now swap min with 1 and so on.
2. For swapping we will swap content of the data not the node itself.

3. It is an in-place algorithm

### Solution

public static Node<Integer> selectionSort(Node<Integer> head) {

Node<Integer> temp = head;

while (temp != null) {

Node<Integer> minNode = temp;

Node<Integer> dummy = temp.getNext();

while (dummy != null) {

if (dummy.getData() < minNode.getData()) {

minNode=dummy;

}

dummy=dummy.getNext();

}

int data = minNode.getData();

minNode.setData(temp.getData());

temp.setData(data);

temp = temp.getNext();

}

return head;

}

### Time and space complexity

Time - O(n^2)

Space – O(1)

## Bubble Sort in array

### Problem

Apply bubble sort algorithm to sort the array.

### Reference

ARRAY

### Approach

1. Compare 0 element with 1 and arrange them. Then it take 1 element with 2 and arrange.

So after 1st iteration largest element moved to the end of array.

From next iteration onwards we will start from 0 and ignore last element as it is already in the correct position.

2. Stable, In-place

### Solution

for (int i = 0; i < array.length-1; i++) {

// if after entire below loop no swap happen then it means array is already sorted

boolean swap=false;

for (int j = 0; j < array.length - i -1; j++) {

if (array[j].compareTo(array[j+1]) > 0) {

swap(array, j, j+1);

swap=true;

}

}

//break as array is now sorted.

if(!swap) {

break;

}

}

### Time and space complexity

Time - O(n^2)

Space – O(1)

## Bubble Sort in Single Linked list

### Problem

Apply bubble sort algorithm to sort the linked list.

### Reference

SINGLE LINKED LIST

### Approach

1. Compare 0 element with 1 and arrange them. Then it take 1 element with 2 and arrange.

So after 1st iteration largest element moved to the end of array.

From next iteration onwards we will start from 0 and ignore last element as it is already in the correct position.

2. Stable, In-place

3. We will swap data instead of swapping nodes for simpler solution

### Solution

public static Node<Integer> bubbleSort(Node<Integer> head) {

Node<Integer> end = null;

while (end != head) {

Node<Integer> next = head;

while (next.getNext() != null && next.getNext() != end) {

if (next.getData() > next.getNext().getData()) {

int t = next.getData();

next.setData(next.getNext().getData());

next.getNext().setData(t);

}

next = next.getNext();

}

end = next;

}

return head;

}

### Time and space complexity

Time - O(n^2)

Space – O(1)

## Merge Sort in array

Merge sort has best and worst case both as nlogn. But it takes extra memory. On other hand quick sort has avg case nlogn but worst case n^2

### Problem

Apply merge sort algorithm to sort the array.

### Reference

ARRAY, GEEKSFORGEEKS

### Approach

1. divide and conquer strategy
2. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.



2. Stable, take extra memory

### Solution

public static void mergeSort(int[] arr) {

mergeSort(arr, 0, arr.length - 1);

}

private static void mergeSort(int[] arr, int i, int j) {

if (i < j) {

int mid = (i + j) / 2;

mergeSort(arr, i, mid); // divide left sub array

mergeSort(arr, mid + 1, j); // divide right sub array

merge(arr, i, j, mid); // merge the two sorted array.

}

}

private static void merge(int[] arr, int l, int r, int mid) {

int[] temp = new int[r - l + 1];

int i = l;

int j = mid + 1;

int count = 0;

while (i <= mid && j <= r) {

if (arr[i] <= arr[j]) {

temp[count++] = arr[i++];

} else {

temp[count++] = arr[j++];

}

}

while (i <= mid) {

temp[count++] = arr[i++];

}

while (j <= r) {

temp[count++] = arr[j++];

}

for (int p = l; p <= r; p++) {

arr[p] = temp[p - l];

}

}

### Time and space complexity

Time - O(nlogn)

Space – O(n)

## 8. Merge Sort in linked list

Merge sort has best and worst case both as nlogn. It takes space O(logn) in linked list.

### Problem

Apply merge sort algorithm to sort the single linked list.

### Reference

ARRAY, LEETCODE

### Approach

1. divide and conquer strategy
2. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.
3. So, basically we will find middle element of the list and detach middle.next

So, first half head will have data till middle and middleNext element will be the starting point of second list.

And then we merge them together (for merging we can use both iterative solution as well as recursive solution)

### Solution

public static Node<Integer> mergeSort(Node<Integer> head) {

if (head == null || head.getNext() == null) {

return head;

}

//find middle element

Node<Integer> middle = findMiddleOfLinkedList(head);

//point middleNext to the start of second half of list

Node<Integer> middleNext = middle.getNext();

//set end of first half to null. (for clear separation of two list)

middle.setNext(null);

//now call merge sort for first half

Node<Integer> left = mergeSort(head);

//call merge sort for second half

Node<Integer> right = mergeSort(middleNext);

// merge sorted list

return mergeSortedLists(left, right);

}

private static Node<Integer> mergeSortedLists(Node<Integer> first, Node<Integer> second) {

Node<Integer> dummy = new Node<>(Integer.MAX\_VALUE);

Node<Integer> head=dummy;

while (first != null && second != null) {

if (first.getData() <= second.getData()) {

dummy.setNext(first);

first = first.getNext();

} else {

dummy.setNext(second);

second = second.getNext();

}

dummy = dummy.getNext();

}

while (first != null) {

dummy.setNext(first);

first = first.getNext();

dummy = dummy.getNext();

}

while (second != null) {

dummy.setNext(second);

second = second.getNext();

dummy = dummy.getNext();

}

return head.getNext();

}

public static Node<Integer> findMiddleOfLinkedList(Node<Integer> head) {

if (head == null) {

return head;

}

Node<Integer> fast = head;

Node<Integer> slow = head;

while (fast.getNext() != null && fast.getNext().getNext() != null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

}

return slow;

}

### Time and space complexity

Time - O(nlogn)

Space – O(logn) (this is because of max element in stack at any given point can be logn)

## 9. Quick Sort in array

Quick sort has avg-case as nlogn and worst case as n^2.

In-place, Not Stable,Not adaptive,Divide and conquer. It picks an element as pivot and partitions the given array around the picked pivot.

Perform quick sort algo to sort the data. for primitive it is preferred and for objects merge sort is preferred.

as merge sort takes extra o(n) memory it is not preferred for array.

but for linked list merge sort does not need extra space

**Steps**-

We will take one element as pivot(here the last one) and will try to put it in right position.

By right position we mean that all element to the left of it are small and right to it are greater than pivot.

So, basically we fixing pivot position one by one.

### Problem

Apply quick sort algorithm to sort array.

### Reference

ARRAY, LOCAL\_GIT

### Approach

**Partition –**

In partition algorithm we fix the pivot (here high index element) and if current element is less than pivot

We swap current element with low and increment low by 1.

Once loop terminates low element will be at index where high should be present so we swap it.

1. select high index element as pivot.

2. go from j=low to high-1.

3. check arr[j] < arr[high] . i.e. current element is smaller than pivot if yes.

4. swap j with low. and increment low by 1.

after loop terminates swap high with low. and return low which is our new pivot position.

e.g.

15, 17, 13, 6, 14

low high

j

step 1 -

15, 17, 13, 6, 14

low j high

15, 17, 13, 6, 14

low j high

13, 17, 15, 6, 14

low j high

13, 6, 15, 17, 14

low high

So,now swap high with low. and return low i.e. index 2.

So,that we will have two partition {13,6} {17,14} now and index 2 element is fixed

In this algorithm to save variable we will consider high as pivot.

### Solution

public void quickSort(int[] arr,int i, int j){

if (i < j) {

int p = partition(arr, i, j);

quickSort(arr, i, p-1);

quickSort(arr, p+1, j);

}

}

private int partition(int[] arr,int l,int h) {

int mid = (l+h)/2;

swap(arr,mid,h);

for(int i=l; i<h;i++){

if(arr[i]<arr[h]){

swap(arr,i,l);

l++;

}

}

swap(arr,l,h);

return l;

}

private void swap(int[] a, int i,int j){

int temp=a[i];

a[i]=a[j];

a[j]=temp;

}

### Time and space complexity

Avg Time - O(nlogn), worst case – O(n^2)

Space – O(1)

# 8. Selection problem

## 1. Find kth smallest element in array (Quick Select)

### Problem

Find kth smallest element in given array

### Reference

ARRAY, LEETCODE

### Approach

1. It is just little modification in quick sort algorithm.
2. Partition will remain same. And if index returned by partition is equal to k that means arr[k] is the kth smallest
3. If k is less than index received. It means we need to search between left, index-1.
4. Else search between index+1,right

### Solution

public int findkMinElement(int[] arr, int k) {

return kthSmallest(arr, 0, arr.length - 1, k - 1);

}

private int kthSmallest(int[] arr, int l, int r, int k) {

int p = partition(arr, l, r);

if (k == p) {

return arr[p];

} else if (k < p) {

return kthSmallest(arr, l, p - 1, k);

}

return kthSmallest(arr, p + 1, r, k);

}

public int partition(int[] arr, int start, int end) {

int initPivot = (start + end) / 2;

swap(arr, initPivot, end);

// after that end index will be our pivot. and we will put pivot to its correct

// position after loop terminates

for (int i = start; i < end; i++) {

if (arr[i] < arr[end]) {

swap(arr, i, start);

start++;

}

}

swap(arr, start, end);

return start;

}

private void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

### Time and space complexity

Time - The worst case time complexity of this method is O(n2), but it works in O(n) on average.

Space - logn

# Bitwise operator

<https://www.youtube.com/watch?v=pv1C0_6k78A>

**1 Left shift <<**

a<<b will appends b number of zero to the end of binary equivalent of a. and return the decimal output.

It has similar effect as of multiplying the number ‘a’ with ‘b’ power of two.

For example –

a = 5 = 00000101

a << 1 = 00001010 = 10 (add 1 zero to last)

a << 2 = 00010100 = 20 (add 2 zero to last)

Basically a<<1 is multiplying a with 2^1 and a<<2 is multiplying a with 2^2

**2 Right shift >>**

a>>b will remove b number of digits from the end of binary equivalent of a. and return the decimal output.

It has similar effect as of dividing the number ‘a’ with ‘b’ power of two.

For example –

a = 10 = 00001010

a >> 1 = 00000101 = 5 (remove last digit. here it is 0)

a >> 2 = 00000010 = 2 (remove last two digits. here it is 10)

Basically a>>1 is dividing ‘a’ with 2^1 and a>>2 is dividing ‘a’ with 2^2.

**3 Bitwise OR (|)**

It returns bit by bit OR of input values, i.e, if either of the bits is 1, it gives 1, else it gives 0.

For example,

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise OR Operation of 5 and 7

0101

| 0111

\_\_\_\_\_\_

0111 = 7 (In decimal)

**4 Bitwise AND &**

It returns bit by bit AND of input values, i.e., if both bits are 1, it gives 1, else it gives 0.

For example,

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise AND Operation of 5 and 7

0101

& 0111

\_\_\_\_\_\_\_\_

0101 = 5 (In decimal)

**5 Bitwise XOR (^)**

It returns bit by bit XOR of input values, i.e, if corresponding bits are different, it gives 1, else it gives 0.

For example,

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise XOR Operation of 5 and 7

0101

^ 0111

\_\_\_\_\_\_\_

0010 = 2 (In decimal)

**6 Bitwise Complement (~)**

It returns the one’s compliment representation of the input value, i.e, with all bits inversed, means it makes every 0 to 1, and every 1 to 0.

a = 5 = 0101 (In Binary)

Bitwise Compliment Operation of 5

~ 0101

\_\_\_\_\_\_\_\_

1010 = 10 (In decimal)

## 1. Convert Binary Number in a Linked List to Integer

### Problem

Given head which is a reference node to a singly-linked list. The value of each node in the linked list is either 0 or 1. The linked list holds the binary representation of a number.

Return the *decimal value* of the number in the linked list.

**Example 1:**



**Input:** head = [1,0,1]

**Output:** 5

**Explanation:** (101) in base 2 = (5) in base 10

### Reference

LEETCODE, GEEKSFORGEEKS

### Approach

1. start from head and keep on multiplying result with 2.
2. Result=result\*2 + head.val

For 100

Result = 0\*2+1=1

Result=1\*2+0=2

Result=2\*2+0=4

1. for multiplication by 2 we can use result<<1 also.

### Solution

public int getDecimalValue(ListNode head) {

int number = 0;

while(head!=null){

number=(number\*2)+head.val;

head=head.next;

}

return number;

}

### Time and space complexity

Time - O(n)

Space - O(1)

# 10. Binary Tree

1. A Tree with max 2 child’s are binary tree
2. Complete/full binary tree has all nodes with 2 child except leaf.
3. Binary Search Tree has property that left child is smaller than root and root is smaller than right.
4. Trees can be traversed in different ways. Following are the generally used ways for traversing trees.



*Example Tree*

1. Depth First Traversals:  
   (a) Inorder (Left, Root, Right) : It is just like printing node when visited second time

4 2 5 1 3  
(b) Preorder (Root, Left, Right): It is just like printing node when visited first time

1 2 4 5 3  
(c) Postorder (Left, Right, Root): It is just like printing node when visited third time

4 5 2 3 1

1. Breadth First or Level Order Traversal : printing node left to right level-wise from top to bottom.

1 2 3 4 5

1. Skewed binary tree is equivalent of single linked list.
2. To construct a unique binary tree we need in order traversal and any of these (pre-order/post order/level order)

**Therefore, following combination can uniquely identify a tree.**

Inorder and Preorder.

Inorder and Postorder.

Inorder and Level-order.

**And following do not.**

Postorder and Preorder.

Preorder and Level-order.

Postorder and Level-order.

1. Number of nodes in complete binary tree of height h is -> 2^(h+1) -1

18

/ \

15 30

/ \ / \ here h=2, so number of nodes => 2^3 - 1 => 7

40 50 100 40

1. Number of unordered binary tree possible with N Nodes-> 2ncn/(n+1). For 3 nodes 6c3/4

(6\*5\*4)/(4\*3\*2\*1) = 5

For n = 2, there are two trees

o o

/ \

o o

For n = 3, there are five trees

o o o o o

/ \ / \ / \

o o o o o o

/ \ \ /

o o o o

1. Number of ordered binary tree possible with N Nodes->

n!(2ncn/(n+1))

In above question n=3, total possible tree = 30 as every node can be root. So, 3! Possible combinations.

1. In Binary Tree every operation require entire tree traversal.
2. Degree of a node is the number of descendants of a node. If the degree is zero, it is called leaf node of a tree.

## Pre-Order Traversal Recursive

### Problem

Given a binary tree, return the preorder traversal of its nodes' values.

### Reference

LEETCODE, GEEKSFORGEEKS, Ravindra Babu

### Approach

PreOrder (Root, Left, Right)

1

/ \

2 3

/ \

4 5

Output - 12453

i.e. whenever we visit node first time we print it.

Algorithm Preorder(tree)

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left-subtree)

3. Traverse the right subtree, i.e., call Preorder(right-subtree)

### Solution

**private** **void** preOrder(TreeNode<T> node) {

**if** (node != **null**) {

System.out.println(node.getData());

preOrder(node.getLeft());

preOrder(node.getRight());

}

}

### Time and space complexity

O(n)

O(n)

## Pre-Order Traversal Iterative

### Problem

Given a binary tree, return the pre-order traversal of its nodes' values.

### Reference

LEETCODE, GEEKSFORGEEKS

### Approach

1. It can be implemented using stack. We use our stack to hold objects just like in recursion.
2. Just add right child first and then left. As by this left object will be at the top of stack and popped first.
3. Pop all items one by one. Do following for every popped item

a) print it

b) push its right child

c) push its left child

Note that right child is pushed first so that left is processed first

### Solution

**public** **void** printPreOrderIterative(TreeNode<Integer> node) {

**if** (node == **null**) {

**return**;

}

Deque<TreeNode<Integer>> stack = **new** LinkedList<>();

stack.push(node);

**while** (!stack.isEmpty()) {

node = stack.pop();

System.***out***.print(node.getData() + " ");

**if** (node.getRight() != **null**) {

stack.push(node.getRight());

}

**if** (node.getLeft() != **null**) {

stack.push(node.getLeft());

}

}

System.***out***.println();

}

### Time and space complexity

O(n)

O(n)

## In-Order Traversal Recursive

### Problem

Given a binary tree, return the in-order traversal of its nodes' values.

### Reference

LEETCODE, GEEKSFORGEEKS, Ravindra Babu

### Approach

Inorder (Left, Root, Right)

1

/ \

2 3

/ \

4 5

Output - 42513

i.e. whenever we visit node second time we print it.

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call Inorder(right-subtree)

### Solution

void printInorder(Node node) {

        if (node == null)

            return;

        printInorder(node.left);

        System.out.print(node.key + " ");

        printInorder(node.right); }

### Time and space complexity

O(n)

O(n)

## 4. In-Order Traversal Iterative

### Problem

Given a binary tree, return the in-order traversal of its nodes' values.

### Reference

LEETCODE, GEEKSFORGEEKS

### Approach

1. It can be implemented using stack. We use our stack to hold objects just like in recursion.

2. We will push all left nodes first and after that pop element which will be the last left node.

3. Print node and set current right. Now again repeat above step for current node

### Solution

**public** **void** printInOrderIterative(TreeNode<Integer> node) {

Deque<TreeNode<Integer>> st = **new** LinkedList<>();

**while** (!st.isEmpty() || node != **null**) {

**while** (node != **null**) {

st.push(node);

node = node.getLeft();

}

node = st.pop();

System.***out***.print(node.getData() + " ");

node = node.getRight();

}

System.***out***.println();

}

### Time and space complexity

O(n)

O(n)

## 5. Post-Order Traversal Recursive

### Problem

Given a binary tree, return the post order traversal of its nodes' values.

### Reference

LEETCODE, GEEKSFORGEEKS, Ravindra Babu

### Approach

postorder (Left, Root, Right)

1

/ \

2 3

/ \

4 5

Output - 45231

i.e. whenever we visit node third time we print it.

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)

2. Traverse the right subtree, i.e., call Postorder(right-subtree)

3. Visit the root.

### Solution

private void postorder(TreeNode root){

if(root!=null){

postorder(root.left);

postorder(root.right);

System.out.println(root.val);

}

}

### Time and space complexity

O(n)

O(n)

## Post-Order Traversal Iterative

### Problem

Given a binary tree, return the post order traversal of its nodes' values.

### Reference

LEETCODE, GEEKSFORGEEKS

### Approach

It is just like reversing the output of preorder traversal and hence we follow same algo as pre-order traversal iterative and to reverse the output we will take another stack.so, that when we pop from it data will be opposite

### Solution

**public** **void** printPostOrderIterative(TreeNode<Integer> node) {

**if** (node == **null**) {

**return**;

}

Deque<TreeNode<Integer>> stack = **new** LinkedList<>();

Deque<TreeNode<Integer>> result = **new** LinkedList<>();

stack.push(node);

**while** (!stack.isEmpty()) {

node = stack.pop();

result.push(node);

**if** (node.getLeft() != **null**) {

stack.push(node.getLeft());

}

**if** (node.getRight() != **null**) {

stack.push(node.getRight());

}

}

**while** (!result.isEmpty()) {

System.***out***.print(result.pop().getData() + " ");

}

System.***out***.println();

}

### Time and space complexity

O(n)

O(2n)

## Level-Order Traversal

### Problem

Given a binary tree, return the level order traversal of its nodes' values.

### Reference

LEETCODE, GEEKSFORGEEKS, Iterative

### Approach

1. We will use queue data structure to store the node.
2. Get poll item from queue and For every node – first print data and then put left and right node in queue.

### Solution

**public** **void** printLevelOrder(TreeNode<Integer> root) {

**if** (root == **null**) {

**return**;

}

Queue<TreeNode<Integer>> queue = **new** LinkedList<>();

queue.add(root);

System.***out***.println("Level Order Traversal : ");

**while** (!queue.isEmpty()) {

TreeNode<Integer> node = queue.poll();

System.***out***.print(node.getData() + " ");

**if** (node.getLeft() != **null**) {

queue.add(node.getLeft());

}

**if** (node.getRight() != **null**) {

queue.add(node.getRight());

}

}

System.***out***.println();

}

### Time and space complexity

O(n)

O(n)

## Zig-Zag Order Traversal Iterative

### Problem

Given a binary tree, return the Zig-Zag traversal of its nodes' values. For the below binary tree the zigzag order traversal will be **1 3 2 7 6 5 4**



### Reference

LEETCODE, GEEKSFORGEEKS

### Approach

1. print data in level order traversal wise.
2. just print data left to right and right to left alternatively take two stacks.
3. one that is holding current level data and second using next level data.
4. just on alternate levels add left child first and then right child first.
5. in this way zig zag order will be maintained.
6. boolean is kept to alter the level once current level finished.
7. in case of current level is empty just swap it with with next level stack. As we are popping from current only.

### Solution

**public** **void** ZigZagTraversal(TreeNode<Integer> root) {

**if** (root == **null**) {

**return**;

}

Deque<TreeNode<Integer>> current = **new** LinkedList<>();

Deque<TreeNode<Integer>> next = **new** LinkedList<>();

current.push(root);

**boolean** leftToRight = **true**;

System.***out***.println("Zig Zag order traversal is : ");

**while** (!current.isEmpty()) {

TreeNode<Integer> node = current.pop();

System.***out***.print(node.getData() + " ");

**if** (leftToRight) {

**if** (node.getLeft() != **null**) {

next.push(node.getLeft());

}

**if** (node.getRight() != **null**) {

next.push(node.getRight());

}

} **else** {

**if** (node.getRight() != **null**) {

next.push(node.getRight());

}

**if** (node.getLeft() != **null**) {

next.push(node.getLeft());

}

}

**if** (current.isEmpty()) {

leftToRight = !leftToRight;

Deque<TreeNode<Integer>> temp = current;

current = next;

next = temp;

}

}

System.***out***.println();

}

### Time and space complexity

O(n)

O(n)

## 9. Binary Tree Tilt

### Problem

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 :

4

/ \

2 9

/ \ \

3 5 7

Output : 15

Explanation:

Tilt of node 3 : 0

Tilt of node 5 : 0

Tilt of node 7 : 0

Tilt of node 2 : |3-5| = 2

Tilt of node 9 : |0-7| = 7

Tilt of node 4 : |(3+5+2)-(9+7)| = 6

Tilt of binary tree : 0 + 0 + 0 + 2 + 7 + 6 = 15

### Reference

LEETCODE, GEEKSFORGEEKS

### Approach

1. Will use recursive solution to get the tilt value of full tree.
2. Will take global variable sum to hold the total sum.
3. Call left tree and then right.
4. we will store the total tilt value inside sum variable and return value of current node
5. for e.g. will return 0+0+3 as left and 0+0+5 as right for node 2
6. So, will update sum= 0+|3-5| => 2 for node 2.
7. We will do for all the nodes like this.

### Solution

class Model{

int sum;

}

public int findTilt(TreeNode root) {

Model m = new Model();

findTilt(root,m);

return m.sum;

}

private int findTilt(TreeNode root,Model m){

if(root==null){

return 0;

}

int l = findTilt(root.left,m);

int r = findTilt(root.right,m);

m.sum = Math.abs(l-r)+m.sum;

return l+r+root.val;

}

### Time and space complexity

O(n)

O(n)

## 11. Number of nodes in binary tree

### Problem

Find number of nodes in binary tree.

### Reference

LEETCODE, Recursive, Ravindra Babu ravula

### Approach

1. For recursive approach –
2. We can divide problem into subparts where total nodes will be root+ leftchildOfRoot + rightchildOfRoot.
3. So, totalnodes(left)+totalnodes(right)+1 ( 1 is added for root)

1. if node is null return 0

2. else return totalnodes(left)+totalnodes(right)+1

### Solution

public TreeNode totalNodes(TreeNode root){

if(root==null) return 0;

return 1+totalNodes(root.left)+totalNodes(root.right);

}

### Time and space complexity

O(n)

O(n)

## 12. Number of leaves in binary tree

### Problem

Find number of leaves in binary tree.

### Reference

LEETCODE, Recursive, Ravindra Babu ravula

### Approach

Since we don’t want to count any other nodes accept leaf. We will return 1 if current node is leaf

1. if node is null return 0
2. if node.left and right both null return 1
3. else return totalnodes(left)+totalnodes(right)

Similarly if we want to calculate non leaf nodes just in step 1,2 return 0 as we don’t want to count these nodes and return total(left)+total(right)+1;

### Solution

public TreeNode numberOfLeaf(TreeNode root) {

if (node == null) {

return 0;

}

if (node.getLeft() == null && node.getRight() == null) {

return 1;

}

return numberOfLeaf(node.getLeft()) + numberOfLeaf(node.getRight());

}

### Time and space complexity

O(n)

O(n)

## 13. Number of Full nodes in binary tree

### Problem

Find number of Full nodes in binary tree. A node is full node if it has both the child.

### Reference

LEETCODE, Recursive, GEEKSFORGEEKS,GITHUB

### Approach

1. Since we don’t want to count any other nodes accept full. We need to add 1 and 0 conditionally to recursive call.
2. if node is null return 0
3. if node.left and right both not null we will add 1 to count. We have not returned for here as in such case we wanted to still traverse the child for further counting. If we return 1 directly then program terminates directly in first attempt itself as root always has left and right child not null.
4. else return totalnodes(left)+totalnodes(right) +c;

### Solution

**public** **int** numberOfFullNodes(TreeNode<T> head) {

**if** (head == **null**) {

**return** 0;

}

**int** c = 0;// don’t count node by default

**if** (head.getLeft() != **null** && head.getRight() != **null**) {

c = 1;//this is our node and hence count it and process child further

}

**return** c + numberOfFullNodes(head.getLeft()) + numberOfFullNodes(head.getRight());

}

### Time and space complexity

O(n)

O(n)

## 14. Maximum Depth/height in binary tree

### Problem

Find height of the binary tree. (maximum depth of binary tree).

For root=null height will be 0

For 1 node tree height will be 1

### Reference

LEETCODE, GEEKSFORGEEKS, Recursive, Iterative, **Ravindra Babu ravula**

### Approach

1. For iterative approach – We will use level order traversal and keep incrementing level for each level.
2. For recursive approach

1. if node is null return 0

2. else return max(left,right)+1

### Solution

Iterative -

class Solution {

public int maxDepth(TreeNode root) {

if(root==null){

return 0;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);

int level = 0;

while(!queue.isEmpty()){

int size = queue.size();

level++;

for(int i=0;i<size;i++){

TreeNode node = queue.poll();

if(node.left!=null){

queue.add(node.left);

}

if(node.right!=null){

queue.add(node.right);

}

}

}

return level;

}

}

Recursive -

class Solution {

public int maxDepth(TreeNode root) {

if(root==null){

return 0;

}

return Math.max(maxDepth(root.left),maxDepth(root.right) + 1;

}

}

### Time and space complexity

O(n)

O(n)

## 15. Minimum Depth in binary tree iteratively

### Problem

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.

**Example:**

Given binary tree [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

return its minimum depth = 2.

### Reference

LEETCODE, GEEKSFORGEEKS, Iterative

### Approach

1. Iterative approach – We will use level order traversal and keep incrementing level for each level.
2. In side loop if we found any node with left and right both child null. Return level. It is the minimum depth.

### Solution

public int minDepth(TreeNode root) {

if(root==null){

return 0;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);

int level = 0;

while(!queue.isEmpty()){

int size = queue.size();

level++;

for(int i=0;i<size;i++){

TreeNode node = queue.poll();

if(node.left==null && node.right==null){

return level;

}

if(node.left!=null){

queue.add(node.left);

}

if(node.right!=null){

queue.add(node.right);

}

}

}

return level;

}

### Time and space complexity

O(n)

O(n)

## 16. Maximum element in binary tree

### Problem

Find maximum element in a binary tree.

### Reference

Recursive, GITHUB

### Approach

1. Call max method recursively for left and right.
2. Then find maximum element from left tree and right tree. Find max out of these two.
3. Compare max with root and return max.

For Iterative Solution – use level order traversal approach and keep track of max. after queue is empty max can be returned.

### Solution

**public** Integer findMax(TreeNode<Integer> root) {

**if** (root == **null**) {

**return** -1;

}

**int** lmax = *findMax*(root.getLeft());

**int** rmax = *findMax*(root.getRight());

**int** max = lmax > rmax ? lmax : rmax;

**return** max > root.getData() ? max : root.getData();

}

### Time and space complexity

O(n)

O(n)

## 17. Find Node in a binary tree

### Problem

Given a Binary Node and a item. The task is to search and check if the given item exits in the binary tree or not.

return Node if exists else return null.

### Reference

Recursive, GITHUB

### Approach

1. Check if current node is item if yes return node.
2. Call recursively for left node. If value returned is null call right else return directly.

### Solution

**public** TreeNode<Integer> findNode(TreeNode<Integer> root, **int** item) {

**if** (root == **null**) {

**return** **null**;

}

**if** (root.getData() == item) {

**return** root;

}

TreeNode<Integer> res = *findNode*(root.getLeft(), item);

**if** (res == **null**) {

res = *findNode*(root.getRight(), item);

}

**return** res;

}

### Time and space complexity

O(n)

O(n)

## 18. Insert item in a binary tree

### Problem

Given a Binary Node and item. The task is to add item in a binary tree.

### Reference

Iterative, GITHUB

### Approach

We can add item in an optimized way. Use level order approach and as soon as we find empty left or right child we insert item there and break the loop.

### Solution

**public** **void** insert(TreeNode<T> root,T data) {

TreeNode<T> node = **new** TreeNode<>(data, **null**, **null**);

**if** (root == **null**) {

root = node;

} **else** {

insertNode(root, node);

}

size++;

}

**private** **void** insertNode(TreeNode<T> node, TreeNode<T> newNode) {

Queue<TreeNode<T>> queue = **new** LinkedList<>();

queue.add(node);

**while** (!queue.isEmpty()) {

**int** size = queue.size();

**for** (**int** i = 0; i < size; i++) {

TreeNode<T> next = queue.poll();

**if** (next.getLeft() == **null**) {

next.setLeft(newNode);

**break**;

} **else** {

queue.add(next.getLeft());

}

**if** (next.getRight() == **null**) {

next.setRight(newNode);

**break**;

} **else** {

queue.add(next.getRight());

}

}

}

}

### Time and space complexity

O(n)

O(n)

## Level-Order Traversal Reverse Order

### Problem

Given a binary tree, return the level order traversal of its nodes' in reverse order.

### Reference

LEETCODE, GITHUB, Iterative

### Approach

1. Same as level order except printing the node we will add that node into stack.
2. After all nodes traversed. Just start popping item from stack and printing it.

### Solution

**public** **void** printLevelOrderReverse(TreeNode<Integer> root) {

**if** (root == **null**) {

**return**;

}

Queue<TreeNode<Integer>> queue = **new** LinkedList<>();

queue.add(root);

Deque<Integer> stack = **new** LinkedList<>();

**while** (!queue.isEmpty()) {

TreeNode<Integer> node = queue.poll();

stack.push(node.getData());

**if** (node.getRight() != **null**) {

queue.add(node.getRight());

}

**if** (node.getLeft() != **null**) {

queue.add(node.getLeft());

}

}

System.***out***.println("Level Order Traversal in reverse order : ");

**while** (!stack.isEmpty()) {

System.***out***.print(stack.pop() + " ");

}

System.***out***.println();

}

### Time and space complexity

O(n)

O(n)

## Diameter of Binary tree

### Problem

Given a binary tree, return the Diamater.

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two end nodes. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



### Reference

GEEKSFORGEEKS, GITHUB, Recursive

### Approach

The diameter of a tree T is the largest of the following quantities:

\* the diameter of T’s left subtree

\* the diameter of T’s right subtree

\* the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T).

**Approach 1 - takes o(n2)**

1. So, one approach can be to calculate height of left and height of right
2. Then find diameter of left and diameter of right.
3. After that find max(height\_left+height\_right+1,max(diameter\_left,diameter\_right);

But in above approach we are traversing same node twice. And hence complexity increases to o(n2)

Better approach can be to iterate every node single time and keep track of diameter in global variable.

**Approach 2 – takes o(n) and simple**

\* Diameter of a tree can be calculated by only using the height function,

\* because the diameter of a tree is nothing but maximum value of (left\_height +

\* right\_height + 1) for each node.

So we just need to calculate height of left and right of each node and update global max diameter variable if that current diameter i.e. left\_height +right\_height + 1 is greater than previous one. So, once loop finishes max global variable will have maximum diameter.

### Solution

Approach 1 –

**public** **int** getDiameter(TreeNode<Integer> root) {

**if** (root == **null**) {

**return** 0;

}

**int** res = *getHeight*(root.getLeft()) + *getHeight*(root.getRight());

**int** l = *getDiameter*(root.getLeft());

**int** r = *getDiameter*(root.getRight());

**return** MathUtil.*max*(res + 1, Math.*max*(l, r));

}

**public** **int** getHeight(TreeNode<Integer> root) {

**if** (root == **null**) {

**return** 0;

}

**int** l = *getHeight*(root.getLeft());

**int** r = *getHeight*(root.getRight());

**return** MathUtil.*max*(l, r)+1;

}

Approach 2 -

**public** **int** diameterOptimized(TreeNode<Integer> root) {

**int**[] max = **new** **int**[1];

*diameterOptimized*(root, max);

**return** max[0];

}

**private** **int** diameterOptimized(TreeNode<Integer> root, **int**[] max) {

**if** (root == **null**) {

**return** 0;

}

**int** l = *diameterOptimized*(root.getLeft(), max);

**int** r = *diameterOptimized*(root.getRight(), max);

max[0] = MathUtil.*max*(l + r + 1, max[0]); //update max- checking current node height

**return** MathUtil.*max*(l, r) + 1;//return height

}

### Time and space complexity

O(n)

O(n)

# 11. Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties: The left subtree of a node contains only nodes with keys lesser than the node's key. The right subtree of a node contains only nodes with keys greater than the node's key.

200px-Binary_search_tree.svg

1. Searching is faster as compared to binary tree as we ignore half tree every time
2. In case of balanced BST time to find node – logn
3. In worst case if data added in sorted order in tree. It will became skew tree (like single linked list) & will take o(n)
4. AVL tree, Red Black Tree are balanced binary search tree which ensure every operation takes logn time
5. In order traversal of Binary Search Tree is sorted data.
6. If we have pre-order or postorder only we can construct a unique bst. This is because we can get in-order by sorting post order or pre order.
7. Whenever we need to traverse tree bottom up we can try in-order approach.

## 1. Search in Binary Search Tree

### Problem

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.

### Reference

LEETCODE, BINARY SEARCH TREE, BST, Recursive

### Approach

1. Call recursively till we match node.val with val
2. If val<root.val call searchBST(root.left,val) else searchBST(root.right,val)

### Solution

public TreeNode searchBST(TreeNode root, int val) {

if(root==null||root.val==val){

return root;

}

if(val<root.val){

return searchBST(root.left,val);

}

return searchBST(root.right,val);

}

### Time and space complexity

O(logn)- avg case and o(n) worst case

O(n)

## 2. Insert in Binary Search Tree

### Problem

Given the root node of a binary search tree (BST). You need to insert node in the BST

100 100

/ \ Insert 40 / \

20 500 ---------> 20 500

/ \ / \

10 30 10 30

\

40

### Reference

LEETCODE, BINARY SEARCH TREE, BST, Recursive

### Approach

1. Visit left tree if node.data > data else visit right.
2. Do that till we reach either left null or right null. Update corresponding left child or right child to newly created node.

### Solution

**public** TreeNode<Integer> insertNode(TreeNode<Integer> root, **int** data) {

**if** (root == **null**) {

**return** **new** TreeNode<>(data);

}

**if** (root.getData() > data) {

root.setLeft(*insertNode*(root.getLeft(), data));

} **else** {

root.setRight(*insertNode*(root.getRight(), data));

}

**return** root;

}

### Time and space complexity

O(logn)- avg case and o(n) worst case

O(n)

## 3. Construct Binary Search Tree from given pre-order and in-order

### Problem

Return the root node of a binary **search** tree that matches the given pre-order and in-order traversal.

Inorder sequence: D B E A F C  
Preorder sequence: A B D E C F

### Reference

LEETCODE, BINARY TREE, BST, Recursive

### Approach

1. In pre-order we know that first index is root and if we search that element in in-order we can find out all elements on left side of ‘A’ are in left subtree and elements on right are in right subtree.

So we know below structure now.

A

/ \

/ \

D B E F C

We recursively follow above steps and get the following tree.

A

/ \

/ \

B C

/ \ /

/ \ /

D E F

**Steps** –

1. Take element one by one from pre-order and create a new tree node with that element.
2. Find picked element index from inorder. Let it be inIndex.
3. Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.
4. Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.
5. return tNode.

To find the node we can use binary search algorithm or can store elements in hashmap with index as value.

### Solution

public TreeNode bstFromPreorder(int[] preorder,int[] inorder) {

return bst(preorder,inorder,0,preorder.length-1,new int[1]);

}

private TreeNode bst(int[] pre,int[] in,int l,int r,int[] c){

if(l>r) {

return null;

}

TreeNode node = new TreeNode(pre[c[0]]);

c[0]+=1;

int mid = find(in,node.val,l,r);

node.left=bst(pre,in,l,mid-1,c);

node.right=bst(pre,in,mid+1,r,c);

return node;

}

private int find(int[] in,int item,int l,int r){

while(l<=r){

int mid=(l+r)/2;

if(in[mid]==item){

return mid;

}

if(in[mid]<item){

l=mid+1;

} else {

r=mid-1;

}

}

return -1;

}

### Time and space complexity

O(n) if hashmap is used else additional logn time to search element using binary search.

O(n)

## 4. Construct Binary Search Tree from given pre-order

### Problem

**Return the root node of a binary search tree that matches the given preorder traversal.**

**Example 1:**

**Input:** [8,5,1,7,10,12]

**Output:** [8,5,10,1,7,null,12]



### Reference

LEETCODE, BINARY SEARCH TREE, BST, Recursive

### Approach

1. Approach 1 - Can be to first obtain in-order by sorting the array. And follow above question to get bst from in-order and pre-order. But in our case sorting take nlogn time. And this problem can be solved in o(n) time using approach 2.
2. Approach 2 – We know that first node of preorder will always be root. And second node from pre-order can be left or right. The rule to check is that we know that left node cannot be greater than parent. Keeping that in mind we design below algo –
3. The trick is to set a range {min .. max} for every node. Initialize the range as {INT\_MIN .. INT\_MAX}. The first node will definitely be in range, so create root node. To construct the left subtree, set the range as {INT\_MIN …root->data}. If a values is in the range {INT\_MIN .. root->data}, the values is part part of left subtree. To construct the right subtree, set the range as {root->data..max .. INT\_MAX}.

### Solution

**public** TreeNode<Integer> getBstFromPreOrder(**int**[] pre) {

**return** *getBstFromPreOrder*(pre, Integer.***MIN\_VALUE***, Integer.***MAX\_VALUE***, **new** **int**[1]);

}

**private** TreeNode<Integer> getBstFromPreOrder(**int**[] pre, **int** minValue, **int** maxValue, **int**[] c) {

**if** (c[0] >= pre.length) {

**return** **null**;

}

**int** key = pre[c[0]];

TreeNode<Integer> node = **null**;

**if** (key > minValue && key < maxValue) {

node = **new** TreeNode<>(pre[c[0]]);

c[0] += 1;

node.setLeft(*getBstFromPreOrder*(pre, minValue, key, c));

node.setRight(*getBstFromPreOrder*(pre, key, maxValue, c));

}

**return** node;

}

### Time and space complexity

O(n)

O(n)

# 12. Dynamic Programming

## 1. Maximum sum in Contiguous Sub-Array

### Problem

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

Input: [-2,1,-3,4,-1,2,1,-5,4],

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6.

### Reference

LEETCODE, UDEMY, DP

### Approach

\*Take global\_max which hold the max overall

\* And curr\_max will hold the max till curr iteration.

\* We will update curr\_max by this - store max of (current element, curr\_max+current element)

\* By this we make sure that either current is taken or previous one is included in contiguous space.

### Solution

public int maxSubArray(int[] nums) {

int curr\_max = nums[0];

int global\_max = nums[0];

for (int i = 1; i < nums.length; i++) {

curr\_max = MathUtil.max(nums[i], nums[i] + curr\_max);

if (curr\_max > global\_max) {

global\_max = curr\_max;

}

}

return curr\_max;

}

### Time and space complexity

O(n)

O(n)

# 13. Design data structure

## 1. Design LRU cache with put and get in o(1) time

### Problem

Design and implement a data structure for [Least Recently Used (LRU) cache](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU). 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.

The cache is initialized with a **positive** capacity.

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

### Reference

LEETCODE, GIT

### Approach

1. since we want to retrieve data in o(1) we will use hashmap. And we will use double-linked-list to store data.

2. Head and last are used to hold linked list.

3. we will remove from last and insert at head.

4. when we get item. In that case we remove it and insert node at head.

5. when we put if capacity is full. We remove from last and insert node at head.

6. to simplify exceptional cases like only last or only head is present or delete last or head element. We keep head and last as dummy value with 0 data.

7. so, we are always sure that head and last always exist and we use map.size to know current capacity.

### Solution

**public** **class** LRUCacheSimplified {

**class** Node {

**int** key;

**int** value;

Node prev;

Node next;

Node(**int** k, **int** v) {

key = k;

value = v;

}

}

**private** Map<Integer, Node> map = **new** HashMap<>();

**private** Node head;

**private** Node last;

**private** **int** capacity;

**public** LRUCacheSimplified(**final** **int** capacity) {

**this**.capacity = capacity;

head = **new** Node(0, 0);

last = **new** Node(0, 0);

head.next = last;

last.prev = head;

}

**public** **int** get(**int** key) {

Node node = map.get(key);

**if** (node == **null**) {

**return** -1;

}

remove(node);

insert(node);

**return** node.value;

}

**private** **void** insert(Node node) {

map.put(node.key, node);

node.next = head.next;

head.next.prev = node;

node.prev = head;

head.next = node;

}

**public** **void** put(**int** key, **int** value) {

**if** (map.containsKey(key)) {

remove(map.get(key));

}

**if** (map.size() == capacity) {

remove(last.prev);

}

insert(**new** Node(key, value));

}

**private** **void** remove(Node node) {

map.remove(node.key);

node.prev.next = node.next;

node.next.prev = node.prev;

}

}

### Time and space complexity

O(1) – for put and get

O(n), n is no. of the key stored in the hashmap and the doubly linked list

# 13. Heap

1. It is basically an array representation of complete binary tree.
2. Best way to implement Priority Queue.
3. Two types of Heaps – MIN Heap and MAX Heap.
4. No need to do balancing as it will always be balanced and data will be inserted in level order wise from left to right.
5. Applications – Dijkstra and Prims algo
6. In Max heap parent node is greater than child’s.



1. Whenever it is ask to find kth maximum or minimum heap can be used. Quick select algorithm can be also be used in such case. Heap take nlogk time with extra o(k) space. Where quick select take avg o(n) and worst case o(n2). So, It is personal choice which one to use. If size is fixed and k is large quick select can be used else heap can be used.
2. For any node at index I child node will be present at index 2i+1 and 2i+2.
3. Insert – we just add element to next available position which take o(1) time and then check parent and swap if necessary. Swap will go till parent at max and hence this heapify operation will take o(logn) time.

Suppose the Heap is a Max-Heap as:

10

/ \

5 3

/ \

2 4

The new element to be inserted is 15.

**Process**:

**Step 1:** Insert the new element at the end.

10

/ \

5 3

/ \ /

2 4 15

**Step 2**: Heapify the new element following bottom-up approach.

-> 15 is more than its parent 3, swap them.

10

/ \

5 15

/ \ /

2 4 3

-> 15 is again more than its parent 10, swap them.

15

/ \

5 10

/ \ /

2 4 3

Therefore, the final heap after insertion is:

15

/ \

5 10

/ \ /

2 4 3

1. For remove. Just remove root and replace last element from array to root index and then heapify from top to bottom. - logn

Suppose the Heap is a Max-Heap as:

10

/ \

5 3

/ \

2 4

*The element to be deleted is root, i.e. 10.*

**Process**:

The last element is 4.

**Step 1:** Replace the last element with root, and delete it.

4

/ \

5 3

/

2

**Step 2**: Heapify root.

Final Heap:

5

/ \

4 3

/

2

## 1. Add and delete element from Max heap.

### Problem

Create Max Heap data structure with below methods –

1. Add – add element to the heap

2. poll – remove max element from heap and return element.

3. size – current size of heap.

### Reference

LEETCODE, GIT

### Approach

1. Add – put element in last index and keep updating parent to see heap property. As soon as parent>child we terminate condition.
2. Poll – remove 0 index element and copy last index to 0 index. Then heapify from 0 index to down.
3. Size – normal size variable.

### Solution

**public** **class** HeapMax {

**private** **final** **int**[] table;

**private** **int** DEFAULT\_CAPACITY = 10;

**private** **int** size;

**private** Predicate<**int**[]> isFullPredicate = t -> t.length == getSize();

**public** HeapMax(**int** capacity) {

table = **new** **int**[capacity];

}

**public** HeapMax() {

table = **new** **int**[DEFAULT\_CAPACITY];

}

**public** **int** getSize() {

**return** size;

}

**private** **void** checkRange() {

**if** (isFullPredicate.test(table)) {

**throw** **new** ArrayIndexOutOfBoundsException();

}

}

/\*\*

\* poll will remove max element from heap

\*/

**public** **int** poll() {

**if** (size == 0) {

**return** -1;

}

**int** res = table[0];

table[0] = table[--size];

// fix downwards

heapify(0);

**return** res;

}

**private** **void** heapify(**int** index) {

**if** (index >= size) {// check if index passed is out of range

**return**;

}

**int** l = 2 \* index + 1;

**int** r = 2 \* index + 2;

**if** (l >= size) {// check if left child is null

**return**;

}

**int** max = l;

**if** (r < size) {// check if right also present

max = table[l] > table[r] ? l : r;

}

**if** (table[max] > table[index]) {

swap(index, max);

heapify(max);

}

}

/\*\*

\* peek will get max element from heap

\*/

**public** **int** peek() {

**return** size == 0 ? -1 : table[0];

}

**public** **void** add(**int** element) {

checkRange();

table[size++] = element;

// fix upward

**int** i = size - 1;

**while** (i > 0) {

**int** p = (i - 1) / 2;

**if** (table[p] < table[i]) {

swap(i, p);

} **else** {

**break**;

}

i = p;

}

}

**private** **void** swap(**int** i, **int** p) {

**int** temp = table[p];

table[p] = table[i];

table[i] = temp;

}

@Override

**public** String toString() {

StringBuilder sb = **new** StringBuilder(getSize());

**for** (**int** i = 0; i < getSize(); i++) {

sb.append(table[i] + ",");

}

**return** sb.toString();

}

}

### Time and space complexity

Add and poll take o(logn) time.

## 2. Last Stone Weight

### Problem

We have a collection of stones, each stone has a positive integer weight.

Each turn, we choose the two **heaviest** stones and smash them together.  Suppose the stones have weights x and y with x <= y.  The result of this smash is:

* If x == y, both stones are totally destroyed;
* If x != y, the stone of weight x is totally destroyed, and the stone of weight y has new weight y-x.

At the end, there is at most 1 stone left.  Return the weight of this stone (or 0 if there are no stones left.)

**Example 1:**

**Input:** [2,7,4,1,8,1]

**Output:** 1

**Explanation:**

We combine 7 and 8 to get 1 so the array converts to [2,4,1,1,1] then,

we combine 2 and 4 to get 2 so the array converts to [2,1,1,1] then,

we combine 2 and 1 to get 1 so the array converts to [1,1,1] then,

we combine 1 and 1 to get 0 so the array converts to [1] then that's the value of last stone.

**Note:**

1. 1 <= stones.length <= 30
2. 1 <= stones[i] <= 1000

### Reference

LEETCODE, GIT

### Approach

1. We will use max heap and poll first two elements and subtract them and add result again in heap.
2. Will do above operation till 1 element left.

### Solution

public int lastStoneWeight(int[] stones) {

HeapMax heapMax = new HeapMax(stones.length);

for(int i:stones){

heapMax.add(i);

}

while(heapMax.size()>1){

int x=heapMax.poll();

int y=heapMax.poll();

heapMax.add(x-y);

}

return heapMax.poll();

}

### Time and space complexity

Add and poll take o(logn) time.