Contents

[1. Array 27](#_Toc50131111)

[1. Find duplicate in the array 27](#_Toc50131112)

[Problem 27](#_Toc50131113)

[Reference 27](#_Toc50131114)

[Approach 27](#_Toc50131115)

[Solution 28](#_Toc50131116)

[Time and space complexity 28](#_Toc50131117)

[2. Find Single element in the array 28](#_Toc50131118)

[Problem 28](#_Toc50131119)

[Reference 28](#_Toc50131120)

[Approach 28](#_Toc50131121)

[Solution 29](#_Toc50131122)

[Time and space complexity 29](#_Toc50131123)

[3. Find Numbers with Even Number of Digits 29](#_Toc50131124)

[Problem 29](#_Toc50131125)

[Reference 29](#_Toc50131126)

[Approach 29](#_Toc50131127)

[Solution 29](#_Toc50131128)

[Time and space complexity 30](#_Toc50131129)

[4. Largest Number 30](#_Toc50131130)

[Problem 30](#_Toc50131131)

[Reference 30](#_Toc50131132)

[Approach 30](#_Toc50131133)

[Solution 30](#_Toc50131134)

[Time and space complexity 31](#_Toc50131135)

[5. Search Insert Position (Binary Search) 31](#_Toc50131136)

[Problem 31](#_Toc50131137)

[Reference 31](#_Toc50131138)

[Approach 31](#_Toc50131139)

[Solution 32](#_Toc50131140)

[Time and space complexity 32](#_Toc50131141)

[6. Count Negative Numbers in a Sorted Matrix 32](#_Toc50131142)

[Problem 32](#_Toc50131143)

[Reference 33](#_Toc50131144)

[Approach 33](#_Toc50131145)

[Solution 33](#_Toc50131146)

[Time and space complexity 34](#_Toc50131147)

[7. Move Zeroes to end of array 34](#_Toc50131148)

[Problem 34](#_Toc50131149)

[Reference 34](#_Toc50131150)

[Approach 34](#_Toc50131151)

[Solution 34](#_Toc50131152)

[Time and space complexity 35](#_Toc50131153)

[8. Two Sum 35](#_Toc50131154)

[Problem 35](#_Toc50131155)

[Reference 35](#_Toc50131156)

[Approach 35](#_Toc50131157)

[Solution 35](#_Toc50131158)

[Time and space complexity 35](#_Toc50131159)

[9. Merge two sorted array 35](#_Toc50131160)

[Problem 35](#_Toc50131161)

[Reference 36](#_Toc50131162)

[Approach 36](#_Toc50131163)

[Solution 36](#_Toc50131164)

[Time and space complexity 36](#_Toc50131165)

[10. Best Time to Buy & Sell Stock 37](#_Toc50131166)

[Problem 37](#_Toc50131167)

[Reference 37](#_Toc50131168)

[Approach 37](#_Toc50131169)

[Solution 37](#_Toc50131170)

[Time and space complexity 38](#_Toc50131171)

[11. Best Time to Buy & Sell Stock II 38](#_Toc50131172)

[Problem 38](#_Toc50131173)

[Reference 38](#_Toc50131174)

[Approach 38](#_Toc50131175)

[Solution 39](#_Toc50131176)

[Time and space complexity 39](#_Toc50131177)

[12. Length of longest contiguous subarray with equal number of 0 and 1. 39](#_Toc50131178)

[Problem 39](#_Toc50131179)

[Reference 39](#_Toc50131180)

[Approach 39](#_Toc50131181)

[Solution 39](#_Toc50131182)

[Time and space complexity 40](#_Toc50131183)

[13. Product of Array Except Self. 40](#_Toc50131184)

[Problem 40](#_Toc50131185)

[Reference 40](#_Toc50131186)

[Approach 40](#_Toc50131187)

[Solution 40](#_Toc50131188)

[Time and space complexity 41](#_Toc50131189)

[14. Minimum Path sum in 2D-Array. 41](#_Toc50131190)

[Problem 41](#_Toc50131191)

[Reference 41](#_Toc50131192)

[Approach 41](#_Toc50131193)

[Solution 41](#_Toc50131194)

[Time and space complexity 42](#_Toc50131195)

[15. Number of Islands in 2-D Array. 42](#_Toc50131196)

[Problem 42](#_Toc50131197)

[Reference 42](#_Toc50131198)

[Approach 42](#_Toc50131199)

[Solution 42](#_Toc50131200)

[Time and space complexity 43](#_Toc50131201)

[16. Find First occurrence of 1 in sorted binary array 43](#_Toc50131202)

[Problem 43](#_Toc50131203)

[Reference 43](#_Toc50131204)

[Approach 43](#_Toc50131205)

[Solution 43](#_Toc50131206)

[Time and space complexity 44](#_Toc50131207)

[17. Subarray Sum Equals K 44](#_Toc50131208)

[Problem 44](#_Toc50131209)

[Reference 44](#_Toc50131210)

[Approach 44](#_Toc50131211)

[Solution 45](#_Toc50131212)

[Time and space complexity 45](#_Toc50131213)

[18. Jump Game 45](#_Toc50131214)

[Problem 45](#_Toc50131215)

[Reference 45](#_Toc50131216)

[Approach 46](#_Toc50131217)

[Solution 46](#_Toc50131218)

[Time and space complexity 46](#_Toc50131219)

[19. First Bad Version 46](#_Toc50131220)

[Problem 46](#_Toc50131221)

[Reference 47](#_Toc50131222)

[Approach 47](#_Toc50131223)

[Solution 47](#_Toc50131224)

[Time and space complexity 47](#_Toc50131225)

[20. Squares of a Sorted Array 47](#_Toc50131226)

[Problem 47](#_Toc50131227)

[Reference 48](#_Toc50131228)

[Approach 48](#_Toc50131229)

[Solution 48](#_Toc50131230)

[Time and space complexity 48](#_Toc50131231)

[21. Majority Element 48](#_Toc50131232)

[Problem 48](#_Toc50131233)

[Reference 48](#_Toc50131234)

[Approach 49](#_Toc50131235)

[Solution 49](#_Toc50131236)

[Time and space complexity 49](#_Toc50131237)

[22. Find the Town Judge 49](#_Toc50131238)

[Problem 49](#_Toc50131239)

[Reference 50](#_Toc50131240)

[Approach 50](#_Toc50131241)

[Solution 50](#_Toc50131242)

[Time and space complexity 50](#_Toc50131243)

[23. Flood Fill 51](#_Toc50131244)

[Problem 51](#_Toc50131245)

[Reference 51](#_Toc50131246)

[Approach 51](#_Toc50131247)

[Solution 51](#_Toc50131248)

[Time and space complexity 52](#_Toc50131249)

[24. Single Element in a Sorted Array 52](#_Toc50131250)

[Problem 52](#_Toc50131251)

[Reference 52](#_Toc50131252)

[Approach 52](#_Toc50131253)

[Solution 52](#_Toc50131254)

[Time and space complexity 53](#_Toc50131255)

[25. Remove K Digits 53](#_Toc50131256)

[Problem 53](#_Toc50131257)

[Reference 54](#_Toc50131258)

[Approach 54](#_Toc50131259)

[Solution 54](#_Toc50131260)

[Time and space complexity 55](#_Toc50131261)

[26. Sort Characters By Frequency 55](#_Toc50131262)

[Problem 55](#_Toc50131263)

[Reference 56](#_Toc50131264)

[Approach 56](#_Toc50131265)

[Solution 56](#_Toc50131266)

[Time and space complexity 56](#_Toc50131267)

[27. Interval List Intersections 56](#_Toc50131268)

[Problem 56](#_Toc50131269)

[Reference 57](#_Toc50131270)

[Approach 57](#_Toc50131271)

[Solution 57](#_Toc50131272)

[Time and space complexity 57](#_Toc50131273)

[28. Search a 2D Matrix (data is sorted in matrix) 58](#_Toc50131274)

[Problem 58](#_Toc50131275)

[Reference 58](#_Toc50131276)

[Approach 58](#_Toc50131277)

[Solution 59](#_Toc50131278)

[Time and space complexity 59](#_Toc50131279)

[29. Two City Scheduling 59](#_Toc50131280)

[Problem 59](#_Toc50131281)

[Reference 60](#_Toc50131282)

[Approach 60](#_Toc50131283)

[Solution 60](#_Toc50131284)

[Time and space complexity 60](#_Toc50131285)

[30. Random Pick with Weight 60](#_Toc50131286)

[Problem 60](#_Toc50131287)

[Reference 61](#_Toc50131288)

[Approach 61](#_Toc50131289)

[Solution 61](#_Toc50131290)

[Time and space complexity 62](#_Toc50131291)

[32. Remove Duplicates from Sorted Array 62](#_Toc50131292)

[Problem 62](#_Toc50131293)

[Reference 63](#_Toc50131294)

[Approach 63](#_Toc50131295)

[Solution 63](#_Toc50131296)

[Time and space complexity 63](#_Toc50131297)

[33 Duplicate Zeros 63](#_Toc50131298)

[Problem 63](#_Toc50131299)

[Reference 64](#_Toc50131300)

[Approach 64](#_Toc50131301)

[Solution 66](#_Toc50131302)

[Time and space complexity 67](#_Toc50131303)

[34 H-Index II 67](#_Toc50131304)

[Problem 67](#_Toc50131305)

[Reference 67](#_Toc50131306)

[Approach 67](#_Toc50131307)

[Solution 68](#_Toc50131308)

[Time and space complexity 68](#_Toc50131309)

[35 Create Target Array 68](#_Toc50131310)

[Problem 68](#_Toc50131311)

[Reference 69](#_Toc50131312)

[Approach 69](#_Toc50131313)

[Solution 69](#_Toc50131314)

[Time and space complexity 70](#_Toc50131315)

[36 Split a String in Balanced Strings 70](#_Toc50131316)

[Problem 70](#_Toc50131317)

[Reference 71](#_Toc50131318)

[Approach 71](#_Toc50131319)

[Solution 71](#_Toc50131320)

[Time and space complexity 71](#_Toc50131321)

[37 Cells with Odd Values in a Matrix 71](#_Toc50131322)

[Problem 71](#_Toc50131323)

[Reference 72](#_Toc50131324)

[Approach 72](#_Toc50131325)

[Solution 73](#_Toc50131326)

[Time and space complexity 73](#_Toc50131327)

[38 Print all permutations of array 73](#_Toc50131328)

[Problem 73](#_Toc50131329)

[Reference 73](#_Toc50131330)

[Approach 73](#_Toc50131331)

[Solution 74](#_Toc50131332)

[Time and space complexity 74](#_Toc50131333)

[39 Print all unique permutations of array 74](#_Toc50131334)

[Problem 74](#_Toc50131335)

[Reference 75](#_Toc50131336)

[Approach 75](#_Toc50131337)

[Solution 75](#_Toc50131338)

[Time and space complexity 75](#_Toc50131339)

[40 Contains duplicate II 75](#_Toc50131340)

[Problem 75](#_Toc50131341)

[Reference 76](#_Toc50131342)

[Approach 76](#_Toc50131343)

[Solution 76](#_Toc50131344)

[Time and space complexity 77](#_Toc50131345)

[41 Contains duplicate III 77](#_Toc50131346)

[Problem 77](#_Toc50131347)

[Reference 77](#_Toc50131348)

[Approach 77](#_Toc50131349)

[Solution 77](#_Toc50131350)

[Time and space complexity 78](#_Toc50131351)

[42 Next Permutation (Next greater element) 78](#_Toc50131352)

[Problem 78](#_Toc50131353)

[Reference 78](#_Toc50131354)

[Approach 78](#_Toc50131355)

[Solution 79](#_Toc50131356)

[Time and space complexity 79](#_Toc50131357)

[2. String 79](#_Toc50131358)

[1. Check if a string is substring of source. (Rabin Karp Algorithm) 79](#_Toc50131359)

[Problem 79](#_Toc50131360)

[Reference 79](#_Toc50131361)

[Approach 80](#_Toc50131362)

[Solution 80](#_Toc50131363)

[Time and space complexity 81](#_Toc50131364)

[2. Anagram 81](#_Toc50131365)

[Problem 81](#_Toc50131366)

[Reference 81](#_Toc50131367)

[Approach 81](#_Toc50131368)

[Solution 81](#_Toc50131369)

[Time and space complexity 82](#_Toc50131370)

[3. Group Anagrams 82](#_Toc50131371)

[Problem 82](#_Toc50131372)

[Reference 82](#_Toc50131373)

[Approach 82](#_Toc50131374)

[Solution 83](#_Toc50131375)

[Time and space complexity 83](#_Toc50131376)

[4. Backspace String Compare 84](#_Toc50131377)

[Problem 84](#_Toc50131378)

[Reference 84](#_Toc50131379)

[Approach 84](#_Toc50131380)

[Solution 84](#_Toc50131381)

[Time and space complexity 85](#_Toc50131382)

[5. Perform String Shifts 85](#_Toc50131383)

[Problem 85](#_Toc50131384)

[Reference 86](#_Toc50131385)

[Approach 86](#_Toc50131386)

[Solution 86](#_Toc50131387)

[Time and space complexity 86](#_Toc50131388)

[6. Check Valid Parenthesis String 87](#_Toc50131389)

[Problem 87](#_Toc50131390)

[Reference 87](#_Toc50131391)

[Approach 87](#_Toc50131392)

[Solution 87](#_Toc50131393)

[Time and space complexity 88](#_Toc50131394)

[7. Permutation in String 88](#_Toc50131395)

[Problem 88](#_Toc50131396)

[Reference 88](#_Toc50131397)

[Approach 89](#_Toc50131398)

[Solution 89](#_Toc50131399)

[Time and space complexity 89](#_Toc50131400)

[8. Validate IP Address 89](#_Toc50131401)

[Problem 89](#_Toc50131402)

[Reference 90](#_Toc50131403)

[Approach 90](#_Toc50131404)

[Solution 91](#_Toc50131405)

[Time and space complexity 91](#_Toc50131406)

[9. Longest Duplicate Substring 92](#_Toc50131407)

[Problem 92](#_Toc50131408)

[Reference 92](#_Toc50131409)

[Approach 92](#_Toc50131410)

[Solution 93](#_Toc50131411)

[Time and space complexity 93](#_Toc50131412)

[10. Print all permutations of a String 94](#_Toc50131413)

[Problem 94](#_Toc50131414)

[Reference 94](#_Toc50131415)

[Approach 94](#_Toc50131416)

[Solution 94](#_Toc50131417)

[Time and space complexity 94](#_Toc50131418)

[11. Repeated Substring Pattern 94](#_Toc50131419)

[Problem 94](#_Toc50131420)

[Reference 95](#_Toc50131421)

[Approach 95](#_Toc50131422)

[Solution 96](#_Toc50131423)

[Time and space complexity 96](#_Toc50131424)

[12. Partition Labels 96](#_Toc50131425)

[Problem 96](#_Toc50131426)

[Reference 97](#_Toc50131427)

[Approach 97](#_Toc50131428)

[Solution 97](#_Toc50131429)

[Time and space complexity 97](#_Toc50131430)

[3. Math 97](#_Toc50131431)

[1. Palindrome Number 97](#_Toc50131432)

[Problem 97](#_Toc50131433)

[Reference 98](#_Toc50131434)

[Approach 98](#_Toc50131435)

[Solution 98](#_Toc50131436)

[Time and space complexity 98](#_Toc50131437)

[2. Happy Number 98](#_Toc50131438)

[Problem 98](#_Toc50131439)

[Reference 99](#_Toc50131440)

[Approach 99](#_Toc50131441)

[Solution 99](#_Toc50131442)

[Time and space complexity 99](#_Toc50131443)

[2. Check If It Is a Straight Line 99](#_Toc50131444)

[Problem 99](#_Toc50131445)

[Reference 100](#_Toc50131446)

[Approach 101](#_Toc50131447)

[Solution 101](#_Toc50131448)

[Time and space complexity 101](#_Toc50131449)

[3. Check If It Is a Valid Perfect Square 101](#_Toc50131450)

[Problem 101](#_Toc50131451)

[Reference 101](#_Toc50131452)

[Approach 101](#_Toc50131453)

[Solution 102](#_Toc50131454)

[Time and space complexity 102](#_Toc50131455)

[4. Linked List 102](#_Toc50131456)

[1. Reverse single linked list (Iterative + recursive) 102](#_Toc50131457)

[Problem 102](#_Toc50131458)

[Reference 102](#_Toc50131459)

[Approach 102](#_Toc50131460)

[Solution 102](#_Toc50131461)

[Time and space complexity 103](#_Toc50131462)

[2. Reverse single linked list recursive 103](#_Toc50131463)

[Problem 103](#_Toc50131464)

[Reference 103](#_Toc50131465)

[Approach 103](#_Toc50131466)

[Solution 103](#_Toc50131467)

[Time and space complexity 104](#_Toc50131468)

[3. Check Palindrome single linked list 104](#_Toc50131469)

[Problem 104](#_Toc50131470)

[Reference 104](#_Toc50131471)

[Approach 104](#_Toc50131472)

[Solution 104](#_Toc50131473)

[Time and space complexity 106](#_Toc50131474)

[4. Find middle element of single linked list 106](#_Toc50131475)

[Problem 106](#_Toc50131476)

[Reference 106](#_Toc50131477)

[Approach 106](#_Toc50131478)

[Solution 107](#_Toc50131479)

[Time and space complexity 107](#_Toc50131480)

[5. Detect cycle in linked list 107](#_Toc50131481)

[Problem 107](#_Toc50131482)

[Reference 108](#_Toc50131483)

[Approach 108](#_Toc50131484)

[Solution 108](#_Toc50131485)

[Time and space complexity 108](#_Toc50131486)

[6. Detect cycle in linked list and return starting node of loop 108](#_Toc50131487)

[Problem 108](#_Toc50131488)

[Reference 108](#_Toc50131489)

[Approach 108](#_Toc50131490)

[Solution 109](#_Toc50131491)

[Time and space complexity 109](#_Toc50131492)

[7. Intersection of Two Linked Lists 109](#_Toc50131493)

[Problem 109](#_Toc50131494)

[Reference 109](#_Toc50131495)

[Approach 110](#_Toc50131496)

[Solution 110](#_Toc50131497)

[Time and space complexity 111](#_Toc50131498)

[8. Delete Linked List Elements 111](#_Toc50131499)

[Problem 111](#_Toc50131500)

[Reference 111](#_Toc50131501)

[Approach 111](#_Toc50131502)

[Solution 111](#_Toc50131503)

[Time and space complexity 112](#_Toc50131504)

[9. Separate odd even nodes 112](#_Toc50131505)

[Problem 112](#_Toc50131506)

[Reference 112](#_Toc50131507)

[Approach 112](#_Toc50131508)

[Solution 112](#_Toc50131509)

[Time and space complexity 113](#_Toc50131510)

[10. Remove Nth Node From End of List 113](#_Toc50131511)

[Problem 113](#_Toc50131512)

[Reference 114](#_Toc50131513)

[Approach 114](#_Toc50131514)

[Solution 114](#_Toc50131515)

[Time and space complexity 114](#_Toc50131516)

[11. Rotate List 114](#_Toc50131517)

[Problem 114](#_Toc50131518)

[Reference 115](#_Toc50131519)

[Approach 115](#_Toc50131520)

[Solution 115](#_Toc50131521)

[Time and space complexity 116](#_Toc50131522)

[12. Swap Nodes in Pairs Recursive 116](#_Toc50131523)

[Problem 116](#_Toc50131524)

[Reference 116](#_Toc50131525)

[Approach 116](#_Toc50131526)

[Solution 116](#_Toc50131527)

[Time and space complexity 117](#_Toc50131528)

[13. Delete node from linked list (without head given) 117](#_Toc50131529)

[Problem 117](#_Toc50131530)

[Reference 117](#_Toc50131531)

[Approach 117](#_Toc50131532)

[Solution 117](#_Toc50131533)

[Time and space complexity 118](#_Toc50131534)

[5. Stack 118](#_Toc50131535)

[1. Next greatest element in array 118](#_Toc50131536)

[Problem 118](#_Toc50131537)

[Reference 118](#_Toc50131538)

[Approach 118](#_Toc50131539)

[Solution 119](#_Toc50131540)

[Time and space complexity 119](#_Toc50131541)

[2. Online Stock Span 120](#_Toc50131542)

[Problem 120](#_Toc50131543)

[Reference 120](#_Toc50131544)

[Approach 120](#_Toc50131545)

[Solution 120](#_Toc50131546)

[Time and space complexity 121](#_Toc50131547)

[3. Valid Parentheses 121](#_Toc50131548)

[Problem 121](#_Toc50131549)

[Reference 121](#_Toc50131550)

[Approach 121](#_Toc50131551)

[Solution 122](#_Toc50131552)

[Time and space complexity 122](#_Toc50131553)

[4. Implement Queue using Stack 122](#_Toc50131554)

[Problem 122](#_Toc50131555)

[Reference 122](#_Toc50131556)

[Approach 122](#_Toc50131557)

[Solution 123](#_Toc50131558)

[Time and space complexity 124](#_Toc50131559)

[5. Current Maximum element in stack 124](#_Toc50131560)

[Problem 124](#_Toc50131561)

[Reference 124](#_Toc50131562)

[Approach 124](#_Toc50131563)

[Solution 125](#_Toc50131564)

[Time and space complexity 127](#_Toc50131565)

[6. Queue 127](#_Toc50131566)

[1. Implement Stack using queue 127](#_Toc50131567)

[Problem 127](#_Toc50131568)

[Reference 127](#_Toc50131569)

[Approach 127](#_Toc50131570)

[Solution 128](#_Toc50131571)

[Time and space complexity 128](#_Toc50131572)

[7. Sorting 128](#_Toc50131573)

[1. Insertion Sort in array 129](#_Toc50131574)

[Problem 129](#_Toc50131575)

[Reference 129](#_Toc50131576)

[Approach 129](#_Toc50131577)

[Solution 129](#_Toc50131578)

[Time and space complexity 130](#_Toc50131579)

[2. Insertion Sort in Single Linked list 130](#_Toc50131580)

[Problem 130](#_Toc50131581)

[Reference 130](#_Toc50131582)

[Approach 130](#_Toc50131583)

[Solution 130](#_Toc50131584)

[Time and space complexity 131](#_Toc50131585)

[3. Selection Sort in array 131](#_Toc50131586)

[Problem 131](#_Toc50131587)

[Reference 131](#_Toc50131588)

[Approach 131](#_Toc50131589)

[Solution 131](#_Toc50131590)

[Time and space complexity 132](#_Toc50131591)

[4. Selection Sort in Single linked list 132](#_Toc50131592)

[Problem 132](#_Toc50131593)

[Reference 132](#_Toc50131594)

[Approach 132](#_Toc50131595)

[Solution 132](#_Toc50131596)

[Time and space complexity 133](#_Toc50131597)

[5. Bubble Sort in array 133](#_Toc50131598)

[Problem 133](#_Toc50131599)

[Reference 133](#_Toc50131600)

[Approach 133](#_Toc50131601)

[Solution 133](#_Toc50131602)

[Time and space complexity 133](#_Toc50131603)

[6. Bubble Sort in Single Linked list 134](#_Toc50131604)

[Problem 134](#_Toc50131605)

[Reference 134](#_Toc50131606)

[Approach 134](#_Toc50131607)

[Solution 134](#_Toc50131608)

[Time and space complexity 134](#_Toc50131609)

[7. Merge Sort in array 135](#_Toc50131610)

[Problem 135](#_Toc50131611)

[Reference 135](#_Toc50131612)

[Approach 135](#_Toc50131613)

[Solution 135](#_Toc50131614)

[Time and space complexity 137](#_Toc50131615)

[8. Merge Sort in linked list 137](#_Toc50131616)

[Problem 137](#_Toc50131617)

[Reference 137](#_Toc50131618)

[Approach 137](#_Toc50131619)

[Solution 137](#_Toc50131620)

[Time and space complexity 139](#_Toc50131621)

[9. Quick Sort in array 139](#_Toc50131622)

[Problem 139](#_Toc50131623)

[Reference 139](#_Toc50131624)

[Approach 139](#_Toc50131625)

[Solution 140](#_Toc50131626)

[Time and space complexity 141](#_Toc50131627)

[10. Counting sort (Sort Colors) 141](#_Toc50131628)

[Problem 141](#_Toc50131629)

[Reference 141](#_Toc50131630)

[Approach 142](#_Toc50131631)

[Solution 142](#_Toc50131632)

[Time and space complexity 142](#_Toc50131633)

[8. Selection problem 142](#_Toc50131634)

[1. Find kth smallest element in array (Quick Select) 142](#_Toc50131635)

[Problem 142](#_Toc50131636)

[Reference 143](#_Toc50131637)

[Approach 143](#_Toc50131638)

[Solution 143](#_Toc50131639)

[Time and space complexity 144](#_Toc50131640)

[9 Bitwise operator 144](#_Toc50131641)

[1. Convert Binary Number in a Linked List to Integer 146](#_Toc50131642)

[Problem 146](#_Toc50131643)

[Reference 146](#_Toc50131644)

[Approach 146](#_Toc50131645)

[Solution 146](#_Toc50131646)

[Time and space complexity 146](#_Toc50131647)

[2. Bitwise AND of Numbers Range 147](#_Toc50131648)

[Problem 147](#_Toc50131649)

[Reference 147](#_Toc50131650)

[Approach 147](#_Toc50131651)

[Solution 147](#_Toc50131652)

[Time and space complexity 148](#_Toc50131653)

[3. Number Complement 148](#_Toc50131654)

[Problem 148](#_Toc50131655)

[Reference 148](#_Toc50131656)

[Approach 148](#_Toc50131657)

[Solution 149](#_Toc50131658)

[Time and space complexity 149](#_Toc50131659)

[4. Counting Bits (number of 1’s) 149](#_Toc50131660)

[Problem 149](#_Toc50131661)

[Reference 149](#_Toc50131662)

[Approach 149](#_Toc50131663)

[Solution 149](#_Toc50131664)

[Time and space complexity 149](#_Toc50131665)

[5. Power of Two 150](#_Toc50131666)

[Problem 150](#_Toc50131667)

[Reference 150](#_Toc50131668)

[Approach 150](#_Toc50131669)

[Solution 150](#_Toc50131670)

[Time and space complexity 150](#_Toc50131671)

[6. Power of Four 150](#_Toc50131672)

[Problem 150](#_Toc50131673)

[Reference 151](#_Toc50131674)

[Approach 151](#_Toc50131675)

[Solution 151](#_Toc50131676)

[Time and space complexity 151](#_Toc50131677)

[7. Power of Three 151](#_Toc50131678)

[Problem 151](#_Toc50131679)

[Reference 152](#_Toc50131680)

[Approach 152](#_Toc50131681)

[Solution 152](#_Toc50131682)

[Time and space complexity 152](#_Toc50131683)

[8. Single Number II 152](#_Toc50131684)

[Problem 152](#_Toc50131685)

[Reference 153](#_Toc50131686)

[Approach 153](#_Toc50131687)

[Solution 153](#_Toc50131688)

[Time and space complexity 154](#_Toc50131689)

[9. Single Number III 154](#_Toc50131690)

[Problem 154](#_Toc50131691)

[Reference 155](#_Toc50131692)

[Approach 155](#_Toc50131693)

[Solution 155](#_Toc50131694)

[Time and space complexity 155](#_Toc50131695)

[10. Binary Tree 155](#_Toc50131696)

[1. Pre-Order Traversal Recursive 157](#_Toc50131697)

[Problem 157](#_Toc50131698)

[Reference 157](#_Toc50131699)

[Approach 157](#_Toc50131700)

[Solution 157](#_Toc50131701)

[Time and space complexity 158](#_Toc50131702)

[2. Pre-Order Traversal Iterative 158](#_Toc50131703)

[Problem 158](#_Toc50131704)

[Reference 158](#_Toc50131705)

[Approach 158](#_Toc50131706)

[Solution 158](#_Toc50131707)

[Time and space complexity 158](#_Toc50131708)

[3. In-Order Traversal Recursive 158](#_Toc50131709)

[Problem 158](#_Toc50131710)

[Reference 158](#_Toc50131711)

[Approach 158](#_Toc50131712)

[Solution 159](#_Toc50131713)

[Time and space complexity 159](#_Toc50131714)

[4. In-Order Traversal Iterative 159](#_Toc50131715)

[Problem 159](#_Toc50131716)

[Reference 159](#_Toc50131717)

[Approach 159](#_Toc50131718)

[Solution 159](#_Toc50131719)

[Time and space complexity 160](#_Toc50131720)

[5. Post-Order Traversal Recursive 160](#_Toc50131721)

[Problem 160](#_Toc50131722)

[Reference 160](#_Toc50131723)

[Approach 160](#_Toc50131724)

[Solution 160](#_Toc50131725)

[Time and space complexity 160](#_Toc50131726)

[6. Post-Order Traversal Iterative 161](#_Toc50131727)

[Problem 161](#_Toc50131728)

[Reference 161](#_Toc50131729)

[Approach 161](#_Toc50131730)

[Solution 161](#_Toc50131731)

[Time and space complexity 161](#_Toc50131732)

[10. Level-Order Traversal 161](#_Toc50131733)

[Problem 161](#_Toc50131734)

[Reference 161](#_Toc50131735)

[Approach 161](#_Toc50131736)

[Solution 161](#_Toc50131737)

[Time and space complexity 163](#_Toc50131738)

[8. Zig-Zag Order Traversal Iterative 163](#_Toc50131739)

[Problem 163](#_Toc50131740)

[Reference 163](#_Toc50131741)

[Approach 163](#_Toc50131742)

[Solution 163](#_Toc50131743)

[Time and space complexity 164](#_Toc50131744)

[9. Binary Tree Tilt 164](#_Toc50131745)

[Problem 164](#_Toc50131746)

[Reference 165](#_Toc50131747)

[Approach 165](#_Toc50131748)

[Solution 165](#_Toc50131749)

[Time and space complexity 165](#_Toc50131750)

[10. Path Sum 165](#_Toc50131751)

[Problem 165](#_Toc50131752)

[Reference 166](#_Toc50131753)

[Approach 166](#_Toc50131754)

[Solution 166](#_Toc50131755)

[Time and space complexity 166](#_Toc50131756)

[11. Number of nodes in binary tree 166](#_Toc50131757)

[Problem 166](#_Toc50131758)

[Reference 166](#_Toc50131759)

[Approach 167](#_Toc50131760)

[Solution 167](#_Toc50131761)

[Time and space complexity 167](#_Toc50131762)

[12. Number of leaves in binary tree 167](#_Toc50131763)

[Problem 167](#_Toc50131764)

[Reference 167](#_Toc50131765)

[Approach 167](#_Toc50131766)

[Solution 167](#_Toc50131767)

[Time and space complexity 168](#_Toc50131768)

[13. Number of Full nodes in binary tree 168](#_Toc50131769)

[Problem 168](#_Toc50131770)

[Reference 168](#_Toc50131771)

[Approach 168](#_Toc50131772)

[Solution 168](#_Toc50131773)

[Time and space complexity 168](#_Toc50131774)

[14. Maximum Depth/height in binary tree 168](#_Toc50131775)

[Problem 168](#_Toc50131776)

[Reference 168](#_Toc50131777)

[Approach 168](#_Toc50131778)

[Solution 169](#_Toc50131779)

[Time and space complexity 170](#_Toc50131780)

[15. Binary Tree Paths (root to leaf path) 170](#_Toc50131781)

[Problem 170](#_Toc50131782)

[Reference 170](#_Toc50131783)

[Approach 170](#_Toc50131784)

[Solution 170](#_Toc50131785)

[Time and space complexity 171](#_Toc50131786)

[16. Minimum Depth in binary tree iteratively 171](#_Toc50131787)

[Problem 171](#_Toc50131788)

[Reference 171](#_Toc50131789)

[Approach 171](#_Toc50131790)

[Solution 171](#_Toc50131791)

[Time and space complexity 172](#_Toc50131792)

[17. Maximum element in binary tree 172](#_Toc50131793)

[Problem 172](#_Toc50131794)

[Reference 172](#_Toc50131795)

[Approach 172](#_Toc50131796)

[Solution 173](#_Toc50131797)

[Time and space complexity 173](#_Toc50131798)

[18. Insert item in a binary tree 173](#_Toc50131799)

[Problem 173](#_Toc50131800)

[Reference 173](#_Toc50131801)

[Approach 173](#_Toc50131802)

[Solution 173](#_Toc50131803)

[Time and space complexity 174](#_Toc50131804)

[19. Level-Order Traversal Reverse Order 174](#_Toc50131805)

[Problem 174](#_Toc50131806)

[Reference 174](#_Toc50131807)

[Approach 174](#_Toc50131808)

[Solution 174](#_Toc50131809)

[Time and space complexity 174](#_Toc50131810)

[20. Diameter of Binary tree 174](#_Toc50131811)

[Problem 174](#_Toc50131812)

[Reference 175](#_Toc50131813)

[Approach 175](#_Toc50131814)

[Solution 175](#_Toc50131815)

[Time and space complexity 176](#_Toc50131816)

[21. Construct Binary Tree from given post-order and in-order 176](#_Toc50131817)

[Problem 176](#_Toc50131818)

[Reference 177](#_Toc50131819)

[Approach 177](#_Toc50131820)

[Solution 177](#_Toc50131821)

[Time and space complexity 178](#_Toc50131822)

[22. Identical Structure of Two Tree (ISOMORPHIC) 178](#_Toc50131823)

[Problem 178](#_Toc50131824)

[Reference 178](#_Toc50131825)

[Approach 178](#_Toc50131826)

[Solution 179](#_Toc50131827)

[Time and space complexity 179](#_Toc50131828)

[23. Symmetric Tree (Mirror) 179](#_Toc50131829)

[Problem 179](#_Toc50131830)

[Reference 179](#_Toc50131831)

[Approach 180](#_Toc50131832)

[Solution 180](#_Toc50131833)

[Time and space complexity 181](#_Toc50131834)

[24. Populating Next Right Pointers in Each Node(Next Sibling) 181](#_Toc50131835)

[Problem 181](#_Toc50131836)

[Reference 182](#_Toc50131837)

[Approach 182](#_Toc50131838)

[Solution 182](#_Toc50131839)

[Time and space complexity 184](#_Toc50131840)

[25. Populating Next Right Pointers in Each Node II 184](#_Toc50131841)

[Problem 184](#_Toc50131842)

[Reference 184](#_Toc50131843)

[Approach 184](#_Toc50131844)

[Solution 185](#_Toc50131845)

[Time and space complexity 185](#_Toc50131846)

[26. Lowest Common Ancestor 186](#_Toc50131847)

[Problem 186](#_Toc50131848)

[Reference 186](#_Toc50131849)

[Approach 186](#_Toc50131850)

[Solution 187](#_Toc50131851)

[Time and space complexity 187](#_Toc50131852)

[27. Invert Binary Tree 187](#_Toc50131853)

[Problem 187](#_Toc50131854)

[Reference 187](#_Toc50131855)

[Approach 188](#_Toc50131856)

[Solution 188](#_Toc50131857)

[Time and space complexity 188](#_Toc50131858)

[28. Find Node in a binary tree 188](#_Toc50131859)

[Problem 188](#_Toc50131860)

[Reference 188](#_Toc50131861)

[Approach 188](#_Toc50131862)

[Solution 188](#_Toc50131863)

[Time and space complexity 189](#_Toc50131864)

[29. Find Maximum Node in a binary tree 189](#_Toc50131865)

[Problem 189](#_Toc50131866)

[Reference 189](#_Toc50131867)

[Approach 189](#_Toc50131868)

[Solution 189](#_Toc50131869)

[Time and space complexity 189](#_Toc50131870)

[30. Find Maximum Width of a binary tree 189](#_Toc50131871)

[Problem 189](#_Toc50131872)

[Reference 190](#_Toc50131873)

[Approach 190](#_Toc50131874)

[Solution 190](#_Toc50131875)

[Time and space complexity 190](#_Toc50131876)

[31. Count Complete Tree Nodes 190](#_Toc50131877)

[Problem 190](#_Toc50131878)

[Reference 191](#_Toc50131879)

[Approach 191](#_Toc50131880)

[Solution 191](#_Toc50131881)

[Time and space complexity 192](#_Toc50131882)

[11. Binary Search Tree 192](#_Toc50131883)

[1. Search in Binary Search Tree 192](#_Toc50131884)

[Problem 192](#_Toc50131885)

[Reference 193](#_Toc50131886)

[Approach 193](#_Toc50131887)

[Solution 193](#_Toc50131888)

[Time and space complexity 193](#_Toc50131889)

[2. Insert in Binary Search Tree 193](#_Toc50131890)

[Problem 193](#_Toc50131891)

[Reference 194](#_Toc50131892)

[Approach 194](#_Toc50131893)

[Solution 194](#_Toc50131894)

[Time and space complexity 194](#_Toc50131895)

[3. Construct Binary Search Tree from given pre-order and in-order 194](#_Toc50131896)

[Problem 194](#_Toc50131897)

[Reference 194](#_Toc50131898)

[Approach 194](#_Toc50131899)

[Solution 195](#_Toc50131900)

[Time and space complexity 196](#_Toc50131901)

[4. Construct Binary Search Tree from given pre-order 196](#_Toc50131902)

[Problem 196](#_Toc50131903)

[Reference 196](#_Toc50131904)

[Approach 196](#_Toc50131905)

[Solution 196](#_Toc50131906)

[Time and space complexity 197](#_Toc50131907)

[5. Construct Binary Search Tree from given post-order 197](#_Toc50131908)

[Problem 197](#_Toc50131909)

[Reference 197](#_Toc50131910)

[Approach 197](#_Toc50131911)

[Solution 197](#_Toc50131912)

[Time and space complexity 198](#_Toc50131913)

[6. Check if Binary tree is Binary Search Tree 198](#_Toc50131914)

[Problem 198](#_Toc50131915)

[Reference 199](#_Toc50131916)

[Approach 199](#_Toc50131917)

[Solution 199](#_Toc50131918)

[Time and space complexity 199](#_Toc50131919)

[7. Lowest Common Ancestor 199](#_Toc50131920)

[Problem 199](#_Toc50131921)

[Reference 200](#_Toc50131922)

[Approach 200](#_Toc50131923)

[Solution 200](#_Toc50131924)

[Time and space complexity 200](#_Toc50131925)

[8. Kth smallest element in binary search tree 201](#_Toc50131926)

[Problem 201](#_Toc50131927)

[Reference 201](#_Toc50131928)

[Approach 201](#_Toc50131929)

[Solution 201](#_Toc50131930)

[Time and space complexity 202](#_Toc50131931)

[9. Range Sum of BST 202](#_Toc50131932)

[Problem 202](#_Toc50131933)

[Reference 202](#_Toc50131934)

[Approach 202](#_Toc50131935)

[Solution 202](#_Toc50131936)

[Time and space complexity 203](#_Toc50131937)

[12. Dynamic Programming 203](#_Toc50131938)

[1. Maximum sum in Contiguous Sub-Array 203](#_Toc50131939)

[Problem 203](#_Toc50131940)

[Reference 203](#_Toc50131941)

[Approach 203](#_Toc50131942)

[Solution 203](#_Toc50131943)

[Time and space complexity 204](#_Toc50131944)

[2. Maximum Sum in Circular Contiguous Subarray 204](#_Toc50131945)

[Problem 204](#_Toc50131946)

[Reference 205](#_Toc50131947)

[Approach 205](#_Toc50131948)

[Solution 205](#_Toc50131949)

[Time and space complexity 205](#_Toc50131950)

[3. Count Square Submatrices with All Ones 205](#_Toc50131951)

[Problem 205](#_Toc50131952)

[Reference 206](#_Toc50131953)

[Approach 206](#_Toc50131954)

[Solution 206](#_Toc50131955)

[Time and space complexity 207](#_Toc50131956)

[4. Uncrossed Lines 207](#_Toc50131957)

[Problem 207](#_Toc50131958)

[Reference 208](#_Toc50131959)

[Approach 208](#_Toc50131960)

[Solution 208](#_Toc50131961)

[Time and space complexity 208](#_Toc50131962)

[5. Edit Distance 208](#_Toc50131963)

[Problem 208](#_Toc50131964)

[Reference 209](#_Toc50131965)

[Approach 209](#_Toc50131966)

[Solution 210](#_Toc50131967)

[Time and space complexity 210](#_Toc50131968)

[13. Design data structure 210](#_Toc50131969)

[1. Design LRU cache with put and get in o(1) time 210](#_Toc50131970)

[Problem 210](#_Toc50131971)

[Reference 211](#_Toc50131972)

[Approach 211](#_Toc50131973)

[Solution 211](#_Toc50131974)

[Time and space complexity 212](#_Toc50131975)

[2. Insert Delete GetRandom O(1) 212](#_Toc50131976)

[Problem 212](#_Toc50131977)

[Reference 213](#_Toc50131978)

[Approach 213](#_Toc50131979)

[Solution 213](#_Toc50131980)

[Time and space complexity 214](#_Toc50131981)

[14. Heap 214](#_Toc50131982)

[1. Add and delete element from Max heap. 216](#_Toc50131983)

[Problem 216](#_Toc50131984)

[Reference 216](#_Toc50131985)

[Approach 216](#_Toc50131986)

[Solution 216](#_Toc50131987)

[Time and space complexity 218](#_Toc50131988)

[2. Last Stone Weight 218](#_Toc50131989)

[Problem 218](#_Toc50131990)

[Reference 219](#_Toc50131991)

[Approach 219](#_Toc50131992)

[Solution 219](#_Toc50131993)

[Time and space complexity 219](#_Toc50131994)

[3. K Closest Points to Origin 219](#_Toc50131995)

[Problem 219](#_Toc50131996)

[Reference 220](#_Toc50131997)

[Approach 220](#_Toc50131998)

[Solution 220](#_Toc50131999)

[Time and space complexity 221](#_Toc50132000)

[15. Trie 221](#_Toc50132001)

[1. Implement Trie (Prefix Tree) 222](#_Toc50132002)

[Problem 222](#_Toc50132003)

[Reference 223](#_Toc50132004)

[Approach 223](#_Toc50132005)

[Solution 223](#_Toc50132006)

[Time and space complexity 224](#_Toc50132007)

[16. Graph 224](#_Toc50132008)

[1. Finding in and out degrees of all vertices in a graph 230](#_Toc50132009)

[Problem 230](#_Toc50132010)

[Reference 230](#_Toc50132011)

[Approach 230](#_Toc50132012)

[Solution 231](#_Toc50132013)

[Time and space complexity 231](#_Toc50132014)

[2. Topological order of graph (Topological Sorting) 231](#_Toc50132015)

[Problem 231](#_Toc50132016)

[Reference 232](#_Toc50132017)

[Approach 232](#_Toc50132018)

[Solution 233](#_Toc50132019)

[Time and space complexity 234](#_Toc50132020)

[3. Course Schedule (detect cycle in DAG) 234](#_Toc50132021)

[Problem 234](#_Toc50132022)

[Reference 235](#_Toc50132023)

[Approach 235](#_Toc50132024)

[Solution 235](#_Toc50132025)

[Time and space complexity 236](#_Toc50132026)

[4. Possible Bipartition ( check If graph is bi-partite) 236](#_Toc50132027)

[Problem 236](#_Toc50132028)

[Reference 237](#_Toc50132029)

[Approach 237](#_Toc50132030)

[Solution 239](#_Toc50132031)

[Time and space complexity 240](#_Toc50132032)

[5. Surrounded Regions 240](#_Toc50132033)

[Problem 240](#_Toc50132034)

[Reference 240](#_Toc50132035)

[Approach 240](#_Toc50132036)

[Solution 241](#_Toc50132037)

[Time and space complexity 241](#_Toc50132038)

[6. Course Schedule II 242](#_Toc50132039)

[Problem 242](#_Toc50132040)

[Reference 242](#_Toc50132041)

[Approach 242](#_Toc50132042)

[Solution 243](#_Toc50132043)

[Time and space complexity 243](#_Toc50132044)

[7. Dijkstra Algorithm (find shortest path from source to all vertex) 243](#_Toc50132045)

[Problem 243](#_Toc50132046)

[Reference 244](#_Toc50132047)

[Approach 244](#_Toc50132048)

[Solution 250](#_Toc50132049)

[Time and space complexity 251](#_Toc50132050)

[8. Cheapest Flights Within K Stops 251](#_Toc50132051)

[Problem 251](#_Toc50132052)

[Reference 252](#_Toc50132053)

[Approach 252](#_Toc50132054)

[Solution 252](#_Toc50132055)

[Time and space complexity 253](#_Toc50132056)

[9. Bellman Ford Algorithm (find shortest path from source to all vertex with negative) 253](#_Toc50132057)

[Problem 253](#_Toc50132058)

[Reference 253](#_Toc50132059)

[Approach 253](#_Toc50132060)

[Solution 255](#_Toc50132061)

[Time and space complexity 256](#_Toc50132062)

[10. DAG Shortest/Longest Path algorithm (find shortest path from source to all vertex with negative) 256](#_Toc50132063)

[Problem 256](#_Toc50132064)

[Reference 256](#_Toc50132065)

[Approach 256](#_Toc50132066)

[Solution 256](#_Toc50132067)

[Time and space complexity 256](#_Toc50132068)

[11. Evaluate Division 257](#_Toc50132069)

[Problem 257](#_Toc50132070)

[Reference 257](#_Toc50132071)

[Approach 257](#_Toc50132072)

[Solution 257](#_Toc50132073)

[Time and space complexity 258](#_Toc50132074)

[17. Backtracking 259](#_Toc50132075)

[1. Rat in a Maze (Maze Solver) 259](#_Toc50132076)

[Problem 259](#_Toc50132077)

[Reference 260](#_Toc50132078)

[Approach 260](#_Toc50132079)

[Solution 260](#_Toc50132080)

[Time and space complexity 260](#_Toc50132081)

[18. Disjoint sets 261](#_Toc50132082)

[1. Detect cycle in un-directed graph 262](#_Toc50132083)

[Problem 262](#_Toc50132084)

[Reference 262](#_Toc50132085)

[Approach 262](#_Toc50132086)

[Solution 262](#_Toc50132087)

[Time and space complexity 263](#_Toc50132088)

[19. Regex 263](#_Toc50132089)

# 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.

Log 10 of positive integer gives number of digit-1.

### 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 concat 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, HASHMAP

### 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 Math.max(max,i-map.get(sum))

### Solution

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

**int** max = 0;

**int** sum = 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)) {

max = MathUtil.*max*(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).

## Find First occurrence of 1 in sorted binary array

### Problem

Given an array of integers, return index of first one in binary array. If no such 1 exist return -1.

### Reference

LEETCODE, ARRAY,BINARY SEARCH

### Approach

To solve it in logn time use binary search. We find middle element and check if it is one and mid-1 is 0 that is the answer.

If 0 is middle search right else search left.

### Solution

public int leftMostColumnWithOne(int[] bm) {

int s=0;

int e=bm.length-1;

while(s<=e) {

int mid = (e+s)/2;

if(bm[mid]==1 && (mid-1<0 || bm[mid-1]==0)) {

return mid;

}

if(elem==0){

s=mid+1;

} else {

e=mid-1;

}

}

return -1;

}

### Time and space complexity

Time - O(logn)

Space – O(1)

## Subarray Sum Equals K

### Problem

Given an array of integers and an integer **k**, you need to find the total number of continuous subarrays whose sum equals to **k**.

Input : arr[] = {10, 2, -2, -20, 10},

k = -10

Output : 3

Subarrays: arr[0...3], arr[1...4], arr[3..4]

have sum exactly equal to -10.

Input : arr[] = {9, 4, 20, 3, 10, 5},

k = 33

Output : 2

Subarrays : arr[0...2], arr[2...4] have sum

exactly equal to 33.

### Reference

LEETCODE, ARRAY, GEEKSFORGEEKS

### Approach

To avoid looping twice we can store sum of nums array as we proceed in map and if at any point we get k sum increment counter.

At any point if we get back sum-k it means we have reached k sum again. So we just update c += map.get(sum - k)

Update map with sum as key and value counter by 1. If not present set to 1 else update value.

### Solution

**public** **int** subarraySum(**int**[] nums, **int** k) {

**if** (nums.length < 1) {

**return** 0;

}

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

**int** c = 0;

**int** sum = 0;

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

sum += nums[i];

**if** (sum == k) {

c++;

}

**if** (map.containsKey(sum - k)) {

c += map.get(sum - k);

}

map.put(sum, map.getOrDefault(sum, 0) + 1);

}

**return** c;

}

### Time and space complexity

Time - O(n)

Space – O(n)

## Jump Game

### Problem

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

**Example 1:**

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

**Output:** true

**Explanation:** Jump 1 step from index 0 to 1, then 3 steps to the last index.

**Example 2:**

**Input:** [3,2,1,0,4]

**Output:** false

**Explanation:** You will always arrive at index 3 no matter what. Its maximum

  jump length is 0, which makes it impossible to reach the last index.

### Reference

LEETCODE, ARRAY, GITHUB,YOUTUBE

### Approach

We will start from 0 and go till last index and keep track of how far we can go for every index. if we reach an index to which it is not possible to go we exit with false.

If loop finishes without exit we return true.

“Far” variable is initialize to 0 and we for every index we set far = max of (far,nums[i]+i).

Example for 3,2,1,04.

1. Farthest we can go for i=0 is at 3 + 0 index. Set far=3

2. Farthest we can go for i=1 is at (2+1) index. far=3

3. Farthest we can go for i=2 is at (1+2) index. far=3

4. Farthest we can go for i=3 is at (0+3) index. far=3

5. we reached index i=4 for which far=3. It means we have reached a position where it is not possible to go. As we can go farthest at 3 index by any combo. Hence return false.

### Solution

**public** **boolean** canJumpGame(**int**[] nums) {

**int** far = 0;

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

**if** (far < i) {

**return** **false**;

}

far = MathUtil.*max*(far, i + nums[i]);

}

**return** **true**;

}

### Time and space complexity

Time - O(n)

Space – O(1)

## First Bad Version

### Problem

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which will return whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

**Example:**

Given n = 5, and version = 4 is the first bad version.

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

### Reference

LEETCODE, ARRAY, GITHUB, BINARY SEARCH

### Approach

Binary search can be used to find first true value among number in logn time.

FFTTTT (The first two are correct version, the rest are bad ones)  
So we can use binary search to find the rightmost F or the leftmost T

To find the leftmost T ---🡪

We will go from i=1 and j=n till i<=j. and check if we found true update j=mid-1 and res=mid else i=mid+1.

So when loop terminates we have res pointing to first true.

### Solution

**public int firstBadVersion(int n) {**

int l=1;

int r=n;

int res=n;

while(l<=r){

int mid=l+((r-l)/2);

if(isBadVersion(mid)) {

r=mid-1;

res=mid;

} else {

l=mid+1;

}

}

return res;

**}**

### Time and space complexity

Time - O(logn)

Space – O(1)

## Squares of a Sorted Array

### Problem

Given an array of integers A sorted in non-decreasing order, return an array of the squares of each number, also in sorted non-decreasing order.

**Example 1:**

**Input:** [-4,-1,0,3,10]

**Output:** [0,1,9,16,100]

**Example 2:**

**Input:** [-7,-3,2,3,11]

**Output:** [4,9,9,49,121]

### Reference

LEETCODE, ARRAY

### Approach

We can take two pointers i=0 and j=a.length-1 and will check squares of a[i] and a[j] will store larger of the two number square in result array. If j element is considered decrement j else increment i.

### Solution

**public int[] sortedSquares(int[] a) {**

int[] arr = new int[a.length];

int c=arr.length-1;

for(int i=0,j=arr.length-1;c>=0;c--){

if(a[i]\*a[i]<a[j]\*a[j]){

arr[c]=a[j]\*a[j];

j--;

} else {

arr[c]=a[i]\*a[i];

i++;

}

}

return arr;

**}**

### Time and space complexity

Time - O(n)

Space – O(n)

## Majority Element

### Problem

Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** ⌊ n/2 ⌋ times.

You may assume that the array is non-empty and the majority element always exist in the array.

**Example 1:**

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

**Output:** 3

**Example 2:**

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

**Output:** 2

### Reference

LEETCODE, ARRAY, Boyer-Moore Voting Algorithm, tech dose

### Approach

**Approach**1 can be to use map and keep on adding element and from with-in loop if particular eleemtn count>n/2 return element. O(n),O(n)

**Approach2** is to use Boyer-Moore Voting Algorithm. Which will guaranty that if majority exist it will return element else there is no such majority element exist. Since we are sure here majority is there we don’t need verify method to iterate array again check count of selected candidate return by Algorithm. O(n),O(1)

Just consider first element as candidate and count=1.if we found same element again c++ else c--. At any point if c reaches 0. It means there are equal number of majority element and rest of elements. We update currenet element as candidate. When loop finished candidate will have majority element.

### Solution

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

**int** candidate = nums[0];

**int** c = 1;

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

**if** (candidate == nums[i]) {

c++;

} **else** {

c--;

}

**if** (c == 0) {

candidate = nums[i];

c = 1;

}

}

**return** candidate;

}

### Time and space complexity

Time - O(n)

Space – O(1)

## Find the Town Judge

### Problem

In a town, there are N people labelled from 1 to N.  There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

1. The town judge trusts nobody.
2. Everybody (except for the town judge) trusts the town judge.
3. There is exactly one person that satisfies properties 1 and 2.

You are given trust, an array of pairs trust[i] = [a, b] representing that the person labelled a trusts the person labelled b.

If the town judge exists and can be identified, return the label of the town judge.  Otherwise, return -1.

**Example 1:**

**Input:** N = 2, trust = [[1,2]]

**Output:** 2

**Example 2:**

**Input:** N = 3, trust = [[1,3],[2,3]]

**Output:** 3

**Example 3:**

**Input:** N = 3, trust = [[1,3],[2,3],[3,1]]

**Output:** -1

**Example 4:**

**Input:** N = 3, trust = [[1,2],[2,3]]

**Output:** -1

**Example 5:**

**Input:** N = 4, trust = [[1,3],[1,4],[2,3],[2,4],[4,3]]

**Output:** 3

### Reference

LEETCODE, ARRAY, GITHUB

### Approach

1. Our interest it to find judge. And it is clear if judge exist all other citizen should have trusted him and judge should trust none. It means judge trustscore should be n-1.
2. We will calculate trustScore of every citizen.i.e if a showed trust on b we will subtract 1 from a total and add 1 to b total. In this way after loop terminates if trsutstore of any element =n-1 return that element. This works because if judge has trusted some other people in that case his trustscore will get decreased from n-1 to n-2. And hence we will get -1.

### Solution

**public** **int** findJudge(**int** n, **int**[][] trust) {

**int**[] trustFactor = **new** **int**[n + 1];

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

trustFactor[trust[i][0]] -= 1;

trustFactor[trust[i][1]] += 1;

}

**for** (**int** k = 1; k <= n; k++) {

**if** (trustFactor[k] == n - 1) {

**return** k;

}

}

**return** -1;

}

### Time and space complexity

Time - O(n)

Space – O(n)

## Flood Fill

### Problem

An image is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, "flood fill" the image.

To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the newColor.

At the end, return the modified image.

**Example 1:**

**Input:**

image = [[1,1,1],[1,1,0],[1,0,1]]

sr = 1, sc = 1, newColor = 2

**Output:** [[2,2,2],[2,2,0],[2,0,1]]

**Explanation:**

From the center of the image (with position (sr, sc) = (1, 1)), all pixels connected

by a path of the same color as the starting pixel are colored with the new color.

Note the bottom corner is not colored 2, because it is not 4-directionally connected

to the starting pixel.

**Note:**

 The length of image and image[0] will be in the range [1, 50].

 The given starting pixel will satisfy 0 <= sr < image.length and 0 <= sc < image[0].length.

### Reference

LEETCODE, ARRAY, GITHUB,DFS

### Approach

1. We will use dfs to go for every possible index. If color is not old-color or it is already colored with new color we return else call dfs function for 4 possible indexes sr-1,sr+1,sc-1,sc+1.

### Solution

**public** **int**[][] floodFill(**int**[][] image, **int** sr, **int** sc, **int** newColor) {

floodFill2(image, sr, sc, newColor, image[sr][sc]);

**return** image;

}

**private** **void** floodFill2(**int**[][] image, **int** sr, **int** sc, **int** newColor, **int** oldColor) {

**if** (image[sr][sc] != oldColor || image[sr][sc] == newColor) {

**return**;

}

image[sr][sc] = newColor;

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

floodFill2(image, sr - 1, sc, newColor, oldColor);

}

**if** (sr + 1 < image.length) {

floodFill2(image, sr + 1, sc, newColor, oldColor);

}

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

floodFill2(image, sr, sc - 1, newColor, oldColor);

}

**if** (sc + 1 < image[0].length) {

floodFill2(image, sr, sc + 1, newColor, oldColor);

}

}

### Time and space complexity

Time - O(sr\*sc)

Space – O(sr\*sc) – due to recursion stack

## Single Element in a Sorted Array

### Problem

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once. Find this single element that appears only once.

**Example 1:**

**Input:** [1,1,2,3,3,4,4,8,8]

**Output:** 2

**Example 2:**

**Input:** [3,3,7,7,10,11,11]

**Output:** 10

### Reference

LEETCODE, ARRAY, GITHUB, YOUTUBE, TECH DOSE

### Approach

1. One approach is to loop from 0 to n and xor element. After loop terminates we have found the unique element.
2. But since data is sorted we can use binary search.
3. If we consider index it is clear that if mid index is odd and mid+1 element is same as mid. All other elements present from low to mid are duplicates. So we make low=mid+1. Else update high=mid
4. Similarly if mid is even. And mid and mid-1 element are same -> All other elements present from low to mid are duplicates. So we make low=mid+1. Else update high=mid.
5. When low < high condition break we have low pointing to the unique element index.

### Solution

**APPROACH 1 -**

public int singleNonDuplicate(int[] nums) {  
        int res=nums[0];  
        for(int i=1;i<nums.length;i++) {  
            res=res^nums[i];  
        }    
        return res;  
    }

**APPROACH** 2 –

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

**int** low = 0;

**int** high = nums.length - 1;

**int** mid;

**while** (low < high) {

mid = low + (high - low) / 2;

**if** ((mid % 2 == 0 && nums[mid] == nums[mid + 1]) || (mid % 2 != 0 && nums[mid] == nums[mid - 1])) {

low = mid + 1;

} **else** {

high = mid;

}

}

**return** nums[low];

}

### Time and space complexity

Time - O(logn)

Space – O(1)

## Remove K Digits

### Problem

Given a non-negative integer *num* represented as a string, remove *k* digits from the number so that the new number is the smallest possible.

**Example 1:**

Input: num = "1432219", k = 3

Output: "1219"

Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.

**Example 2:**

Input: num = "10200", k = 1

Output: "200"

Explanation: Remove the leading 1 and the number is 200. Note that the output must not contain leading zeroes.

**Example 3:**

Input: num = "10", k = 2

Output: "0"

Explanation: Remove all the digits from the number and it is left with nothing which is 0.

### Reference

LEETCODE, ARRAY, GITHUB, YOUTUBE, TECH DOSE

### Approach

1. We can use stack to store elements.
2. Go from 0 to last element. And for each element if stack.peek <= currenet element we push it to stack.
3. We will pop max k times as we want n-k element in result stack.
4. So, basically we will keep popping element and decrement k if k>0 && stack is not empty and stack.peek() > num.charAt(i).
5. Else push element. But if at any point stack is empty and current element to be added is 0. We wont add it to stack and move forward for next element.
6. After above loop completes. And we have k>0 we will pop it. Because it might be possible that data is in increasing order and hence we never popped the stack. E.g. 123456 and k=3. So in such case we need to pop three times. So that resultant will be 123.
7. Now we have stack with required answer. Just prefix stack.pop to a string s. and return s.

### Solution

**public** String removeKdigits(String num, **int** k) {

**if** (num.length() <= k) {

**return** "0";

}

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

**int** i = 0;

**while** (i < num.length()) {

**while** (k > 0 && !stack.isEmpty() && stack.peek() > num.charAt(i)) {

stack.pop();

k--;

}

**if** (!stack.isEmpty() || num.charAt(i) != '0') {

stack.push(num.charAt(i));

}

i++;

}

**while** (k > 0) {

stack.pop();

k--;

}

String s = "";

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

s = stack.pop() + s;

}

**return** s.isEmpty() ? "0" : s;

}

### Time and space complexity

Time - O(n)

Space – O(n)

## Sort Characters By Frequency

### Problem

Given a string, sort it in decreasing order based on the frequency of characters.

**Example 1:**

**Input:**

"tree"

**Output:**

"eert"

**Explanation:**

'e' appears twice while 'r' and 't' both appear once.

So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

**Example 2:**

**Input:**

"cccaaa"

**Output:**

"cccaaa"

**Explanation:**

Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.

Note that "cacaca" is incorrect, as the same characters must be together.

**Example 3:**

**Input:**

"Aabb"

**Output:**

"bbAa"

**Explanation:**

"bbaA" is also a valid answer, but "Aabb" is incorrect.

Note that 'A' and 'a' are treated as two different characters.

### Reference

LEETCODE, ARRAY, GITHUB

### Approach

1. Store frequency and character in array. And sort the array according to frequency.
2. Once sorted loop through element and append characters in StringBuilder.

### Solution

public String frequencySort(String s) {

int[][] arr = new int[128][2];

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

arr[s.charAt(i)][1] += 1;

arr[s.charAt(i)][0] = s.charAt(i);

}

StringBuilder sb = new StringBuilder();

Arrays.sort(arr,(o1,o2)->o2[1]-o1[1]);

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

for(int k=0;k<arr[i][1];k++) {

sb.append((char)arr[i][0]);

}

}

return sb.toString();

}

### Time and space complexity

Time - O(n)

Space – O(256)

## Interval List Intersections

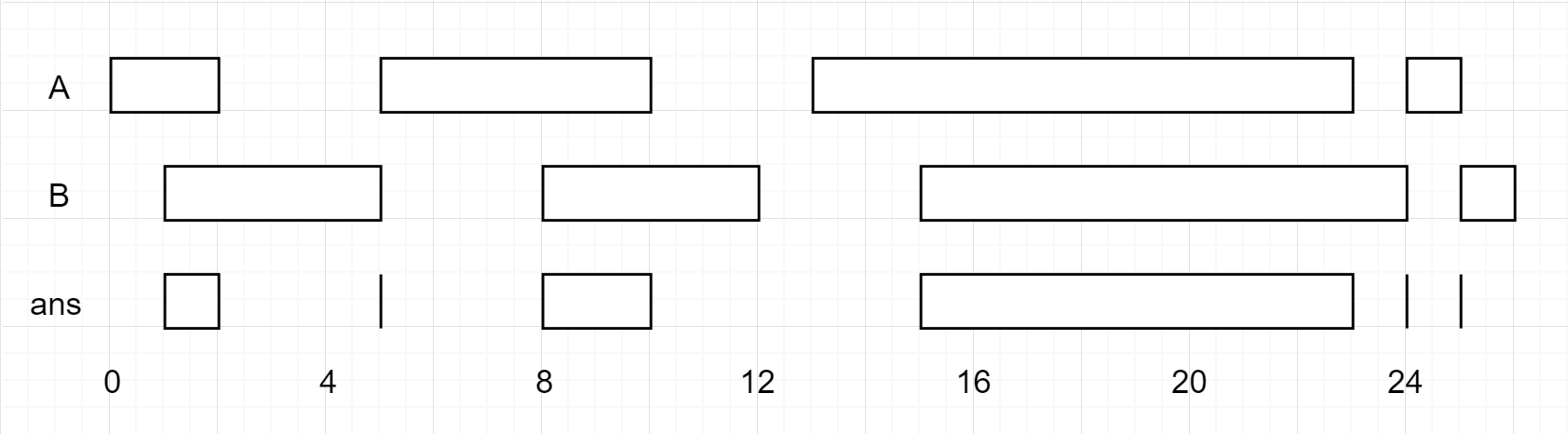
### Problem

Given two lists of **closed** intervals, each list of intervals is pairwise disjoint and in sorted order.

Return the intersection of these two interval lists.

*(Formally, a closed interval [a, b] (with a <= b) denotes the set of real numbers x with a <= x <= b.  The intersection of two closed intervals is a set of real numbers that is either empty, or can be represented as a closed interval.  For example, the intersection of [1, 3] and [2, 4] is [2, 3].)*

**Example 1:**

****

**Input:** A = [[0,2],[5,10],[13,23],[24,25]], B = [[1,5],[8,12],[15,24],[25,26]]

**Output:** [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]

**Reminder:** The inputs and the desired output are lists of Interval objects, and not arrays or lists.

**Note:**

1. 0 <= A.length < 1000
2. 0 <= B.length < 1000
3. 0 <= A[i].start, A[i].end, B[i].start, B[i].end < 10^9

### Reference

LEETCODE, ARRAY, GITHUB

### Approach

1. Take two pointers i and j. find max of first intervals and min of second interval.
2. Check if start <=end add that element interval to list.
3. Increment i if a[i][1]<=b[j][1] else increment j.

### Solution

**public** **int**[][] intervalIntersection(**int**[][] arr, **int**[][] brr) {

List<**int**[]> output = **new** ArrayList<>();

**int** i = 0;

**int** j = 0;

**while** (i < arr.length && j < brr.length) {

**int** start = Math.*max*(arr[i][0], brr[j][0]);

**int** end = Math.*min*(arr[i][1], brr[j][1]);

**if** (start <= end) {

output.add(**new** **int**[] { start, end });

}

**if** (arr[i][1] <= brr[j][1]) {

i++;

} **else** {

j++;

}

}

**return** output.toArray(**new** **int**[output.size()][2]);

}

### Time and space complexity

Time - O(n)

Space – O(n)

## Search a 2D Matrix (data is sorted in matrix)

### Problem

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

* Integers in each row are sorted from left to right.
* The first integer of each row is greater than the last integer of the previous row.

**Example 1:**

**Input:**

matrix = [

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

target = 3

**Output:** true

**Example 2:**

**Input:**

matrix = [

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

target = 13

**Output:** false

### Reference

LEETCODE, ARRAY, GITHUB

### Approach

1. Apply binary search to find the element. The only trick is to find the row and column index to find the data. First find the middle index by considering entire 2d array as single array. So, In our example middle index will be 6. Now rowindex = midIndex/RowLength and colIndex = midIndex%RowLength

### Solution

**public** **boolean** searchMatrix(**int**[][] matrix, **int** target) {

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

**int** start = 0;

**int** end = matrix[0].length \* matrix.length - 1;

**while** (start <= end) {

**int** midIndex = start + (end - start) / 2;

**int** mid = matrix[midIndex / matrix[0].length][midIndex % matrix[0].length];

**if** (mid == target) {

**return** **true**;

} **else** **if** (mid < target) {

start = midIndex + 1;

} **else** {

end = midIndex - 1;

}

}

}

**return** **false**;

}

### Time and space complexity

Time - O(lognm)

Space – O(1)

## Two City Scheduling

### Problem

There are 2N people a company is planning to interview. The cost of flying the i-th person to city A is costs[i][0], and the cost of flying the i-th person to city B is costs[i][1].

Return the minimum cost to fly every person to a city such that exactly N people arrive in each city.

**Example 1:**

**Input:** [[10,20],[30,200],[400,50],[30,20]]

**Output:** 110

**Explanation:**

The first person goes to city A for a cost of 10.

The second person goes to city A for a cost of 30.

The third person goes to city B for a cost of 50.

The fourth person goes to city B for a cost of 20.

The total minimum cost is 10 + 30 + 50 + 20 = 110 to have half the people interviewing in each city.

**Note:**

1. 1 <= costs.length <= 100
2. It is guaranteed that costs.length is even.
3. 1 <= costs[i][0], costs[i][1] <= 1000

### Reference

LEETCODE, ARRAY, GITHUB, GREEDY

### Approach

Based on the problem, it is clear that we have to maximize profit. We can use a greedy approach here. But what is the profit here? Profit can be made by sending a person i to cityA or cityB. So what we can do, find the cost difference between two cities for all the person.

After sorting the value, take the first half of array as A city and remaining half as B.

Example:

input: [10,20], [30,200], [400,50],[30,20]

diff: -10 -170 350 10

sorted: [30,200], [10,20], [30,20], [400,50]

Now take 30, 10 from A & 20, 50 from B. So, total=110

### Solution

**public** **int** twoCitySchedCost(**int**[][] costs) {

Arrays.*sort*(costs, (a, b) -> a[0] - a[1] - b[0] + b[1]);

**int** sum = 0;

**for** (**int** ii = 0; ii < costs.length; ii++) {

**if** (ii < costs.length / 2) {

sum += costs[ii][0];

} **else** {

sum += costs[ii][1];

}

}

**return** sum;

}

### Time and space complexity

Time - O(nlogn)

Space – O(1)

## Random Pick with Weight

### Problem

Given an array w of positive integers, where w[i] describes the weight of index i, write a function pickIndex  which randomly picks an index in proportion to its weight.

Note:

1. 1 <= w.length <= 10000
2. 1 <= w[i] <= 10^5
3. pickIndex will be called at most 10000 times.

**Example 1:**

**Input:**

["Solution","pickIndex"]

[[[1]],[]]

**Output:** [null,0]

**Example 2:**

**Input:**

["Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"]

[[[1,3]],[],[],[],[],[]]

**Output:** [null,0,1,1,1,0]

**Explanation of Input Syntax:**

The input is two lists: the subroutines called and their arguments. Solution's constructor has one argument, the array w. pickIndex has no arguments. Arguments are always wrapped with a list, even if there aren't any.

### Reference

LEETCODE, ARRAY, GITHUB, RANDOM, YOUTUBE

### Approach

We need to return index randomly for every call of pickIndex method. But the catch is the probability of returning any index should be dependent on weight associated with that index.

e.g. - > [1,3,4] here probability of selecting 1 and returning 0 index should be 1/8, for 3 should be 3/8 and for 4 should 4/8. i.e. chances of returning index 2 would be highest.

So, we can keep sum of elements in w array.

e.g. -> [1,4,8]

So now we will call random next int using value as last element. Now we will get random value from 0 to 7 in our example. Now we just need to find the index at which array element is greater than generated value. We can use binary search for that. As data is sorted.

(similar to finding insert position of element in sorted array)

### Solution

**class** Solution {

**private** **int**[] w;

**public** Solution(**int**[] w1) {

w=**new** **int**[w1.length];

w[0]=w1[0];

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

w[i]=w[i-1]+w1[i];

}

}

**public** **int** pickIndex() {

**int** i = **new** Random().nextInt(w[w.length-1])+1;

**int** start = 0;

**int** end = w.length-1;

**while**(start<=end) {

**int** mid = end+(start-end)/2;

**if**(w[mid]==i) {

**return** mid;

} **else** **if**(w[mid]<i){

start=mid+1;

} **else** {

end=mid-1;

}

}

**return** start;

}

}

### Time and space complexity

Time - O(logn) for pick method

Space – O(1) as no extra space taken apart from array needed in question

## Remove Duplicates from Sorted Array

### Problem

Given a sorted array nums, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Example 1:**

Given nums = **[1,1,2]**,

Your function should return length = **2**, with the first two elements of *nums* being **1** and **2** respectively.

It doesn't matter what you leave beyond the returned length.

**Example 2:**

Given nums = **[0,0,1,1,1,2,2,3,3,4]**,

Your function should return length = **5**, with the first five elements of *nums* being modified to **0**, **1**, **2**, **3**, and **4** respectively.

It doesn't matter what values are set beyond the returned length.

**Clarification:**

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means modification to the input array will be known to the caller as well.

Internally you can think of this:

// **nums** is passed in by reference. (i.e., without making a copy)

int len = removeDuplicates(nums);

// any modification to **nums** in your function would be known by the caller.

// using the length returned by your function, it prints the first **len** elements.

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

    print(nums[i]);

}

### Reference

LEETCODE, ARRAY, GITHUB

### Approach

We take j=0 and now loop through array from i=1 to n. we will increment j only when we found different element. In that case we first increment j and then add ith element to j position.

When loop terminates j+1 will be new length of array.

### Solution

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

**if**(nums.length==0){

**return** 0;

}

**int** j=0;

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

**if**(nums[j]!=nums[i]){

nums[++j]=nums[i];

}

}

**return** j+1;

}

### Time and space complexity

Time - O(n)

Space – O(1)

## Duplicate Zeros

### Problem

Given a fixed length array arr of integers, duplicate each occurrence of zero, shifting the remaining elements to the right.

Note that elements beyond the length of the original array are not written.

Do the above modifications to the input array **in place**, do not return anything from your function.

**Example 1:**

**Input:** [1,0,2,3,0,4,5,0]

**Output:** null

**Explanation:** After calling your function, the **input** array is modified to: [1,0,0,2,3,0,0,4]

**Example 2:**

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

**Output:** null

**Explanation:** After calling your function, the **input** array is modified to: [1,2,3]

**Note:**

1. 1 <= arr.length <= 10000
2. 0 <= arr[i] <= 9

### Reference

LEETCODE, ARRAY, GITHUB

### Approach

Simple solution is to iterate array from 0 to length-1. And for each i if arr[i]==0, shift element from j=length-1 to j>i.

Arr[j]=arr[j-1]

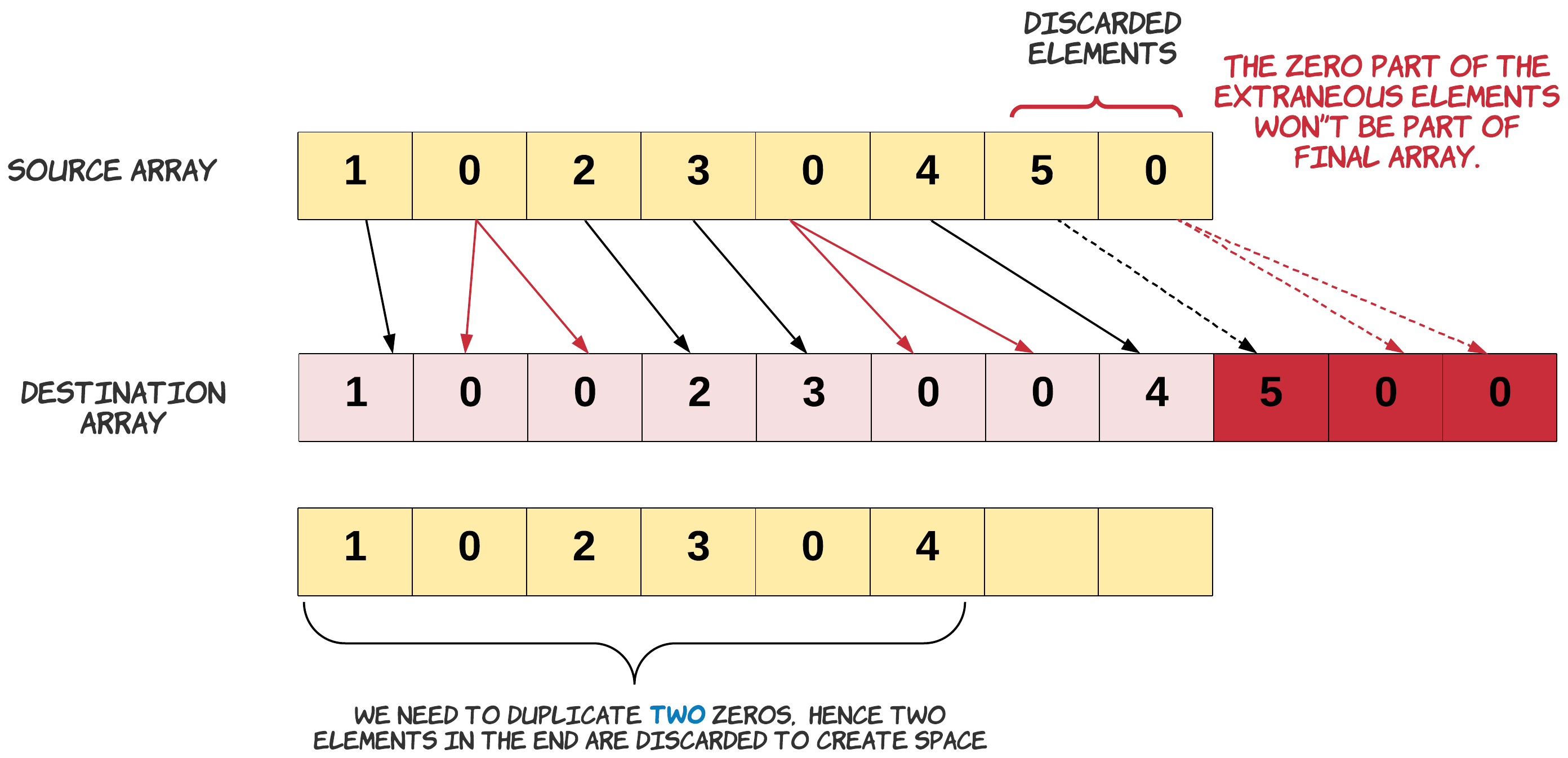
**Approach 2**

If we know the number of elements which would be discarded from the end of the array, we can copy the rest. How do we find out how many elements would be discarded in the end? The number would be equal to the number of extra zeros which would be added to the array. The extra zero would create space for itself by pushing out an element from the end of the array.

Once we know how many elements from the original array would be part of the final array, we can just start copying from the end. Copying from the end ensures we don't lose any element since, the last few extraneous elements can be overwritten.

**Algorithm**

1. Find the number of zeros which would be duplicated. Let's call it possible\_dups. We do need to make sure we are not counting the zeros which would be trimmed off. Since, the discarded zeros won't be part of the final array. The count of possible\_dups would give us the number of elements to be trimmed off the original array. Hence at any point, length\_ - possible\_dups is the number of elements which would be included in the final array.



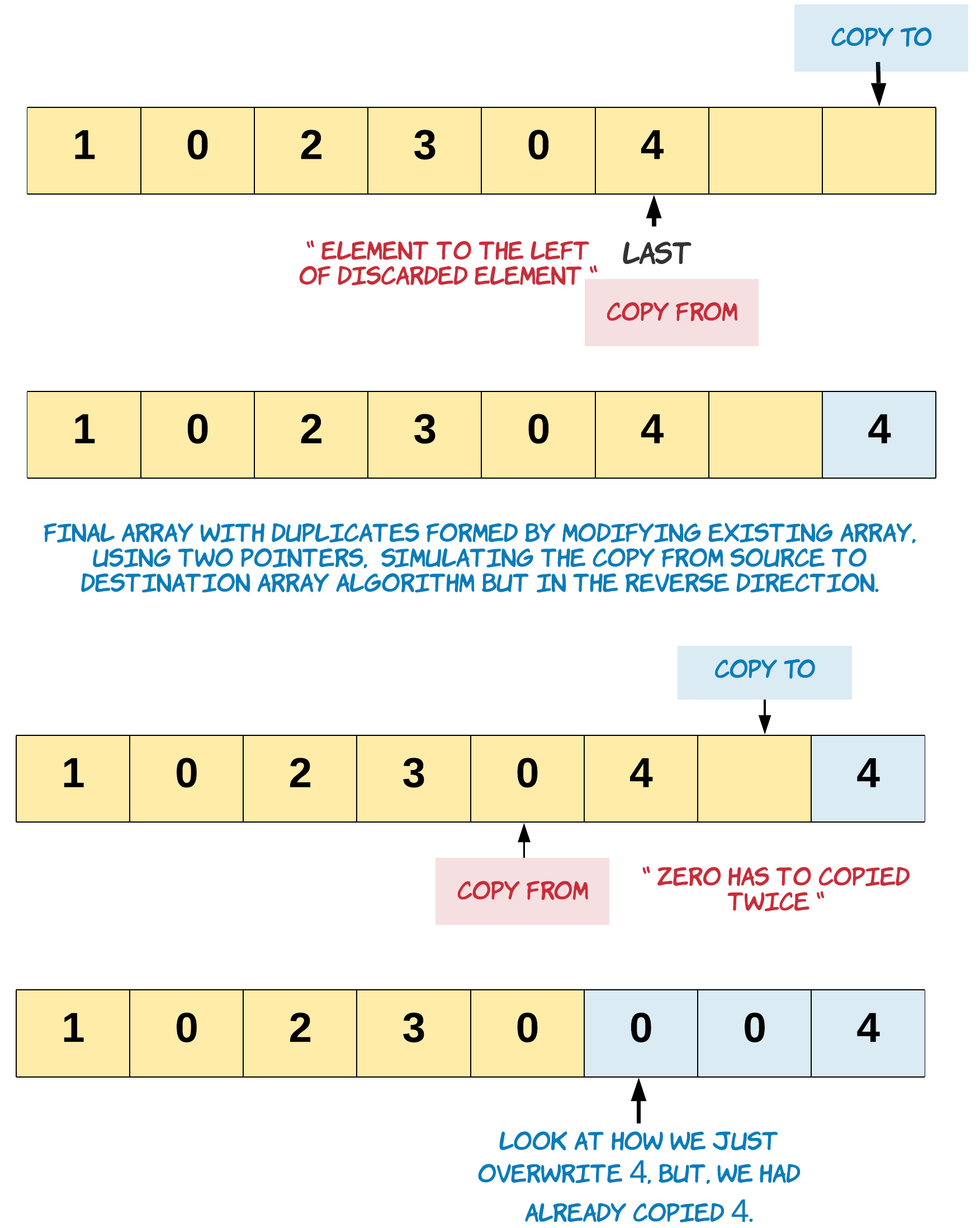
Note: In the diagram above we just show source and destination array for understanding purpose. We will be doing these operations only on one array.

1. Handle the edge case for a zero present on the boundary of the leftover elements.

Let's talk about the edge case of this problem. We need to be extra careful when we are duplicating the zeros in the leftover array. This care should be taken for the zero which is lying on the boundary. Since, this zero might be counted as with possible duplicates, or may be just got included in the left over when there was no space left to accommodate its duplicate. If it is part of the possible\_dups we would want to duplicate it otherwise we don't.

An example of the edge case is - [8,4,5,0,0,0,0,7]. In this array there is space to accommodate the duplicates of first and second occurrences of zero. But we don't have enough space for the duplicate of the third occurrence of zero. Hence when we are copying we need to make sure for the third occurrence we don't copy twice. Result = [8,4,5,0,0,0,0,0]

1. Iterate the array from the end and copy a non-zero element once and zero element twice. When we say we discard the extraneous elements, it simply means we start from the left of the extraneous elements and start overwriting them with new values, eventually right shifting the left over elements and creating space for all the duplicated elements in the array.



### Solution

**public** **void** duplicateZeros(**int**[] arr) {

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

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

**for**(**int** j=arr.length-1;j>i;j--) {

arr[j]=arr[j-1];

}

i++;

}

}}

**public** **void** duplicateZeros(**int**[] arr) {

**int** zeroC = 0;

**int** size = arr.length - 1;

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

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

**if** (i == size - zeroC) {

arr[size] = 0;

size--;

**break**;

}

zeroC++;

}

}

**for** (**int** i = size - zeroC; i >= 0; i--) {

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

arr[i + zeroC] = 0;

zeroC--;

arr[i + zeroC] = 0;

} **else** {

arr[i + zeroC] = arr[i];

}

}

}

### Time and space complexity

Time - O(n)

Space – O(1)

## H-Index II

### Problem

Given an array of citations **sorted in ascending order**(each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the [definition of h-index on Wikipedia](https://en.wikipedia.org/wiki/H-index): "A scientist has index *h* if *h* of his/her *N* papers have **at least** *h* citations each, and the other *N − h* papers have **no more than** *h*citations each."

**Example:**

**Input:** citations = [0,1,3,5,6]

**Output:** 3

**Explanation:** [0,1,3,5,6] means the researcher has 5 papers in total and each of them had

received 0, 1, 3, 5, 6 citations respectively.

  Since the researcher has 3 papers with **at least** 3 citations each and the remaining

  two with **no more than** 3 citations each, her h-index is 3.

**Note:**

If there are several possible values for h, the maximum one is taken as the h-index.

### Reference

Array,binary search,leetcode

### Approach

Naive approach is to use linear search and go from 0 to length and check if element at current index >= length-currentIndex. If condition true return length-currentIndex.

If loop finishes return 0.

Better approach is to use binary search as we know data is sorted.

Apply classic binary search

if arr[mid]==arr[length-mid] return length-mid.

else if arr[mid]>arr[length-mid] set end=mid-1

else set start=mid+1;

once loop terminates we know start will be at correct position and since we wanted to return count. Answer will be length-start.

### Solution

**public** **int** hIndex(**int**[] citations) {

**int** start = 0;

**int** end = citations.length - 1;

**while** (start <= end) {

**int** mid = end - (end - start) / 2;

**if** (citations[mid] == citations.length - mid) {

**return** citations.length - mid;

} **else** **if** (citations[mid] > citations.length - mid) {

end = mid - 1;

} **else** {

start = mid + 1;

}

}

**return** citations.length - start;

}

### Time and space complexity

time - o(logn)

space - o(1)

## Create Target Array

### Problem

Given two arrays of integers nums and index. Your task is to create *target* array under the following rules:

* Initially *target* array is empty.
* From left to right read nums[i] and index[i], insert at index index[i] the value nums[i] in *target* array.
* Repeat the previous step until there are no elements to read in nums and index.

Return the *target* array.

It is guaranteed that the insertion operations will be valid.

**Example 1:**

**Input:** nums = [0,1,2,3,4], index = [0,1,2,2,1]

**Output:** [0,4,1,3,2]

**Explanation:**

nums index target

0 0 [0]

1 1 [0,1]

2 2 [0,1,2]

3 2 [0,1,3,2]

4 1 [0,4,1,3,2]

**Example 2:**

**Input:** nums = [1,2,3,4,0], index = [0,1,2,3,0]

**Output:** [0,1,2,3,4]

**Explanation:**

nums index target

1 0 [1]

2 1 [1,2]

3 2 [1,2,3]

4 3 [1,2,3,4]

0 0 [0,1,2,3,4]

**Example 3:**

**Input:** nums = [1], index = [0]

**Output:** [1]

**Constraints:**

* 1 <= nums.length, index.length <= 100
* nums.length == index.length
* 0 <= nums[i] <= 100
* 0 <= index[i] <= i

### Reference

Array,shift,leetcode

### Approach

Go from 1 to length and check if i> index[i]. if yes perform shift from index[i] to i

Else just add the element.

### Solution

**public** **int**[] createTargetArray(**int**[] nums, **int**[] index) {

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

op[0] = nums[0];

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

**int** ind = index[i];

**if** (ind < i) {

**for** (**int** j = i; j > ind; j--) {

op[j] = op[j - 1];

}

}

op[ind] = nums[i];

}

**return** op;

}

### Time and space complexity

time - o(n2)

space - o(n)

## Split a String in Balanced Strings

### Problem

*Balanced* strings are those who have equal quantity of 'L' and 'R' characters.

Given a balanced string s split it in the maximum amount of balanced strings.

Return the maximum amount of splitted balanced strings.

**Example 1:**

**Input:** s = "RLRRLLRLRL"

**Output:** 4

**Explanation:** s can be split into "RL", "RRLL", "RL", "RL", each substring contains same number of 'L' and 'R'.

**Example 2:**

**Input:** s = "RLLLLRRRLR"

**Output:** 3

**Explanation:** s can be split into "RL", "LLLRRR", "LR", each substring contains same number of 'L' and 'R'.

**Example 3:**

**Input:** s = "LLLLRRRR"

**Output:** 1

**Explanation:** s can be split into "LLLLRRRR".

**Example 4:**

**Input:** s = "RLRRRLLRLL"

**Output:** 2

**Explanation:** s can be split into "RL", "RRRLLRLL", since each substring contains an equal number of 'L' and 'R'

**Constraints:**

* 1 <= s.length <= 1000
* s[i] = 'L' or 'R'

### Reference

Array,String,leetcode

### Approach

Take var j and set it to 0.

Go from 0 to length and check if current element is R. if yes increment j else decrement.

If j==0 increment count.

Once loop finishes return count.

### Solution

**public** **int** balancedStringSplit(String s) {

**int** c = 0;

**int** j = 0;

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

j += s.charAt(i) == 'R' ? 1 : -1;

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

c++;

}

}

**return** c;

}

### Time and space complexity

time - o(n)

space - o(1)

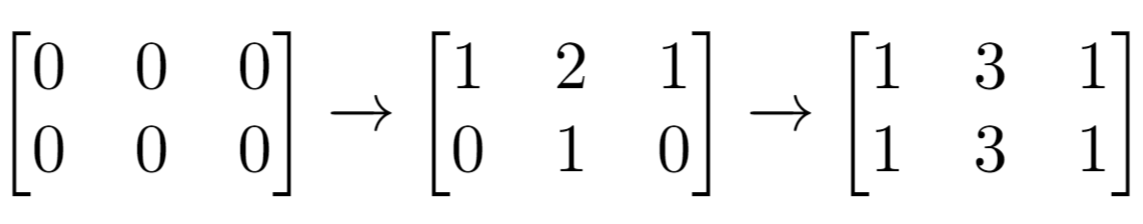
## Cells with Odd Values in a Matrix

### Problem

Given n and m which are the dimensions of a matrix initialized by zeros and given an array indices where indices[i] = [ri, ci]. For each pair of [ri, ci] you have to increment all cells in row ri and column ci by 1.

Return *the number of cells with odd values* in the matrix after applying the increment to all indices.

**Example 1:**



**Input:** n = 2, m = 3, indices = [[0,1],[1,1]]

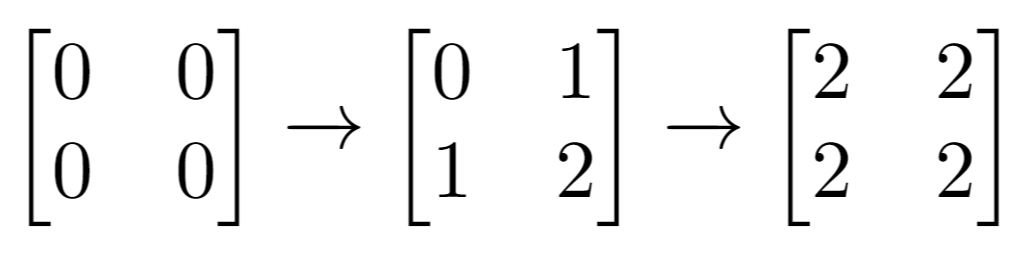
**Output:** 6

**Explanation:** Initial matrix = [[0,0,0],[0,0,0]].

After applying first increment it becomes [[1,2,1],[0,1,0]].

The final matrix will be [[1,3,1],[1,3,1]] which contains 6 odd numbers.

**Example 2:**



**Input:** n = 2, m = 2, indices = [[1,1],[0,0]]

**Output:** 0

**Explanation:** Final matrix = [[2,2],[2,2]]. There is no odd number in the final matrix.

**Constraints:**

* 1 <= n <= 50
* 1 <= m <= 50
* 1 <= indices.length <= 100
* 0 <= indices[i][0] < n
* 0 <= indices[i][1] < m

### Reference

Array2D, leetcode

### Approach

1. Take two 1D Boolean array. rows and cols.
2. Now iterate indices and change rows and cols flag on every occurrence.
3. After loop finishes every row and col will have true if that particular index is odd.
4. So, now iterate row\*col and check if row[i]^col[j] is true. If yes c++.
5. It works here as we know that sum of odd number is even and sum of even number is even. So to be odd count at particular index i,j . one of them must be odd only.

### Solution

**public** **int** oddCells(**int** n, **int** m, **int**[][] indices) {

**boolean**[] rows = **new** **boolean**[n];

**boolean**[] cols = **new** **boolean**[m];

**for** (**int**[] ind : indices) {

rows[ind[0]] = !rows[ind[0]];

cols[ind[1]] = !cols[ind[1]];

}

**int** res = 0;

**for** (**int** r = 0; r < n; r++) {

**for** (**int** c = 0; c < m; c++) {

**if** (rows[r] ^ cols[c])

res++;

}

}

**return** res;

}

### Time and space complexity

time - o(nm)

space - o(n+m)

## Print all permutations of array

### Problem

Given a collection of **distinct** integers, return all possible permutations.

**Example:**

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

**Output:**

[

[1,2,3],

[1,3,2],

[2,1,3],

[2,3,1],

[3,1,2],

[3,2,1]

]

### Reference

Array, leetcode, backtracking

### Approach

1. Implement recursive function where we pass two fields array and index.
2. For each position, starting from 0 we will try to keep that index fix and move ahead with next set of recursive call.
3. For example, in below tree we will try to fix first index by swapping A with A, B and C. then again call recursively. In below tree call will go like pre-order traversal.
4. Since we want to reset the array while back-tracking we swap it again.



### Solution

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

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

permutations(nums, 0, list);

**return** list;

}

**private** **void** permutations(**int**[] nums, **int** i, List<List<Integer>> list) {

**if** (i == nums.length) {

List<Integer> sublist = **new** ArrayList<>();

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

sublist.add(nums[j]);

}

list.add(sublist);

**return**;

}

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

ArrayUtil.*swap*(nums, i, start);

permutations(nums, i + 1, list);

ArrayUtil.*swap*(nums, i, start); //backtrack. i.e reset array

}

}

### Time and space complexity

time - o(n!)

space – o(n!)

## Print all unique permutations of array

### Problem

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

**Example:**

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

**Output:**

[

[1,1,2],

[1,2,1],

[2,1,1]

]

### Reference

Array, leetcode, backtracking

### Approach

1. Same as above question. Just to keep track of duplicate we will take set and add element to it. If same element appears, again we will skip the recursive call.

### Solution

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

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

permute(nums, 0, list);

**return** list;

}

**private** **void** permute(**int**[] nums, **int** i, List<List<Integer>> list) {

**if** (i == nums.length) {

List<Integer> l1 = **new** ArrayList<>();

**for** (**int** n : nums) {

l1.add(n);

}

list.add(l1);

**return**;

}

Set<Integer> set = **new** HashSet<>();

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

**if** (set.add(nums[s])) {

ArrayUtil.*swap*(nums, s, i);

permute(nums, i + 1, list);

ArrayUtil.*swap*(nums, s, i);

}

}

}

### Time and space complexity

time - o(n!)

space – o(n!)

## Contains duplicate II

### Problem

Given an array of integers and an integer *k*, find out whether there are two distinct indices *i* and *j* in the array such that **nums[i] = nums[j]** and the **absolute** difference between *i* and *j* is at most *k*.

**Example 1:**

**Input:** nums = [1,2,3,1], k = 3

**Output:** true

**Example 2:**

**Input:** nums = [1,0,1,1], k = 1

**Output:** true

**Example 3:**

**Input:** nums = [1,2,3,1,2,3], k = 2

**Output:** false

### Reference

Array, leetcode

### Approach

Approach 1-

1. Use map to keep key as element and value as index.
2. If map already contain element and difference of index <=k return true.
3. Add element in every case to map to keep value i.e. index latest.

Approach 2- (sliding window)

1. Instead of using map we can use set also in approach 1 and to reduce the map size from n to k we can start removing item from map as soon as map.size > k.
2. In such case window size always, remain under k. now we do not need index also as if item is present in the map it is under k elements as map capacity is k.
3. The above logic mentioned in point 1 and 2 is used in approach 2 for set.

### Solution

**Approach 1-**

**public** **boolean** containsNearbyDuplicate(**int**[] nums, **int** k) {

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

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

**if** (map.containsKey(nums[i]) && i - map.get(nums[i]) <= k) {

**return** **true**;

}

map.put(nums[i], i);

}

**return** **false**;

}

**Approach 2-**

**public** **boolean** containsNearbyDuplicate(**int**[] nums, **int** k) {

Set<Integer> set = **new** HashSet<>();

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

**if**(set.contains(nums[i])) {

**return** **true**;

}

set.add(nums[i]);

**if**(set.size()>k) {

set.remove(nums[i-k]);//make window size k again by removing nums[i-k] element

}

}

**return** **false**;

}

### Time and space complexity

time - o(n)

space – o(k) – approach 2 and o(n) for approach 1

## Contains duplicate III

### Problem

Given an array of integers, find out whether there are two distinct indices, *i* and *j* in the array such that the **absolute** difference between **nums[i]** and **nums[j]** is at most *t* and the **absolute** difference between *i* and *j* is at most *k*.

**Example 1:**

**Input:** nums = [1,2,3,1], k = 3, t = 0

**Output:** true

**Example 2:**

**Input:** nums = [1,0,1,1], k = 1, t = 2

**Output:** true

**Example 3:**

**Input:** nums = [1,5,9,1,5,9], k = 2, t = 3

**Output:** false

### Reference

Array, leetcode

### Approach

1. It is just like approach 2 of previous question here instead of exact match we need to find closest neighbor in array under k size window.
2. Core logic: sliding window + tree set. Maintain the tree set in such a way that it always contains K elements and slide them accordingly (remove 1st element when we reach size >=k, etc).
3. The absolute diff should always be utmost t means that we need to check the greatest low and lowest high out of all the elements. If that satisfies the given condition, return true. Else continue for next element.

Why it works -> we can see to make difference of current element ‘x’ with ’y’ to be at most ‘t’. in such case from the entire set we need to pick the next largest and smallest element close to x as y.

1. floor(x) will give next greatest element less than x.
2. ceiling(x) will give next smallest element greater than x.

### Solution

**public** **boolean** containsNearbyAlmostDuplicate(**int**[] nums, **int** k, **int** t) {

TreeSet<Long> set = **new** TreeSet<>();

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

Long floor = set.floor((**long**)nums[i]);

**if** (floor!=**null** && nums[i]-floor<=t) {

**return** **true**;

}

Long ceiling = set.ceiling((**long**)nums[i]);

**if**(ceiling!=**null** && ceiling - nums[i]<=t) {

**return** **true**;

}

set.add((**long**)nums[i]);

**if**(set.size()>k) {

set.remove((**long**)nums[i-k]);

}

}

**return** **false**;

}

### Time and space complexity

time – o(n)

space – o(nlogk)

## Next Permutation (Next greater element)

### Problem

Implement **next permutation**, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be [**in-place**](http://en.wikipedia.org/wiki/In-place_algorithm) and use only constant extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

1,2,3 → 1,3,2  
3,2,1 → 1,2,3  
1,1,5 → 1,5,1

### Reference

Array, leetcode

### Approach

1. We will start searching element from right to left until elements are increasing.
2. If index reaches 0. It means next greater element is not possible and hence we just reverse the entire array.
3. Else index-1 is the desired index. after that index we need to find the element from right to left till index whose value is greater than element present at index.
4. Now just swap the two elements present at index-1 and j.
5. Reverse the array after the swapped index location i.

For example –

Input – 1 3 4 2 0

Here by applying above algo step 1 we get i=2, so element at i=1 i.e. 3 needs to be swapped with next greater element from right to left.

Searching from right to left we get j=2 i.e. element 4. So swap both of them

1 4 3 2 0

Now we just need to reverse the remaining element after element 4. We get desired result –

1 4 0 2 3

### Solution

**public** **void** nextPermutation(**int**[] nums) {

**int** i = nums.length-1;

**while**(i>0 && nums[i-1]>=nums[i]) {

i--;

}

**if**(i>0) {

**int** j = nums.length-1;

**while**(nums[j]<=nums[i-1]) {

j--;

}

swap(nums,i-1,j);

}

reverse(nums,i);

}

**private** **void** reverse(**int**[] nums, **int** l) {

**int** h=nums.length-1;

**while**(l<h) {

swap(nums,h--,l++);

}

}

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

nums[i] ^= nums[j];

nums[j] ^= nums[i];

nums[i] ^= nums[j];

}

### Time and space complexity

time – o(n)

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)

## Permutation in String

### Problem

Given two strings **s1** and **s2**, write a function to return true if **s2** contains the permutation of **s1**. In other words, one of the first string's permutations is the **substring** of the second string.

**Example 1:**

**Input:** s1 = "ab" s2 = "eidbaooo"

**Output:** True

**Explanation:** s2 contains one permutation of s1 ("ba").

**Example 2:**

**Input:** s1= "ab" s2 = "eidboaoo"

**Output:** False

**Note:**

1. The input strings only contain lower case letters.
2. The length of both given strings is in range [1, 10,000].

### Reference

LEETCODE, STRING, GITHUB

### Approach

Use sliding window technique along with anagram check. And for s1 increment value and for s2 decrement it.

And iterate from 0 to s2.length – s1.length. and compare all elements are zero if found return true else check for next window.

### Solution

**public** **boolean** checkPermutationInclusion(String s1, String s2) {

**if** (s1.length() > s2.length()) {

**return** **false**;

}

**int**[] ch = **new** **int**[26];

**for** (**int** i = 0; i < s1.length(); i++) {

ch[s1.charAt(i) - 'a'] += 1;

ch[s2.charAt(i) - 'a'] -= 1;

}

**int** i = 0;

**for** (; i < s2.length() - s1.length(); i++) {

**if** (*checkAnagram*(ch)) {

**return** **true**;

}

ch[s2.charAt(i) - 'a'] += 1;

ch[s2.charAt(i + s1.length()) - 'a'] -= 1;

}

**return** *checkAnagram*(ch);

}

**private** **boolean** checkAnagram(**int**[] arr) {

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

**if** (arr[j] != 0) {

**return** **false**;

}

}

**return** **true**;

}

### Time and space complexity

Time - O(n)

Space – 2\*o(26)

## Validate IP Address

### Problem

Write a function to check whether an input string is a valid IPv4 address or IPv6 address or neither.

**IPv4** addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots ("."), e.g.,172.16.254.1;

Besides, leading zeros in the IPv4 is invalid. For example, the address 172.16.254.01 is invalid.

**IPv6** addresses are represented as eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons (":"). For example, the address 2001:0db8:85a3:0000:0000:8a2e:0370:7334 is a valid one. Also, we could omit some leading zeros among four hexadecimal digits and some low-case characters in the address to upper-case ones, so 2001:db8:85a3:0:0:8A2E:0370:7334 is also a valid IPv6 address(Omit leading zeros and using upper cases).

However, we don't replace a consecutive group of zero value with a single empty group using two consecutive colons (::) to pursue simplicity. For example, 2001:0db8:85a3::8A2E:0370:7334 is an invalid IPv6 address.

Besides, extra leading zeros in the IPv6 is also invalid. For example, the address 02001:0db8:85a3:0000:0000:8a2e:0370:7334 is invalid.

**Note:** You may assume there is no extra space or special characters in the input string.

**Example 1:**

**Input:** "172.16.254.1"

**Output:** "IPv4"

**Explanation:** This is a valid IPv4 address, return "IPv4".

**Example 2:**

**Input:** "2001:0db8:85a3:0:0:8A2E:0370:7334"

**Output:** "IPv6"

**Explanation:** This is a valid IPv6 address, return "IPv6".

**Example 3:**

**Input:** "256.256.256.256"

**Output:** "Neither"

**Explanation:** This is neither a IPv4 address nor a IPv6 address.

### Reference

LEETCODE, STRING, GITHUB, REGEX

### Approach

One approach is to use regex to validate ip and in second approach we can apply java logic and meet the conditions.

**Regex** – (see techdoes video for regex different rule)

For valid ipv4 we must have rule – [0-255].[0-255].[0-255].[0-255]

For valid ipv6 we must have rule – [0-9a-fA-F].[0-9a-fA-F].[0-9a-fA-F].[0-9a-fA-F].[0-9a-fA-F].[0-9a-fA-F].[0-9a-fA-F].[0-9a-fA-F]

Now for ipv4 we will make regex for 1 digit 2digit and 3 digit number

([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])

Now repeat above regex separated by dot 3 times and at last digit again.

For ipv6 we will make regex –

([0-9a-fA-F]{1,4})

It means repeat above characters 1 to 4 times. We will append : 7 times and at last above expression again.

^ - indicates that regex must start with.

$ - indicates it should end.

| - used for OR.

[0-9] – means single digit in range 0 to 9.

[0-9a-fA-F] – means one digit which can be 0 to 9 or a to z or A-Z

{1,4} – repeat regex 1 to 4 times.

2[1-9] – means 2 followed by any digit from 1 to 9.

\* - means 0 to many

+ - means 1 to many

### Solution

**public** String validIPAddress(String ip) {

**if** (ip.isEmpty()||ip.charAt(ip.length() - 1)==':'||ip.charAt(ip.length() - 1)=='.') {

**return** "Neither";

}

String ipv4[] = ip.split("\\.");

**if** (ipv4.length == 4) {

**for** (**int** i = 0; i < ipv4.length; i++) {

**try** {

**int** val = Integer.*parseInt*(ipv4[i]);

**if** (ipv4[i].charAt(0)=='-'||val>255||(ipv4[i].charAt(0) == '0'&&ipv4[i].length()>1)){

**return** "Neither";

}

} **catch** (Exception e) {

**return** "Neither";

}

}

**return** "IPv4";

}

String ipv6[] = ip.split(":");

**if** (ipv6.length == 8) {

**for** (**int** i = 0; i < ipv6.length; i++) {

**try** {

**int** val = Integer.*parseInt*(ipv6[i], 16);

**if** (ipv6[i].charAt(0)=='-'||val>65535||(ipv6[i].length() > 4)) {

**return** "Neither";

}

} **catch** (Exception e) {

**return** "Neither";

}

}

**return** "IPv6";

}

**return** "Neither";

}

**public** String validIPAddress(String ip) {

String ipv4 = "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])";

String ipv6 = "([0-9a-fA-F]{1,4})";

Pattern pipv4 = Pattern.*compile*("^(" + ipv4 + "\\.){3}" + ipv4 + "$");

Pattern pipv6 = Pattern.*compile*("^(" + ipv6 + "\\:){7}" + ipv6 + "$");

**if** (pipv4.matcher(ip).matches()) {

**return** "IPv4";

} **else** **if** (pipv6.matcher(ip).matches()) {

**return** "IPv6";

}

**return** "Neither";

}

### Time and space complexity

Time - O(n) – split method

Space – o(1)

## Longest Duplicate Substring

### Problem

Given a string S, consider all *duplicated substrings*: (contiguous) substrings of S that occur 2 or more times.  (The occurrences may overlap.)

Return **any** duplicated substring that has the longest possible length.  (If S does not have a duplicated substring, the answer is "".)

**Example 1:**

**Input:** "banana"

**Output:** "ana"

**Example 2:**

**Input:** "abcd"

**Output:** ""

**Note:**

1. 2 <= S.length <= 10^5
2. S consists of lowercase English letters.

### Reference

LEETCODE, STRING, GITHUB, (LongestRepeatedSubString OR LongestDuplicateSubString OR longestCommonSubString)

### Approach

TECHDOSE can also be refer

We can use rabin karp algorithm to match substring but since we don’t know what is the max length possible we will go by checking different possible lengths.

just to avoid collision we will use hash trick. so that in this algo no hash collision occurs at all.

so how to handle it -

let's take example for digits.

how 91 is not same as 19.

because it is calculated as 9 \* 10+1\* 1 = 91

and 1 \* 10+9 \* 1 = 19

why we multiply it in power of 10? because digit ranges from 0-9.

so for alphabet ranges from a-z(i.e. 26 alphabets). we can simply use power of 26.

so for aba and aab String. result will be

a \* 26^2 + b \* 26^1 + a

a \* 26^2 + a \* 26^1 + b

both of them will give different hash values and hence no collision.

and since they can go out of range we will use long data type for hash value.

now we will use binary search to avoid un-necessary comparison.

for e.g. -> for String banana length is 6. so we will first try match for length 3 if we found result we will got towards right i.e. from 4 to 6 again. else go left from 0 to 2.

we will repeat the process till start<=end.

### Solution

**public** **class** LongestRepeatedSubString {

**public** String longestDupSubstring(String s) {

**int** start = 1;

**int** end = s.length();

String max = "";

**while** (start <= end) {

**int** mid = end - (end - start) / 2;

String str = rabinKarp(s, mid);

**if** (str == **null**) {

end = mid - 1;

} **else** {

max = str;

start = mid + 1;

}

}

**return** max;

}

**private** String rabinKarp(String s, **int** len) {

Set<Long> set = **new** HashSet<>();

**long** h = hash(s.substring(0, len));

set.add(h);

**long** pow = 1;

**for** (**int** l = 1; l < len; l++) {

pow \*= 26;

}

**for** (**int** i = 1; i <= s.length() - len; i++) {

h = nextHash(pow, h, s.charAt(i - 1), s.charAt(i + len - 1));

**if** (!set.add(h)) {

**return** s.substring(i, len + i);

}

}

**return** **null**;

}

**private** **long** nextHash(**long** pow, **long** hash, **char** left, **char** right) {

**return** (hash - left \* pow) \* 26 + right;

}

**private** **long** hash(String s) {

**long** hash = 0;

**long** mul = 1;

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

hash += s.charAt(i) \* mul;

mul \*= 26;

}

**return** hash;

}

}

### Time and space complexity

Time - O(nlogn)

Space – o(nlogn)

## Print all permutations of a String

### Problem

Given a string S, return all possible permutations of S in any order.

**Example 1:**

**Input:** "abc"

**Output: [**"abc",”acb”,”bac”,”bca”,”cab”,”cba”]

### Reference

String, GeeksForGeeks

### Approach

1. Similar to print array permutation. Here we will take two String str and prefix.
2. Keep on subtracting ith character from str and add it to the prefix and call it again.
3. Base case would be when str becomes empty and in such case just print the prefix.

### Solution

**public** List<String> permute(String str) {

List<String> list = **new** ArrayList<>();

permuteString(str, "", list);

**return** list;

}

**private** **void** permuteString(String str, String prefix, List<String> list) {

**if** (str.length() == 0) {

list.add(prefix);

**return**;

}

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

**char** ch = str.charAt(i); // get ith character

String res = str.substring(0, i) + str.substring(i + 1); // get remaining string

permuteString(res, prefix + ch, list);

}

}

### Time and space complexity

Time - O(n!)

Space – o(n!)

## Repeated Substring Pattern

### Problem

Given a non-empty string check if it can be constructed by taking a substring of it and appending multiple copies of the substring together. You may assume the given string consists of lowercase English letters only and its length will not exceed 10000.

**Example 1:**

**Input:** "abab"

**Output:** True

**Explanation:** It's the substring "ab" twice.

**Example 2:**

**Input:** "aba"

**Output:** False

**Example 3:**

**Input:** "abcabcabcabc"

**Output:** True

**Explanation:** It's the substring "abc" four times. (And the substring "abcabc" twice.)

### Reference

String, leetcode

### Approach

**Approach 1 -**

1. A string s of length n if repeated m times where m>1 then if we concat string twice and remove first and last character from string. The resultant string will always contain s.
2. If string does not repeat, itself in such case only. s will not be present after step 1.

**Example 1–**

S= abab

S+S = abababab => bababa

As it is visible, abab is present inside above String

**Example 2–**

S= aba

S+S = abaaba => baab

It does not contain aba in it.

**Approach 2 – (better)**

1. Instead of making s double first and then check length we can check substring for different combinations of length and verify if it causes after combining to s again.
2. So we can start by len = s.length/2 first and keep on checking for len-1 combination. For each length, we will check if by concating s with s k times can make s again. If we found any match return true else continue for one less. We will check till 1 length for match
3. One more point to note- to ignore unnecessary comparison of lengths, which cannot make as a whole s we can avoid. Such combinations are where s.length()%i is not 0. It means we cannot break length into `i` sections to make it s. if entire length cannot be divisible by sub-string length then it cannot together make it s.

### Solution

**Approach 1 –**

**public** **boolean** repeatedSubstringPattern(String s) {

**return** (s + s).substring(1, 2 \* s.length() - 1).contains(s);

}

**Approach 2 –**

**public** **boolean** repeatedSubstringPattern(String s) {

**int** len = s.length();

**for** (**int** i = len / 2; i >= 1; i--) { // check for half length first

**if** (len % i == 0) { // if not true substring of length i cannot make it to s

String res = s.substring(0, i);

**int** k = len / i; // number of times s repeats

StringBuilder sb = **new** StringBuilder();

**for** (**int** j = 0; j < k; j++) {

sb.append(res); //repeat res k times

}

**if** (s.equals(sb.toString())) { // check if original can make to sb

**return** **true**;

}

}

}

**return** **false**;

}

### Time and space complexity

Time - O(n)

Space – o(n)

## Partition Labels

### Problem

A string S of lowercase English letters is given. We want to partition this string into as many parts as possible so that each letter appears in at most one part, and return a list of integers representing the size of these parts.

**Example 1:**

**Input:** S = "ababcbacadefegdehijhklij"

**Output:** [9,7,8]

**Explanation:**

The partition is "ababcbaca", "defegde", "hijhklij".

This is a partition so that each letter appears in at most one part.

A partition like "ababcbacadefegde", "hijhklij" is incorrect, because it splits S into less parts.

**Note:**

* S will have length in range [1, 500].
* S will consist of lowercase English letters ('a' to 'z') only.

### Reference

String, leetcode, greedy

### Approach

1. We can use greedy algo to maximize partitions. But since a partition should have all the occurrences of particular character in it’s group. In such case we must know the last occurrences of every character in s.
2. We prepare map to store last indexes of each unique character first in for loop.
3. After that we take two variables start and end which will denote the partition boundaries. Initially it is 0.
4. We will iterate s again and check for each character there last occurrence. And keep on updating max boundary for end. If at any point, the current character that is considered has same last index it means we can close the partition and can start with searching new partition. i.e. start=end+1 for remaining set of elements.

Example –

For partition abcabzx. We cannot just end partition when we reach second a. this is because b present within two a has last index after a occurrence. But yes when we reach second b we can see our current last index as well as current loop variable I both are at same point. It means we have found 1 partition and can be added in list and can update start from 0 to index of z.

### Solution

**public** List<Integer> partitionLabels(String s) {

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

**if** (s == **null** || s.length() == 0)

**return** list;

**int**[] last = **new** **int**[26];

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

last[s.charAt(i) - 'a'] = i;

}

**int** start = 0;

**int** end = 0;

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

end = Math.*max*(end, last[s.charAt(i) - 'a']);

**if** (i == end) {

list.add(end - start + 1);

start = end + 1;

}

}

**return** list;

}

### Time and space complexity

Time - O(n)

Space – o(26) excluding list output

# 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)

## 2. Check If It Is a Straight Line

### Problem

You are given an array coordinates, coordinates[i] = [x, y], where [x, y] represents the coordinate of a point. Check if these points make a straight line in the XY plane.

**Example 1:**



**Input:** coordinates = [[1,2],[2,3],[3,4],[4,5],[5,6],[6,7]]

**Output:** true

**Example 2:**

****

**Input:** coordinates = [[1,1],[2,2],[3,4],[4,5],[5,6],[7,7]]

**Output:** false

### Reference

LEETCODE, MATH

### Approach

1. Straight line equation is y=mx+c. where m is slope and c is constant.
2. For two points y2=x2m+c and y1=x1m+c. m will be y2-y1/x2-x1.
3. In a straight line slope of every two coordinate will be same. So we use this property here.
4. Calculate slope for 0 and 1 coordinate and now check this slope for every coordinate. If we did not get equal slope return false else continue.
5. We can simply find slope of every coordinate with 0th coordinate.

### Solution

**public** **boolean** checkStraightLine(**int**[][] coordinates) {

**double** m = *slope*(coordinates, 1);

**for** (**int** i = 2; i < coordinates.length; i++) {

**if** (m != *slope*(coordinates, i)) {

**return** **false**;

}

}

**return** **true**;

}

**private** **double** slope(**int**[][] coordinates, **int** i) {

**int** xdif = coordinates[i][0] - coordinates[0][0];

**int** ydif = coordinates[i][1] - coordinates[0][1];

**return** ((ydif \* 1.0) / xdif);

}

### Time and space complexity

Time – o(n)

Space – o(1)

## 3. Check If It Is a Valid Perfect Square

### Problem

Given a positive integer *num*, write a function which returns True if *num* is a perfect square else False.

**Note:** **Do not** use any built-in library function such as sqrt.

**Example 1:**

**Input:** 16

**Output:** true

**Example 2:**

**Input:** 14

**Output:** false

### Reference

LEETCODE, MATH

### Approach

1. **Approach 1 –**

Run loop from i=1 to i\*i<n. after loop terminates check i\*i==n. if yes true else false

It takes RootN time.

1. **Approach 2 –**

Use binary search to check perfect square. It will take o(logn) time.

### Solution

1. **Approach 1 –**

**public** **boolean** perfectSquare(int num) {

**int** i = 1;

**for** (; i < num/i; i++);//avoid integer overflow by not using i\*i

**return** i \* i == num;

}

1. **Approach 2 –**

**public** **boolean** perfectSquareBest(int num) {

**if(**num==1) **return** true;

**int** low = 1;

**int** high = num / 2;

**int** mid;

**while** (low <= high) {

mid = low + (high - low) / 2;

**if** (mid \* mid == num) {

**return** **true**;

}

**if** (mid \* mid < num) {

low = mid + 1;

} **else** {

high = mid - 1;

}

}

**return** **false**;

}

### Time and space complexity

Time – o(logn)

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)

## 13. Delete node from linked list (without head given)

### Problem

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Given linked list -- head = [4,5,1,9], which looks like following:



**Example 1:**

**Input:** head = [4,5,1,9], node = 5

**Output:** [4,1,9]

**Explanation:** You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.

**Note:**

* The linked list will have at least two elements.
* All of the nodes' values will be unique.
* The given node will not be the tail and it will always be a valid node of the linked list.
* Do not return anything from your function.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. Since we don’t know the head. And we are sure that we always have two nodes and given node will not be the tail.
2. We can delete next node of the given node and just change value of given node to the next node before deleting.

4->5->1->9

4->1->1->9 (now delete 1 which is after highlighted 1)

4->1->9

### Solution

public void deleteNode(ListNode node) {

node.val=node.next.val;

ListNode temp =node.next;

node.next=node.next.next;

temp.next=null;

}

### Time and space complexity

Time - o(1)

Space – o(1)

# 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

ARRAYS, 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 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. Online Stock Span

### Problem

Write a class StockSpanner which collects daily price quotes for some stock, and returns the span of that stock's price for the current day.

The span of the stock's price today is defined as the maximum number of consecutive days (starting from today and going backwards) for which the price of the stock was less than or equal to today's price.

For example, if the price of a stock over the next 7 days were [100, 80, 60, 70, 60, 75, 85], then the stock spans would be [1, 1, 1, 2, 1, 4, 6].

**Example 1:**

**Input:** ["StockSpanner","next","next","next","next","next","next","next"], [[],[100],[80],[60],[70],[60],[75],[85]]

**Output:** [null,1,1,1,2,1,4,6]

**Explanation:**

First, S = StockSpanner() is initialized. Then:

S.next(100) is called and returns 1,

S.next(80) is called and returns 1,

S.next(60) is called and returns 1,

S.next(70) is called and returns 2,

S.next(60) is called and returns 1,

S.next(75) is called and returns 4,

S.next(85) is called and returns 6.

Note that (for example) S.next(75) returned 4, because the last 4 prices

(including today's price of 75) were less than or equal to today's price.

### Reference

LEETCODE, GEEKSFORGEEKS

### Approach

Approach something similar to previous question. We will take stack which will hold count and value as node object. Count will be stored in a way that it will have all the previous smaller node count including itself. And if a new element does not have any smaller element insert 1.

For every next call we will pop elements from stack till it is not empty and element of top<=newItem.

After loop terminates push (new node(index,newItem)). Return index.

### Solution

**class** StockSpanner {

**private** java.util.Deque<**int**[]> stack = **new** LinkedList<>();

**public** StockSpanner() {

}

**public** **int** next(**int** price) {

**int** tot=1;

**while**(!stack.isEmpty() && stack.peek()[1]<=price){

tot+=stack.pop()[0];

}

stack.push(**new** **int**[]{tot,price});

**return** tot;

}

}

### Time and space complexity

Time - o(n)

Space – o(n)

## 3. 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)

## 4. 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

## 5. 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)

## Counting sort (Sort Colors)

### Problem

Given an array with n objects colored red, white or blue, sort them [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm)so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

**Note:** You are not suppose to use the library's sort function for this problem.

**Example:**

**Input:** [2,0,2,1,1,0]

**Output:** [0,0,1,1,2,2]

### Reference

ARRAY, LOCAL\_GIT,LEETCODE

### Approach

1. Just count the occurrence of all the numbers in array in first pass. And store it in array of size 3 in this case (number of unique elements)

2. now loop through the occurrences and start overwriting it in original array.

### Solution

**Two pass -**

**public** **void** sortColors(**int**[] nums) {

**int**[] op =**new** **int**[3];

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

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

op[0]++;

} **else** **if**(nums[i]==1){

op[1]++;

} **else** {

op[2]++;

}

}

**int** c=0;

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

**for**(**int** j=0;j<op[i];j++){

nums[c++]=i;

}

}

}

**Single pass –**

**public** **void** sortColors(**int**[] nums) {

**int** low=0;

**int** mid=0;

**int** high=nums.length-1;

**while**(mid<=high){

**if**(nums[mid]==0){

swap(mid,low,nums);

low++;

mid++;

} **else** **if**(nums[mid]==1){

mid++;

} **else** {

swap(mid,high,nums);

high--;

}

}

}

### Time and space complexity

Time - O(n)

Space – O(3)

# 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)

To check if number is even or odd. Just perform n&1. If result is 0 number is even. This is because even number always has 0 at last digit in binary representation and if we do and operation with 1 we always get 0

For e.g. – check 6 and 5

110 101

001 001

000 001

2. and of number with 1 will give last digit of that number.

**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.In java it return two’s complement.

## 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)

## 2. Bitwise AND of Numbers Range

### Problem

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

**Example 1:**

**Input:** [5,7]

**Output:** 4

**Example 2:**

**Input:** [0,1]

**Output:** 0

### Reference

LEETCODE, GITHUB

### Approach

1. basic approach is to loop from m to n and keep on doing bitwise and. But it takes o(n) time.
2. We can do it faster just by observing some property of bit.
3. right shift by 1 will remove last digit from binary representation of number
4. left shift by 1 will add 0 after last digit in binary representation of number
5. Suppose m=16 and n=19 ->

M=10000

N =10011

Numbers are –

10000

10001

10010

10011

Result = 10000

It is clear that if we go from left to right (i.e. from MSB to LSB) for m and n only. The common digits will be in the result and remaining digit will be zero in result.

10000 and 10011 has 100 common and after that we have two digits left so we append two 0 at end. So result will be 10000.

1. So, now problem is to find common digits from left to right. We can do it by performing right shift of m and n till m and n are same. And keep counter to count number of shift done.

So, we get 100 whose decimal value is 4 and now if we do left shift of this 4 by count number of times. We get the result. As left shift will append zero to result count number of times.

### Solution

public int rangeBitwiseAnd(int m, int n) {

int c=0;

while(m!=n){

m=m>>1;

n=n>>1;

c++;

}

return m<<c;

}

### Time and space complexity

Time - O(32) constant time as a 32 bit number can be maximum shifted 32 times in while loop above

Space - O(1)

## 3. Number Complement

### Problem

Given a positive integer, output its complement number. The complement strategy is to flip the bits of its binary representation.

**Example 1:**

**Input:** 5

**Output:** 2

**Explanation:** The binary representation of 5 is 101 (no leading zero bits), and its complement is 010. So you need to output 2.

**Example 2:**

**Input:** 1

**Output:** 0

**Explanation:** The binary representation of 1 is 1 (no leading zero bits), and its complement is 0. So you need to output 0.

### Reference

LEETCODE, GITHUB,TED ED

### Approach

1^1=0

1^0=0

0^1=1

0^0=0

1. It means if we do xor by 1 to any digit we will get digit flipped. So now our aim is to find the number of digits.
2. For eg. If input is 10.

1010 ^ 1111 = 0101.

1. We know to find number of digits we can use log(n) +1. But we want to calculate no of digit in binary form. So we need log2(n).In java log base 2 function is not available. So we use log below property –

LogaB = logkB/logkA

In our case log2(n)= (logn/log2)+1

1. Now we know number of digits. In our case 1111 represent number 15. i.e. 2^4 -1. Or 2^digits -1.
2. We can multiply any number by 2 by doing left shift 1 time. To multiply by 2^digit we can do 1<<digits.
3. Below code can refer above approach.

### Solution

public int findComplement(int num) {

int digit = (int)(Math.log(num)/Math.log(2))+1;

int mask = (1<<digit)-1;

return num^mask;

}

### Time and space complexity

Time - O(1)

Space - O(1)

## 4. Counting Bits (number of 1’s)

### Problem

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

**Example 1:**

**Input:** 2

**Output:** [0,1,1]

**Example 2:**

**Input:** 5

**Output:** [0,1,1,2,1,2]

### Reference

LEETCODE, GITHUB

### Approach

1.and with any number will give last digit of the number.

2.right shift of any number will remove last digit in it's binary equivalent. ( in decimal it is divide by 2)

3.we know that for any number we have count already stored in its num/2 position. so we just need to add 1 if last digit is 1 else 0 to it.

### Solution

**public** **int**[] countBits(**int** num) {

**int**[] res = **new** **int**[num + 1];

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

res[i] = res[i >> 1] + (i & 1);

}

**return** res;

}

### Time and space complexity

Time - O(n)

Space - O(n)

## 5. Power of Two

### Problem

Given an integer (signed 32 bits), write a function to check whether it is a power of 2.

**Example 1:**

**Input:** 8

**Output:** true

**Example 2:**

**Input:** 5

**Output:** false

### Reference

LEETCODE, GITHUB

### Approach

**Approach 1 -**

1. We will calculate number of digits-1 in binary form. And then calculate 2 ^ (digit-1).
2. If both are same, it means number is power of two.
3. To calculate power of 2 we can use 1<<digit

**Approach 2 –**

1. n & n-1 will be zero only if number is power of two.

### Solution

**Approach 1 -**

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

**if** (n <= 0)

**return** **false**;

**int** digit = (**int**) (Math.*log*(n) / Math.*log*(2)); //will give no of digit-1

**return** (1 << digit) == n;

}

**Approach 2 -**

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

**if** (n <= 0)

**return** **false**;

**return** (n & n - 1) == 0;

}

### Time and space complexity

Time - O(1)

Space - O(1)

## 6. Power of Four

### Problem

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.

**Example 1:**

**Input:** 16

**Output:** true

**Example 2:**

**Input:** 5

**Output:** false

### Reference

LEETCODE, GITHUB

### Approach

1. we will use the property that a number divisble by 4 will always have even number of zeros in binary form.  
and n & n-1 will give result 0 if number is power of 2. Combining both results will give our answer.

### Solution

**public** **boolean** isPowerOfFour(**int** num) {

**if** (num <= 0)

**return** **false**;

**int** i = num & (num - 1); // check if number is power of two.

**int** digit = (**int**) (Math.*log*(num) / Math.*log*(2));

**return** i == 0 && digit % 2 == 0;

}

### Time and space complexity

Time - O(1)

Space - O(1)

## Power of Three

### Problem

Given an integer, write a function to determine if it is a power of three.

**Example 1:**

**Input:** 27

**Output:** true

**Example 2:**

**Input:** 0

**Output:** false

**Example 3:**

**Input:** 9

**Output:** true

**Example 4:**

**Input:** 45

**Output:** false

### Reference

LEETCODE, GITHUB

### Approach

1. We will calculate number of digits-1 with base 3. And then calculate 3^(digit-1). If it is equal to n passed. Answer is true.
2. This approach will work on any number. for power of x=2,3,4,……
3. Just calculate no of digit in base x. and then calculate power of x^digit and check if they are equal.

If in question it is asked to not to use in-built math function we can use alternative approach where we will use range of integer. This algorithm only works since 3 is a prime number.

1. 319 = 1162261467 is the biggest power of 3 less than Integer.MAX\_VALUE.
2. Since 3 is a prime number, the only divisors of 319 are 30, 31, 32, ... 319, which happen to all be powers of 3.
3. Return true if n is positive and 1162261467 is divisible by n

### Solution

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

**if** (n <= 0) {

**return** **false**;

}

**int** val = (**int**) Math.*round*((Math.*log*(n) / Math.*log*(3)));

**return** (**int**) Math.*round*(Math.*pow*(3, val)) == n;

}

public boolean isPowerOfThree(int n) {

return n > 0 && 1162261467 % n == 0;

}

### Time and space complexity

Time - O(1)

Space - O(1)

## Single Number II

### Problem

Given a **non-empty** array of integers, every element appears three times except for one, which appears exactly once. Find that single one.

**Note:**

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

**Example 1:**

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

**Output:** 3

**Example 2:**

**Input:** [0,1,0,1,0,1,99]

**Output:** 99

### Reference

LEETCODE, GITHUB

### Approach

**Approach 1 –**

We can use hashmap to count frequency in first loop and in second iteration we will just check key with 1 frequency.

**Approach 2 –**

**Bitwise operator- (tech dose)**

1. We can find desired unique number by constructing it bit by bit.
2. We know maximum bits possible in any number is 32.
3. Suppose array is [2,4,2,3,4,4,2]. Now we will apply below algo.
4. Set shift=1, sum=0;
5. For every bit starting from 0 to 32 we will iterate every element in nums array.
6. Calculate number of 1’s in last bit in var c.
7. If c%3==1 it means we have bit set in our unique element. So update sum=sum+shift
8. Just remove last bit also while iterating. So that in next iteration we will count set bit again.
9. After inner loop finished multiply shift by 2. This is because if we need to update answer it will be in power 2.

suppose unique number is 5 i.e. 101 so sum will be 1 + 4

**Approach 3 –**

**Bitwise operator- (tech dose)**

1. Take two variable one and two.
2. Iterate from 0 to nums.length and for each element.
3. Perform one = (one ^ nums[i]) & (~two)
4. Perform two = (two ^ nums[i]) & (~one)
5. Return one.

### Solution

**Approach 1 –**

public int singleNumber(int[] nums) {

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

for(int i:nums){

map.put(i,map.getOrDefault(i,0)+1);

}

for(int k:map.keySet()) {

if(map.get(k)==1){

return k;

}

}

return -1;

}

**Approach 2 –**

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

**int** shift = 1;

**int** sum = 0;

**for** (**int** i = 0; i < 32; i++) {

**int** c = 0;

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

**if** ((nums[j] & 1) != 0) { // & 1 to get last digit

c++; // count all 1 in last digit

}

nums[j] = nums[j] >> 1; // remove last digit from the number.

}

**if** (c % 3 == 1) { // if we have value 1. means our unique element has 1 set.

sum += shift; // sum = sum + shift\_value

}

shift = shift << 1;//multiply by 2 and will be used in creating unique element

}

**return** sum;

}

**Approach 3-**

public int singleNumber(int[] A) {

int ones = 0, twos = 0;

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

ones = (ones ^ A[i]) & ~twos;

twos = (twos ^ A[i]) & ~ones;

}

return ones;

}

### Time and space complexity

Time - O(n)

Space - O(1)

## Single Number III

### Problem

Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

**Example:**

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

**Output:** [3,5]

**Note**:

1. The order of the result is not important. So in the above example, [5, 3] is also correct.

### Reference

LEETCODE, GITHUB

### Approach

https://www.youtube.com/watch?v=3TSC0nlur58

1. First find the xor of all elements. It will give xor result of our two unique elements.

2. Since it is clear two unique elements are different. So they must have atleast 1 bit different in binary representation.

3. Our task is to divide them into groups so that one group contain first unique element and other will contain second element.

4. once we have two groups we can xor again in there respective groups and we get our unique elements.

5. Now to separate them in two groups we can use point2 where we know 1 bit must be different between a and b. so 1st grp contains all elements with lowbit set. And other don’t

6. we can use formula – res & 2’s complement of res.

7. we will find that low bit from left to right and then perform xor with array.

8. now do & operation with lowbit and if entire result is 0 make it in first grp else in second grp.

9. above approach works because ultimately xor of other duplicate elements will result in a 0 for that lowbit also in there each group.

### Solution

**public** **int**[] singleNumber3(**int**[] nums) {

**int**[] res = **new** **int**[2];

**int** xor = 0;

**for** (**int** i : nums) {

xor = xor ^ i;

}

**int** lowbit = xor & ((~xor) + 1); // finding bit set from leftmost position

**for** (**int** i : nums) {

**if** ((i & lowbit) == 0) { // divide into group. 1st will have all ith bit set

res[0] ^= i;

} **else** {

res[1] ^= i;

}

}

**return** res;

}

### 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)

## 6. 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<Integer> result = **new** LinkedList<>();

stack.push(node);

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

node = stack.pop();

result.push(node.getData());

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

stack.push(node.getLeft());

}

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

stack.push(node.getRight());

}

}

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

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

}

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();

}

**Approach 2-**

public List<List<Integer>> levelOrder(TreeNode root) {

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

if(root==null) {

return output;

}

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

queue.add(root);

while(!queue.isEmpty()) {

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

int size = queue.size();

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

TreeNode node = queue.poll();

list.add(node.val);

if(node.left!=null) {

queue.add(node.left);

}

if(node.right!=null) {

queue.add(node.right);

}

}

output.add(list);

}

return output;

}

### Time and space complexity

O(n)

O(n)

## 8. 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)

## Path Sum

### Problem

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

**Note:** A leaf is a node with no children.

**Example:**

Given the below binary tree and sum = 22,

**5**

**/** \

**4** 8

**/** / \

**11** 13 4

/ **\** \

7 **2** 1

return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

### Reference

LEETCODE, GEEKSFORGEEKS, GITHUB

### Approach

1. Will use recursive solution. We keep on subtracting sum variable and will check in recursive call if sum reaches 0 value and node is leaf node. If yes return true. If node reaches null itself return false.

### Solution

public boolean hasPathSum(TreeNode root, int sum) {

if(root==null){

return false;

}

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

return sum==root.val;

}

return hasPathSum(root.left,sum-root.val)||hasPathSum(root.right,sum-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. Binary Tree Paths (root to leaf path)

### Problem

Given a binary tree, return all root-to-leaf paths.

**Note:** A leaf is a node with no children.

**Example:**

**Input:**

1

/ \

2 3

\

5

**Output:** ["1->2->5", "1->3"]

**Explanation:** All root-to-leaf paths are: 1->2->5, 1->3

### Reference

LEETCODE, GITHUB

### Approach

1. Perform recursive call and keep appending nodes to StringBuilder till we reach leaf.
2. Add node to sb and note down the length. We will use it later while back tracking and removing extra childs.
3. If current node if leaf print sb.
4. Else call for left child and append ->.
5. After call stack finishes we need to remove extra added nodes. So we use delete method of sb which takes start index as len which has value till current node and end index as current sb length.
6. Now again call for right child by appending ->.

### Solution

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

*printRootToLeafPaths*(root, **new** StringBuilder());

}

**private** **void** printRootToLeafPaths(TreeNode<Integer> root, StringBuilder sb) {

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

sb.append(root.getData());

**int** len = sb.length();

**if** (root.getLeft() == **null** && root.getRight() == **null**) {

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

} **else** {

*printRootToLeafPaths*(root.getLeft(), sb.append("->"));

sb.delete(len, sb.length());

*printRootToLeafPaths*(root.getRight(), sb.append("->"));

}

}

}

### Time and space complexity

O(n2)

O(n)

## 16. 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)

## 17. 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)

## 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)

## 21. Construct Binary Tree from given post-order and in-order

### Problem

Return the root node of a binary tree that matches the given post-order and in-order traversal.

Input :

in[] = {4, 8, 2, 5, 1, 6, 3, 7}

post[] = {8, 4, 5, 2, 6, 7, 3, 1}

Output : Root of below tree

1

/ \

2 3

/ \ / \

4 5 6 7

\

8

### Reference

LEETCODE, BINARY TREE, GEEKSFORGEEKS, Recursive

### Approach

Let us see the process of constructing tree from in[] = {4, 8, 2, 5, 1, 6, 3, 7} and post[] = {8, 4, 5, 2, 6, 7, 3, 1}

**1)** We first find the last node in post[]. The last node is “1”, we know this value is root as root always appear in the end of postorder traversal.

**2)** We search “1” in in[] to find left and right subtrees of root. Everything on left of “1” in in[] is in left subtree and everything on right is in right subtree.

1

/ \

[4, 8, 2, 5] [6, 3, 7]

**3)** We recur the above process for following two.  
….**b)** Recur for in[] = {6, 3, 7} and post[] = {6, 7, 3}  
…….Make the created tree as right child of root.  
….**a)** Recur for in[] = {4, 8, 2, 5} and post[] = {8, 4, 5, 2}.  
…….Make the created tree as left child of root.

### Solution

public TreeNode buildTree(int[] inorder, int[] post) {

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

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

map.put(inorder[i],i);

}

int[] ar = new int[1];

ar[0]=post.length-1;

return bt(post,0,post.length-1,ar,map);

}

private TreeNode bt(int[] post,int l,int r,int[] c, Map<Integer,Integer> map) {

if(l>r) {

return null;

}

TreeNode node = new TreeNode(post[c[0]]);

c[0]-=1;

int mid = map.get(node.val);

node.right=bt(post,mid+1,r,c,map);

node.left=bt(post,l,mid-1,c,map);

return node;

}

### Time and space complexity

O(n) if hashmap is used else additional logn time to search element using binary search.

O(n)

## 22. Identical Structure of Two Tree (ISOMORPHIC)

### Problem

Given two binary trees. The task is to write a program to check if the two trees are identical in structure.



In the above figure both of the trees, Tree1 and Tree2 are identical in structure. That is, they have the same structure.

**Note**: This problem is different from [Check if two trees are identical](https://www.geeksforgeeks.org/write-c-code-to-determine-if-two-trees-are-identical/) as here we need to compare only the structures of the two trees and not the values at their nodes.

### Reference

BINARY TREE, GEEKSFORGEEKS, Recursive

### Approach

1. Just check if we have node present at one of the tree only return false else check for rest of the tree. If both tree got empty return true.

### Solution

**public** **boolean** identical(TreeNode<Integer> root1, TreeNode<Integer> root2) {

**if** (root1 == **null** && root2 == **null**) {

**return** **true**;

}

**if** (root2 == **null** || root1 == **null**) {

**return** **false**;

}

**return** *identical*(root1.getLeft(), root2.getLeft()) && *identical*(root1.getRight(), root2.getRight());

}

### Time and space complexity

O(n)

O(n)

## 23. Symmetric Tree (Mirror)

### Problem

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree [1,2,2,3,4,4,3] is symmetric:

1

/ \

2 2

/ \ / \

3 4 4 3

But the following [1,2,2,null,3,null,3] is not:

1

/ \

2 2

\ \

3 3

### Reference

LEETCODE, BINARY TREE, GEEKSFORGEEKS, Recursive

### Approach

1. It is just like finding if two tree are mirror of each other.
2. We just pass root.left and root.right in mirror function and verify if nodes are equal. If not equal or symmetric return false. It’s algo is same as finding structure is identical. Just we add extra check to verify value.

### Solution

**Recursive-**

**public** **boolean** mirror(TreeNode<Integer> root1, TreeNode<Integer> root2) {

**if** (root1 == **null** && root2 == **null**) {

**return** **true**;

}

**if** (root1 == **null** || root2 == **null**) {

**return** **false**;

}

**if** (!root1.getData().equals(root2.getData())) {

**return** **false**;

}

**return** *mirror*(root1.getLeft(), root2.getRight()) && *mirror*(root1.getRight(), root2.getLeft());

}

**Iterative-**

public boolean isSymmetric(TreeNode root) {

if(root==null){

return true;

}

return mirror(root.left,root.right);

}

public boolean mirror(TreeNode root1,TreeNode root2) {

Queue<TreeNode> queue1 = new LinkedList<>();

queue1.add(root1);

queue1.add(root2);

while(!queue1.isEmpty()) {

int size1 = queue1.size();

for(int i=0;i<size1/2;i++) {

TreeNode node1 = queue1.poll();

TreeNode node2 = queue1.poll();

if((node1!=null && node2==null) || (node2!=null && node1==null)){

return false;

}

if((node1!=null && node2!=null)){

if(node1.val!=node2.val){

return false;

}

queue1.add(node1.left);

queue1.add(node2.right);

queue1.add(node1.right);

queue1.add(node2.left);

}

}

}

return true;

}

### Time and space complexity

O(n)

O(n)

## 24. Populating Next Right Pointers in Each Node(Next Sibling)

### Problem

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

struct Node {

int val;

Node \*left;

Node \*right;

Node \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.



### Reference

LEETCODE, BINARY TREE, ITERATIVE,YOUTUBE

### Approach

**Approach** 1- Level order approach and keep setting next with in inner for loop.

**Approach** 2- Queue take o(n) space. It can be done in o(1) space also.

1. We will go level by level and keep updating next pointer of every node in that level.
2. We start with the leftmost node at that level and update node.left.next=node.right. We check if node has right child and also node.next.left is also present. Then update node.right.next=node.next.left. Since we want to cover nodes in same level of different parent also. update node=node.right to handle it’s child also.
3. Once entire level traversed we will move to next level by setting currnode=currnode.left;

### Solution

**Approach** 1 –

public Node connect(Node root) {

if(root==null){

return null;

}

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

queue.add(root);

while(!queue.isEmpty()){

int size = queue.size();

Node prev=null;

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

Node node = queue.poll();

if(prev!=null){

prev.next=node;

}

prev=node;

if(node.left!=null){

queue.add(node.left);

}

if(node.right!=null){

queue.add(node.right);

}

}

}

return root;

}

**Approach** 2-

public Node connect(Node root) {

if(root==null){

return null;

}

Node curr=root;

Node levelRoot = null;

while(curr!=null) {

levelRoot = curr;

while(levelRoot!=null) {

if(levelRoot.left!=null) {

levelRoot.left.next=levelRoot.right;

}

if(levelRoot.right!=null && levelRoot.next!=null) {

levelRoot.right.next= levelRoot.next.left;

}

levelRoot=levelRoot.next;

}

curr=curr.left;

}

return root;

}

### Time and space complexity

O(n)

O(n)

## 25. Populating Next Right Pointers in Each Node II

### Problem

Given a binary tree

struct Node {

int val;

Node \*left;

Node \*right;

Node \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

**Example 1:**



**Input:** root = [1,2,3,4,5,null,7]

**Output:** [1,#,2,3,#,4,5,7,#]

**Explanation:** Given the above binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

### Reference

LEETCODE, BINARY TREE, ITERATIVE,GEEKSFORGEEKS

### Approach

1. If we follow level order approach we need extra space for queue. It can be done in o(1) space.
2. We will follow approach as mentioned in previous question. Since it is not complete tree so it might be possible that right child may not be present but node exists in same level. So, in such case we will call utility method which will return next sibling if exists.

### Solution

**public** **void** nextSiblingWithoutQueue(TreeNodeModified root) {

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

**return**;

}

TreeNodeModified levelRoot;

TreeNodeModified currRoot = root;

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

levelRoot = currRoot;

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

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

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

levelRoot.getLeft().setNextSibling(levelRoot.getRight());

} **else** {

levelRoot.getLeft().setNextSibling(*getNextSibling*(levelRoot));

}

}

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

levelRoot.getRight().setNextSibling(*getNextSibling*(levelRoot));

}

levelRoot = levelRoot.getNextSibling();

}

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

currRoot = currRoot.getLeft();

} **else** **if** (currRoot.getRight() != **null**) {

currRoot = currRoot.getRight();

} **else** {

currRoot = *getNextSibling*(currRoot);

}

}

}

**private** **static** TreeNodeModified getNextSibling(TreeNodeModified levelRoot) {

TreeNodeModified temp = levelRoot.getNextSibling();

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

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

**return** temp.getLeft();

}

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

**return** temp.getRight();

}

temp = temp.getNextSibling();

}

**return** temp;

}

### Time and space complexity

O(n)

O(1)

## 26. Lowest Common Ancestor

### Problem

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

Given the following binary tree:  root = [3,5,1,6,2,0,8,null,null,7,4]



**Example 1:**

**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

**Output:** 3

**Explanation:** The LCA of nodes 5 and 1 is 3.

**Example 2:**

**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

**Output:** 5

**Explanation:** The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

**Note:**

* All of the nodes' values will be unique.
* p and q are different and both values will exist in the binary tree.

### Reference

LEETCODE, BINARY TREE, GITHUB,GEEKSFORGEEKS

### Approach

1. call recursively and check if current node matches item1 or item2 if matched return node. Else call for left and right tree.
2. We know that if any of item match we don’t need to check it’s subtree. Just try to find in remaining tree . if node found in other half of tree then current node is lca. And if we did not found any node in remaining tree. It means lca must be present in it’s subtree and hence we directly return it.

### Solution

**public** TreeNode<Integer> lowestCommonAncestor(TreeNode<Integer> root, **int** item1, **int** item2) {

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

**return** **null**;

}

**if** (root.getData() == item1 || root.getData() == item2) {

**return** root;

}

TreeNode<Integer> left = *lowestCommonAncestor*(root.getLeft(), item1, item2);

TreeNode<Integer> right = *lowestCommonAncestor*(root.getRight(), item1, item2);

**if** (left != **null** && right != **null**) {

**return** root;

}

**return** left != **null** ? left : right;

}

### Time and space complexity

O(n)

O(n)

## Invert Binary Tree

### Problem

Invert a binary tree.

**Example:**

Input:

4

/ \

2 7

/ \ / \

1 3 6 9

Output:

4

/ \

7 2

/ \ / \

9 6 3 1

### Reference

LEETCODE, BINARY TREE, GITHUB

### Approach

1. It is just like mirror of method. We just swap the left and right node of current node and call method again for left and right child.

### Solution

public TreeNode invertTree(TreeNode root) {

if(root==null){

return null;

}

TreeNode temp = root.left;

root.left=root.right;

root.right=temp;

invertTree(root.left);

invertTree(root.right);

return root;

}

public TreeNode invertTree(TreeNode root) {

if(root==null){

return root;

}

TreeNode left = invertTree(root.left);

TreeNode right = invertTree(root.right);

root.left=right;

root.right=left;

return root;

}

### Time and space complexity

O(n)

O(n)

## 28. 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)

## 29. Find Maximum Node in a binary tree

### Problem

Given a Binary Node find maximum node.

### Reference

Recursive, GITHUB

### Approach

1. Check if current node is max out of two childs And return max out of three.
2. keep on repeating for all child nodes.

### 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)

## 30. Find Maximum Width of a binary tree

### Problem

Given a binary tree, write a function to get the maximum width of the given tree. Width of a tree is maximum of widths of all levels.

Let us consider the below example tree.

1

/ \

2 3

/ \ \

4 5 8

/ \

6 7

For the above tree,  
width of level 1 is 1,  
width of level 2 is 2,  
width of level 3 is 3  
width of level 4 is 2.

So the maximum width of the tree is 3.

### Reference

LEETCODE, GITHUB

### Approach

1. Perform level order traversal and keep track of max node count across each level.

### Solution

**public** **static** **int** maximumWidth(TreeNode<Integer> root) {

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

**return** 0;

}

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

queue.add(root);

**int** max = 0;

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

**int** size = queue.size();

**if** (size > max) {

max = size;

}

**while** (size-- > 0) {

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

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

queue.add(node.getLeft());

}

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

queue.add(node.getRight());

}

}

}

**return** max;

}

### Time and space complexity

O(n)

O(n)

## 31. Count Complete Tree Nodes

### Problem

Given a **complete** binary tree, count the number of nodes.

**Note:**

**Definition of a complete binary tree from**[**Wikipedia**](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees)**:**  
In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

**Example:**

**Input:**

1

/ \

2 3

/ \ /

4 5 6

**Output:** 6

### Reference

LEETCODE, GITHUB

### Approach

**Approach 1 -**

Just count nodes. but that takes o(n) time.

**Approach 2 -**

1. A fully completed tree has node number-> count = 2 ^ depth - 1  
for example: [1,4,5]  
depth is 2  
count = 2 ^ 2 - 1 = 3  
2. Compare left height and right height, if equal, use the formula, otherwise recursively search left and right at next level  
3. The search pattern is very similar to binary search, the difference of heights either exists in left side or right side  
4. Due to the reason stated in point 3, the time complexity is h ^ 2, there is h times for each level, and h times for calculating height at each level.  
5. To avoid using power function left shift operator is used 1<<h is equal to pow(2,h)

Hence it takes logn\*logn (as it is a complete binary tree)

### Solution

**public** **int** countNodes(TreeNode root) {

**int** left = height(root, **true**);

**int** right = height(root, **false**);

**if** (left == right) {

**return** (1 << left) - 1;

}

**return** countNodes(root.left) + countNodes(root.right) + 1;

}

**private** **int** height(TreeNode root, **boolean** isLeft) {

**int** h = 0;

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

root = isLeft ? root.left : root.right;

h++;

}

**return** h;

}

### Time and space complexity

O(logn\*logn)

O(logn)

# 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 only pre-order or postorder given we can construct a unique bst.
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. We can also use same algo to construct unique binary tree using pre-order and in-order.
2. 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 buildTree(int[] preorder, int[] inorder) {

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

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

map.put(inorder[i],i);

}

return bt(preorder,0,preorder.length-1,new int[1],map);

}

private TreeNode bt(int[] pre,int l,int r,int[] c, Map<Integer,Integer> map){

if(l>r) {

return null;

}

TreeNode node = new TreeNode(pre[c[0]]);

c[0]+=1;

int mid = map.get(node.val);

node.left=bt(pre,l,mid-1,c,map);

node.right=bt(pre,mid+1,r,c,map);

return node;

}

### 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)

## 5. Construct Binary Search Tree from given post-order

### Problem

**Return the root node of a binary search tree that matches the given postorder traversal.**

**Example 1:**

**Input:** [1, 7, 5, 50, 40, 10]

**Output:**

10

/ \

5 40

/ \ \

1 7 50

### Reference

BINARY SEARCH TREE, BST, Recursive

### Approach

1. We know that last node of postorder will always be root. Will follow same logic as done in previous problem. Just start counter from last. And first set right child and then left child. We call right child first because postorder follows - LRN

### Solution

**public** **static** TreeNode<Integer> getBstFromPostOrder(**int**[] post) {

**int**[] ar = **new** **int**[1];

ar[0] = post.length - 1;

**return** *getBstFromPostOrder*(post, Integer.***MIN\_VALUE***, Integer.***MAX\_VALUE***, ar);

}

**private** **static** TreeNode<Integer> getBstFromPostOrder(**int**[] post, **int** minValue, **int** maxValue, **int**[] c) {

**if** (c[0] < 0) {

**return** **null**;

}

**int** key = post[c[0]];

TreeNode<Integer> node = **null**;

**if** (key > minValue && key < maxValue) {

node = **new** TreeNode<>(key);

c[0] -= 1;

node.setRight(*getBstFromPostOrder*(post, key, maxValue, c));

node.setLeft(*getBstFromPostOrder*(post, minValue, key, c));

}

**return** node;

}

### Time and space complexity

O(n)

O(n)

## Check if Binary tree is Binary Search Tree

### Problem

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

* The left subtree of a node contains only nodes with keys **less than** the node's key.
* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.

**Example 1:**

2

/ \

1 3

**Input:** [2,1,3]

**Output:** true

**Example 2:**

5

/ \

1 4

  / \

  3 6

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

**Output:** false

**Explanation:** The root node's value is 5 but its right child's value is 4.

### Reference

BINARY TREE, BST, Recursive,GEEKSFORGEEKS

### Approach

1. To check always check if node is in between min amd max if it is outside the range return false.
2. For root, Integer.MIN\_VALUE and Integer.MAX\_VALUE . for left child valid range will be min,root.data-1 and for right child root.data+1,max. so we keep on recursively calling for left and right child and check narrowed range

### Solution

**public** **boolean** isBinarySearchTree(TreeNode<Integer> root) {

**return** *isBST*(root, Integer.***MIN\_VALUE***, Integer.***MAX\_VALUE***);

}

**private** **boolean** isBST(TreeNode<Integer> root, **int** min, **int** max) {

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

**return** **true**;

}

**if** (root.getData() < min || root.getData() > max) {

**return** **false**;

}

**return** *isBST*(root.getLeft(), min, root.getData() - 1) && *isBST*(root.getRight(), root.getData() + 1, max);

}

### Time and space complexity

O(n)

O(n)

## 7. Lowest Common Ancestor

### Problem

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

Given binary search tree:  root = [6,2,8,0,4,7,9,null,null,3,5]



**Example 1:**

**Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

**Output:** 6

**Explanation:** The LCA of nodes 2 and 8 is 6.

**Example 2:**

**Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

**Output:** 2

**Explanation:** The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

**Note:**

* All of the nodes' values will be unique.
* p and q are different and both values will exist in the BST.

### Reference

LEETCODE, BINARY SEARCH TREE, GITHUB, ITERATIVE

### Approach

Iterative and recursive both can work. Since binary search tree holds property we are sure if p and q both less than parent node search left and similarly if p and q greater then parent search right child. If any of above condition does not meet it means. p and q are present in left and right child both. So current node is the lca node.

### Solution

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if(p.val<root.val && q.val<root.val) {

return lowestCommonAncestor(root.left,p,q);

}

if(p.val>root.val && q.val>root.val) {

return lowestCommonAncestor(root.right,p,q);

}

return root;

}

**Iterative -**

**public** TreeNode<Integer> lowestCommonAncestor(TreeNode<Integer> root, Integer node1, Integer node2) {

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

**if** (root.getData() > node1 && root.getData() > node2) {

root = root.getLeft();

} **else** **if** (root.getData() < node1 && root.getData() < node2) {

root = root.getRight();

} **else** {

**break**;

}

}

**return** root;

}

### Time and space complexity

O(n)

O(1) – iterative.

## 8. Kth smallest element in binary search tree

### Problem

Given a binary search tree, write a function kthSmallest to find the **k**th smallest element in it.

**Note:**  
You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

**Example 1:**

**Input:** root = [3,1,4,null,2], k = 1

3

/ \

1 4

\

  2

**Output:** 1

**Example 2:**

**Input:** root = [5,3,6,2,4,null,null,1], k = 3

5

/ \

3 6

/ \

2 4

/

1

**Output:** 3

### Reference

LEETCODE, BINARY Search TREE, GITHUB

### Approach

1. We will follow in-order traversal and as soon as we find kth element we return that node value. And return same node value from each recursive call stack. and will not call any further methods.
2. We can also use heap to find kth smallest or largest.

### Solution

public int kthSmallest(TreeNode root, int k) {

return in(root,new int[]{k});

}

private int in(TreeNode root,int[] k) {

if(root==null) {

return -1;

}

int res = in(root.left,k);

if(res!=-1){

return res;

}

if(--k[0]==0){

return root.val;

}

return in(root.right,k);

}

### Time and space complexity

O(n)

O(n)

## 9. Range Sum of BST

### Problem

Given the root node of a binary search tree, return the sum of values of all nodes with value between L and R (inclusive).

The binary search tree is guaranteed to have unique values.

**Example 1:**

**Input:** root = [10,5,15,3,7,null,18], L = 7, R = 15

**Output:** 32

**Example 2:**

**Input:** root = [10,5,15,3,7,13,18,1,null,6], L = 6, R = 10

**Output:** 23

### Reference

LEETCODE, BINARY Search TREE, GITHUB

### Approach

1. It is just like adding nodes in binary tree. Just we add node.val if it is in range l and r else we add 0.
2. But since it is BST we can totally skip calling entire left call of the tree in case left > root.val and vice-versa.

### Solution

**public** **int** rangeSumBST(TreeNode root, **int** L, **int** R) {

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

**return** 0;

}

**if**(L<=root.val && root.val<=R) {

**return** rangeSumBST(root.left,L,R)+rangeSumBST(root.right,L,R)+root.val;

}

**if**(L>root.val) {

**return** rangeSumBST(root.right,L,R);

}

**return** rangeSumBST(root.left,L,R);

}

### 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(1)

## Maximum Sum in Circular Contiguous Subarray

### Problem

Given a **circular array** **C** of integers represented by A, find the maximum possible sum of a non-empty subarray of **C**.

Here, a circular array means the end of the array connects to the beginning of the array.  (Formally, C[i] = A[i] when 0 <= i < A.length, and C[i+A.length] = C[i] when i >= 0.)

Also, a subarray may only include each element of the fixed buffer A at most once.  (Formally, for a subarray C[i], C[i+1], ..., C[j], there does not exist i <= k1, k2 <= j with k1 % A.length = k2 % A.length.)

**Example 1:**

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

**Output:** 3

**Explanation:** Subarray [3] has maximum sum 3

**Example 2:**

**Input:** [5,-3,5]

**Output:** 10

**Explanation:** Subarray [5,5] has maximum sum 5 + 5 = 10

**Example 3:**

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

**Output:** 4

**Explanation:** Subarray [2,-1,3] has maximum sum 2 + (-1) + 3 = 4

**Example 4:**

**Input:** [3,-2,2,-3]

**Output:** 3

**Explanation:** Subarray [3] and [3,-2,2] both have maximum sum 3

**Example 5:**

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

**Output:** -1

**Explanation:** Subarray [-1] has maximum sum -1

### Reference

LEETCODE, UDEMY, DP, YOUTUBE, TECH DOSE

### Approach

1. Just use same algorithm like above question. By using normal kadane algo we will get max sum from with-in array. But for second case it might be possible that we get max sum starting from end and to start (circular manner).
2. For e.g. –

[5, -1, -2, 3, 2, 6]

In such case max goes from 3 till 5 i.e. 16. So, one way can be to find global\_min and total sum of array.

And just check Max of(global\_max, sum - global\_min). In above example – 11,13-(-3) = max(11,16).

1. Above approach is full proof because if we eliminate contiguous min we can get max for second case.
2. But for boundary case where all element are negatives. Global\_Max will be having single max element. So return it directly.

### Solution

**public** **int** maxSubarraySumCircular(**int**[] arr) {

**int** global\_max = arr[0];

**int** global\_min = arr[0];

**int** curr\_max = arr[0];

**int** curr\_min = arr[0];

**int** sum = arr[0];

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

curr\_max = MathUtil.*max*(curr\_max + arr[i], arr[i]);

curr\_min = MathUtil.*min*(curr\_min + arr[i], arr[i]);

global\_max = MathUtil.*max*(global\_max, curr\_max);

global\_min = MathUtil.*min*(global\_min, curr\_min);

sum += arr[i];

}

**return** global\_max > 0 ? MathUtil.*max*(sum - global\_min, global\_max) : global\_max;

}

### Time and space complexity

O(n)

O(1)

## Count Square Submatrices with All Ones

### Problem

Given a m \* n matrix of ones and zeros, return how many **square** submatrices have all ones.

**Example 1:**

**Input:** matrix =

[

  [0,1,1,1],

  [1,1,1,1],

  [0,1,1,1]

]

**Output:** 15

**Explanation:**

There are **10** squares of side 1.

There are **4** squares of side 2.

There is **1** square of side 3.

Total number of squares = 10 + 4 + 1 = **15**.

**Example 2:**

**Input:** matrix =

[

[1,0,1],

[1,1,0],

[1,1,0]

]

**Output:** 7

**Explanation:**

There are **6** squares of side 1.

There is **1** square of side 2.

Total number of squares = 6 + 1 = **7**.

**Constraints:**

* 1 <= arr.length <= 300
* 1 <= arr[0].length <= 300
* 0 <= arr[i][j] <= 1

### Reference

LEETCODE, DP

### Approach

TODO

### Solution

**public** **int** countSquares(**int**[][] matrix) {

**int** c = 0;

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

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

**if** (i != 0 && j != 0 && matrix[i][j] != 0) {

matrix[i][j] = Math.*min*(matrix[i - 1][j - 1], Math.*min*(matrix[i - 1][j], matrix[i][j - 1])) + 1;

}

c += matrix[i][j];

}

}

**return** c;

}

### Time and space complexity

O(nm)

O(1)

## Uncrossed Lines

### Problem

We write the integers of A and B (in the order they are given) on two separate horizontal lines.

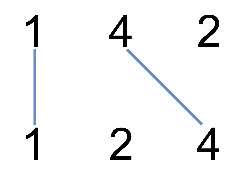
Now, we may draw *connecting lines*: a straight line connecting two numbers A[i] and B[j] such that:

* A[i] == B[j];
* The line we draw does not intersect any other connecting (non-horizontal) line.

Note that a connecting lines cannot intersect even at the endpoints: each number can only belong to one connecting line.

Return the maximum number of connecting lines we can draw in this way.

**Example 1:**



**Input:** A = [1,4,2], B = [1,2,4]

**Output:** 2

**Explanation:** We can draw 2 uncrossed lines as in the diagram.

We cannot draw 3 uncrossed lines, because the line from A[1]=4 to B[2]=4 will intersect the line from A[2]=2 to B[1]=2.

**Example 2:**

**Input:** A = [2,5,1,2,5], B = [10,5,2,1,5,2]

**Output:** 3

**Example 3:**

**Input:** A = [1,3,7,1,7,5], B = [1,9,2,5,1]

**Output:** 2

**Note:**

1. 1 <= A.length <= 500
2. 1 <= B.length <= 500
3. 1 <= A[i], B[i] <= 2000

### Reference

LEETCODE, DP

### Approach

1. It is just like LCS problem.
2. We will keep 2d array of size row+1 and column +1
3. And iterate loop and check if arr[i-1] and brr[j-1] equal if yes add one to i-1 and j-1 element. As in case of match we know we can draw straight line. Hence we add 1 to previous score.
4. Else take maximum of [i-1,j] and [I,j-1].
5. Once loop terminates table[r][c] will have maximum answer.

### Solution

**public** **int** maxUncrossedLines(**int**[] arr, **int**[] brr) {

**int**[][] table = **new** **int**[arr.length + 1][brr.length + 1];

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

**for** (**int** j = 1; j <= brr.length; j++) {

**if** (arr[i - 1] == brr[j - 1]) {

table[i][j] = table[i - 1][j - 1] + 1;

} **else** {

table[i][j] = Math.*max*(table[i - 1][j], table[i][j - 1]);

}

}

}

**return** table[arr.length][brr.length];

}

### Time and space complexity

O(nm)

O(nm)

## 5. Edit Distance

### Problem

Given two words *word1* and *word2*, find the minimum number of operations required to convert *word1* to *word2*.

You have the following 3 operations permitted on a word:

1. Insert a character
2. Delete a character
3. Replace a character

**Example 1:**

**Input:** word1 = "horse", word2 = "ros"

**Output:** 3

**Explanation:**

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

**Example 2:**

**Input:** word1 = "intention", word2 = "execution"

**Output:** 5

**Explanation:**

intention -> inention (remove 't')

inention -> enention (replace 'i' with 'e')

enention -> exention (replace 'n' with 'x')

exention -> exection (replace 'n' with 'c')

exection -> execution (insert 'u')

### Reference

LEETCODE, DP, Tech dosee

### Approach

**Thought process:**  
Given two strings, we're tasked with finding the minimum number of transformations we need to make to arrive with equivalent strings. From the get-go, there doesn't seem to be any way around trying all possibilities, and in this, possibilities refers to inserting, deleting, or replacing a character. Recursion is usually a good choice for trying all possilbilities.

Whenever we write recursive functions, we'll need some way to terminate, or else we'll end up overflowing the stack via infinite recursion. With strings, the natural state to keep track of is the index. We'll need two indexes, one for word1 and one for word2. Now we just need to handle our base cases, and recursive cases.  
What happens when we're done with either word? Some thought will tell you that the minimum number of transformations is simply to insert the rest of the other word. This is our base case. What about when we're not done with either string? We'll either match the currently indexed characters in both strings, or mismatch. In the first case, we don't incur any penalty, and we can continue to compare the rest of the strings by recursing on the rest of both strings. In the case of a mismatch, we either insert, delete, or replace. To recap:

1. base case: word1 = "" or word2 = "" => return length of other string
2. recursive case: word1[0] == word2[0] => recurse on word1[1:] and word2[1:]
3. recursive case: word1[0] != word2[0] => recurse by inserting, deleting, or replacing

TODO

### Solution

**public** **int** minDistance(String word1, String word2) {

**int** m = word1.length();

**int** n = word2.length();

**int**[][] dp = **new** **int**[m + 1][n + 1];

**for** (**int** i = 1; i <= m; i++) {

dp[i][0] = i;

}

**for** (**int** i = 1; i <= n; i++) {

dp[0][i] = i;

}

**for** (**int** i = 1; i <= m; i++) {

**for** (**int** j = 1; j <= n; j++) {

**if** (word1.charAt(i - 1) == word2.charAt(j - 1)) {

dp[i][j] = dp[i - 1][j - 1];

} **else** {

dp[i][j] = Math.*min*(dp[i - 1][j - 1], Math.*min*(dp[i - 1][j], dp[i][j - 1])) + 1;

}

}

}

**return** dp[m][n];

}

### Time and space complexity

O(nm)

O(nm)

# 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

## 2. Insert Delete GetRandom O(1)

### Problem

Design a data structure that supports all following operations in *average* **O(1)** time.

1. insert(val): Inserts an item val to the set if not already present.
2. remove(val): Removes an item val from the set if present.
3. getRandom: Returns a random element from current set of elements. Each element must have the **same probability** of being returned.

**Example:**

// Init an empty set.

RandomizedSet randomSet = new RandomizedSet();

// Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomSet.insert(1);

// Returns false as 2 does not exist in the set.

randomSet.remove(2);

// Inserts 2 to the set, returns true. Set now contains [1,2].

randomSet.insert(2);

// getRandom should return either 1 or 2 randomly.

randomSet.getRandom();

// Removes 1 from the set, returns true. Set now contains [2].

randomSet.remove(1);

// 2 was already in the set, so return false.

randomSet.insert(2);

// Since 2 is the only number in the set, getRandom always return 2.

randomSet.getRandom();

### Reference

LEETCODE, GIT

### Approach

1. to get randomly any value from collection we need arraylist as it can get element from index which is generated by random.nextInt.
2. we will also use map to store the element. Put and get will take o(1). Key is value passed and value is index.
3. Map<val\_passed\_in\_argument,index> , arrayList<val>.
4. On insert we will add item to arraylist if map does not contains that key.
5. On remove we will remove by key from map and by map we know item index. so we just copy the last element from list to that index and delete the last element from list.
6. Random will generate random index from 0,list.size and list will return element.

### Solution

**class** RandomizedSet {

Random r = **new** Random();

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

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

/\*\* Initialize your data structure here. \*/

**public** RandomizedSet() {

}

/\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/

**public** **boolean** insert(**int** val) {

**if**(!map.containsKey(val)) {

map.put(val,list.size());

list.add(val);

**return** **true**;

}

**return** **false**;

}

/\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/

**public** **boolean** remove(**int** val) {

Integer index = map.get(val);

**if** (index == **null**) {

**return** **false**;

}

**int** lastVal = list.get(list.size() - 1);

list.set(index, lastVal);

list.remove(list.size() - 1);

map.put(lastVal,index);

map.remove(val);

**return** **true**;

}

/\*\* Get a random element from the set. \*/

**public** **int** getRandom() {

**int** index = r.nextInt(list.size());

**return** list.get(index);

}

}

### Time and space complexity

O(1)

O(2n)

# 14. 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 root 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.

## 3. K Closest Points to Origin

### Problem

We have a list of points on the plane.  Find the K closest points to the origin (0, 0).

(Here, the distance between two points on a plane is the Euclidean distance.)

You may return the answer in any order.  The answer is guaranteed to be unique (except for the order that it is in.)

**Example 1:**

**Input:** points = [[1,3],[-2,2]], K = 1

**Output:** [[-2,2]]

**Explanation:**

The distance between (1, 3) and the origin is sqrt(10).

The distance between (-2, 2) and the origin is sqrt(8).

Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

We only want the closest K = 1 points from the origin, so the answer is just [[-2,2]].

**Example 2:**

**Input:** points = [[3,3],[5,-1],[-2,4]], K = 2

**Output:** [[3,3],[-2,4]]

(The answer [[-2,4],[3,3]] would also be accepted.)

**Note:**

1. 1 <= K <= points.length <= 10000
2. -10000 < points[i][0] < 10000
3. -10000 < points[i][1] < 10000

### Reference

LEETCODE, GIT,MAX HEAP

### Approach

To calculate distance between two points use in a plane . distance = sqrt((x2-x1)2 + (y2-y1)2)

1. We will use max heap of size k. we will add first k elements of array directly to heap.

2. Now from k to points.length we will add element to heap if current element is smaller then peek element from heap. In such case remove top first and then add new.

### Solution

**public** **int**[][] kClosest(**int**[][] points, **int** k) {

PriorityQueue<**int**[]> maxHeap = **new** PriorityQueue<>(k, (o1, o2) -> *sum*(o2) - *sum*(o1));

**for** (**int** i = 0; i < k; i++) {

maxHeap.add(points[i]);

}

**for** (**int** i = k; i < points.length; i++) {

**if** (*sum*(points[i]) < *sum*(maxHeap.peek())) {

maxHeap.poll();

maxHeap.add(points[i]);

}

}

**int**[][] op = **new** **int**[k][2];

**for** (**int** i = 0; i < k; i++) {

op[i] = maxHeap.poll();

}

**return** op;

}

**private** **int** sum(**int**[] points1) {

**return** points1[0] \* points1[0] + points1[1] \* points1[1];

}

### Time and space complexity

Nlogk time

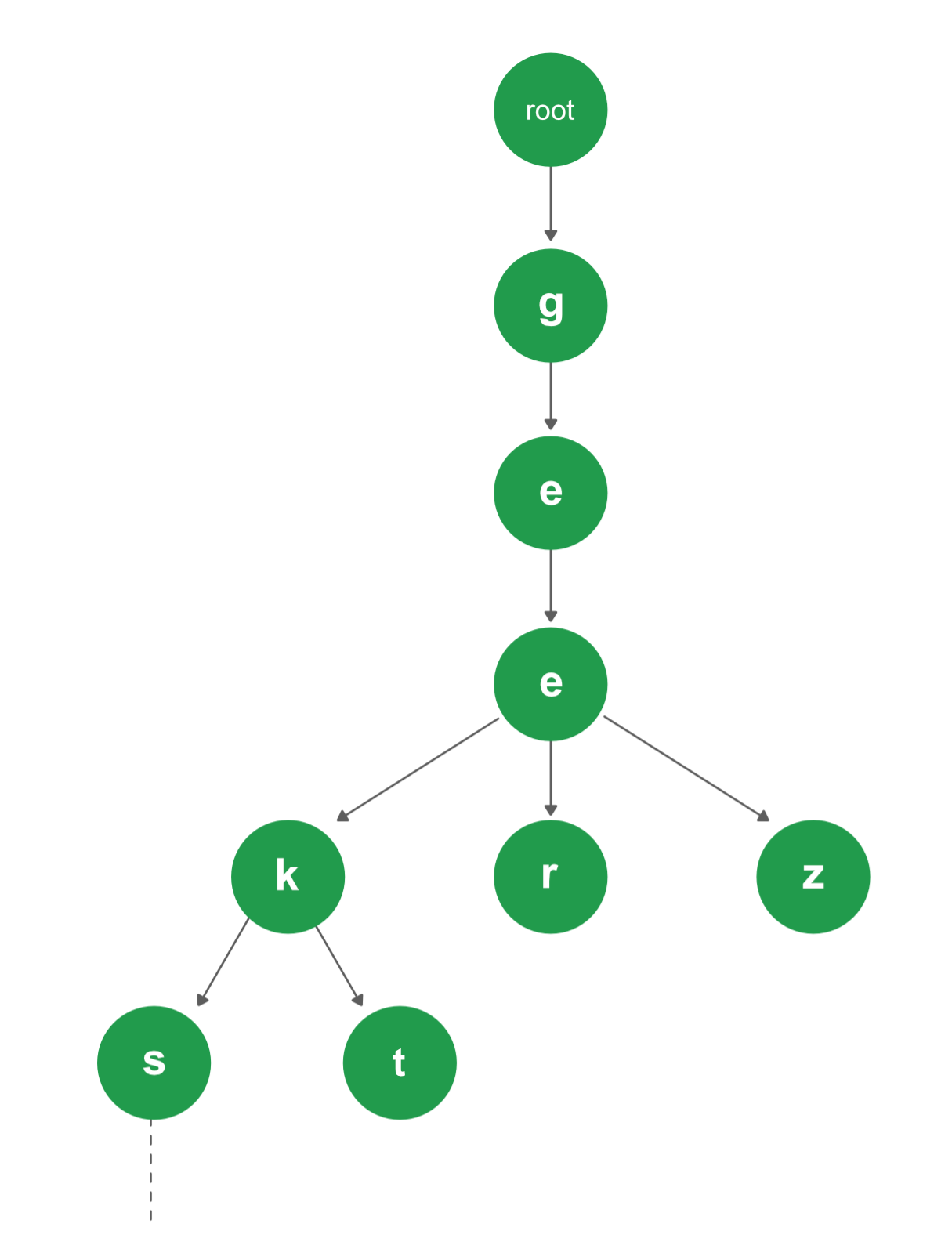
Logk space for heap

# Trie

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field isEndOfWord is used to distinguish the node as end of word node. A simple structure to represent nodes of the English alphabet can be as following,  
  
// Trie node  
struct TrieNode  
{  
     struct TrieNode \*children[ALPHABET\_SIZE];

     // isEndOfWord is true if the node  
     // represents end of a word  
     bool isEndOfWord;  
};

Every character of the input key is inserted as an individual Trie node. Note that the children is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array children



## 1. Implement Trie (Prefix Tree)

### Problem

Implement a Trie with insert, search, and startsWith methods.

**Example:**

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // returns true

trie.search("app"); // returns false

trie.startsWith("app"); // returns true

trie.insert("app");

trie.search("app"); // returns true

**Note:**

* You may assume that all inputs are consist of lowercase letters a-z.
* All inputs are guaranteed to be non-empty strings.

### Reference

LEETCODE, GIT

### Approach

1. Every node contains two fields – character array of 26 size and Boolean flag end which will be true if that node is last character. Index of character will be character – ‘a’
2. We will create root node at global level.
3. Insert – for every character we check if node present for that character index is null. If null we create TrieNode for that index in array. Now will update node to that newly created node and repeat procedure for next character.
4. After loop finished we set end flag to true for last node.
5. Searching is similar like insert. We check and match. If character does not match we return. Else go for child of matched object for next match. After all characters match return true if last node is set true in case of exact match. For prefix match just return true.

### Solution

class Trie {

/\*\* Initialize your data structure here. \*/

public Trie() {

}

private class TrieNode {

TrieNode[] characters = new TrieNode[26];

boolean end;

}

private TrieNode root=new TrieNode();

/\*\* Inserts a word into the trie. \*/

public void insert(String word) {

TrieNode node = root;

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

int level = word.charAt(i)-'a';

if(node.characters[level]==null) {

node.characters[level] = new TrieNode();

}

node=node.characters[level];

}

node.end=true;

}

/\*\* Returns if the word is in the trie. \*/

public boolean search(String word) {

TrieNode node = searchNode(word);

return node==null?false:node.end;

}

/\*\* Returns if there is any word in the trie that starts with the given prefix. \*/

public boolean startsWith(String word) {

return searchNode(word)!=null;

}

private TrieNode searchNode(String word) {

TrieNode node = root;

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

int level = word.charAt(i)-'a';

if(node.characters[level]==null) {

return null;

}

node=node.characters[level];

}

return node;

}

}

### Time and space complexity

Insert, search, prefixmatch take o(level) time.

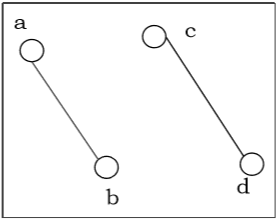
# Graph

1. A graph is a pair(V,E) where v is set of vertices & e is a collection of pair of vertices called edges.
2. Directed edge | undirected edge

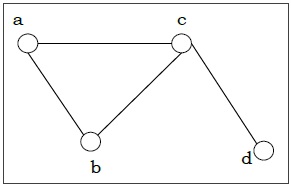
u🡪v u—v

1. In an edge u->v , we say u and v are adjacent vertices.
2. A tree is a type of graph with no cycle and always connected. i.e. acyclic connected graph
3. A graph is connected if there exists a path to go from 1 vertex to other.

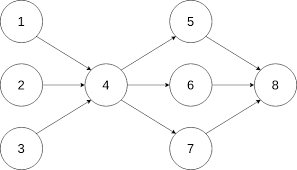
**Disconnected – (it is also called multi-component graph)**



**Connected –**



1. Two edges are parallel if they are connected to same vertices.
2. Degree of vertices is the number of edges incident on it.
3. Directed acyclic graph is directed graph without cycle.



1. A spanning tree of a connected graph is sub graph that contains all the vertices of a graph and is single tree.
2. Minimum spanning tree will have minimum edges to connected all vertices of graph.
3. Number of edges in a spanning tree = v-1
4. In weighted graph every edge has weight (int value).
5. Flight network, highway network, LAN are applications of networks.
6. Shortest path algo is used in GPS to suggest routes.
7. If by removing a vertex v from graph resultant graph become disconnected that vertex is called articulation point. e.g. vertex 4 in above graph is articulation point.

**GRAPH IMPLEMENTATION –**

https://cdncontribute.geeksforgeeks.org/wp-content/uploads/undirectedgraph.png

A. Graph adjacency matrix.

B. Graph adjacency list.

1. **Graph adjacency matrix –**
2. If number of vertices in graph is v we will create 2D array of size v\*v. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.

Adjacency Matrix Representation

1. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.
2. Pros: add, remove and find of edge take o(1) time.
3. Cons: irrespective on number of edges in graph. It will take o(v2) memory.
4. If we know graph is dense (i.e. lot of edges) this approach is best.

**public** **class** GraphAdjacencyMatrixImpl {

**private** **final** **int**[][] matrix;

**private** final **int** v;

**public** GraphAdjacencyMatrixImpl(**int** n) {

v = n;

matrix = **new** **int**[v][v];

}

**public** **void** addEdge(**int** i, **int** j) {

matrix[i][j] = 1;

//matrix[j][1] = 1; un-comment it for un-directed graph.

}

**public** **boolean** edgeExists(**int** i, **int** j) {

**return** matrix[i][j] != 0;

}

**public** **void** removeEdge(**int** i, **int** j) {

matrix[i][j] = 0;

}

**public** **void** print() {

System.***out***.println("Adjacency Matrix data -");

**for** (**int** i = 0; i < v; i++) {

**for** (**int** j = 0; j < v; j++) {

System.***out***.print(" " + matrix[i][j]);

}

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

}

}

}

1. **Graph adjacency list –**

An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. Where array size is v. and v is the number of vertices.

Adjacency List Representation of Graph

1. Pros: In case of sparse graph (less edges) it will take less space. Saves space O(|V|+|E|). Adding a vertex is easier.
2. Cons: Removal is very complex. to check whether an edge exist between two vertices can take o(n) time.

**public** **class** GraphAdjacencyListImpl {

**private** **final** **int** v;

**private** **final** List<Integer>[] adjacencyArray;

**public** GraphAdjacencyListImpl(**int** noOfVertices) {

v = noOfVertices;

adjacencyArray = **new** ArrayList[v];

IntStream.*range*(0, v).forEach(i -> adjacencyArray[i] = **new** ArrayList<>());

}

**public** **void** addEdge(**int** i, **int** j) {

adjacencyArray[i].add(j);

// adjacencyArray[j].add(i); for un-directed graph un-comment it.

}

**public** **boolean** isEdge(**int** i, **int** j) {

**int** res = adjacencyArray[i].stream().filter(x -> x == j).findAny().orElse(-1);

**return** res != -1;

}

**public** **void** print() {

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

System.***out***.print("Adjacency list of vertex: " + i);

adjacencyArray[i].forEach(o -> System.***out***.print("->" + o));

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

}

}

}

**Graph Traversal –**

|  |  |
| --- | --- |
| **BFS** | **DFS** |
| Breadth First search | Depth first search |
| In it we will first visit all the adjacent nodes of current node and then move for next level. | In it we will cover one adjacent node and then check other |
| It is just like level order traversal | It is like pre-order traversal. |
| Queue data structure is used along with visited array to keep of visited node track. | Stack data structure is used along with visited array to keep of visited node track. |
| Shortest path b/w 2 nodes, find all connected component in graph, minimum spanning tree, GPS. | Detect cycle in graph, topological sort, solve puzzle with 1 solution. |
| Java garbage collector, web crawler of browser | Used to find articulation point in graph. |
| **BFS** is **better** when target is closer to Source | **DFS** is **better** when target is far from source |
| **O(v+e) time** for adjacency list implementation | **O(v+e) time** foradjacency list implementation |
| BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles. | DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop. |
| Space complexity for adjacency list– o(V) as in worst case we need to hold all vertexes in queue at once. | Space complexity for adjacency list– o(V) for iterative solution using stack and o(h) for recursive solution. where h is the maximal depth of your tree. |

1. **Graph adjacency matrix BFS and DFS –**

**public** **void** bfs(**int** source) {

**if** (source < 0 || source >= v) {

**return**;

}

**boolean**[] visited = **new** **boolean**[v];

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

queue.add(source);

visited[source] = **true**;

System.***out***.println("bfs starting from vertex " + source + " is-");

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

**int** i = queue.poll();

System.***out***.print(" " + i);

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

**if** (matrix[i][j] != 0 && !visited[j]) {

visited[j] = **true**;

queue.add(j);

}

}

}

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

}

**public** **void** dfs(**int** source) {

**if** (source < 0 || source >= v) {

**return**;

}

**boolean**[] visited = **new** **boolean**[v];

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

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

**int** i = stack.pop();

System.***out***.print(" " + i);

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

**if** (!visited[j] && matrix[i][j] != 0) {

visited[j] = **true**;

stack.push(j);

}

}

}

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

}

**public** **void** dfsRecusrive(**int** source) {

**if** (source < 0 || source >= v) {

**return**;

}

System.***out***.println("DFS starting from vertex : " + source);

dfsUtil(source, **new** **boolean**[v]);

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

}

**private** **void** dfsUtil(**int** source, **boolean**[] visited) {

visited[source] = **true**;

System.***out***.print(" " + source);

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

**if** (!visited[j] && matrix[source][j] != 0) {

dfsUtil(j, visited);

}

}

}

1. **Graph adjacency List BFS and DFS –**

**public** **void** bfs(**int** source) {

**if** (source < 0 || source >= v) {

**return**;

}

System.***out***.println("BFS starting from vertex : " + source);

**boolean**[] visited = **new** **boolean**[v];

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

queue.add(source);

visited[source] = **true**;

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

**int** i = queue.poll();

System.***out***.print(" " + i);

**for** (**int** j : adjacencyArray[i]) {

**if** (!visited[j]) {

queue.add(j);

visited[j] = **true**;

}

}

}

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

}

**public** **void** dfsIterative(**int** source) {

**if** (source < 0 || source >= v) {

**return**;

}

System.***out***.println("DFS starting from vertex : " + source);

**boolean**[] visited = **new** **boolean**[v];

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

stack.push(source);

visited[source] = **true**;

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

**int** i = stack.pop();

System.***out***.print(" " + i);

**for** (**int** j : adjacencyArray[i]) {

**if** (!visited[j]) {

stack.push(j);

visited[j] = **true**;

}

}

}

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

}

**public** **void** dfsRecursive(**int** source) {

**if** (source < 0 || source >= v) {

**return**;

}

System.***out***.println("DFS starting from vertex : " + source);

dfsUtil(source, **new** **boolean**[v]);

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

}

**private** **void** dfsUtil(**int** source, **boolean**[] visited) {

visited[source] = **true**;

System.***out***.print(" " + source);

**for** (**int** i : adjacencyArray[source]) {

**if** (!visited[i]) {

dfsUtil(i, visited);

}

}

}

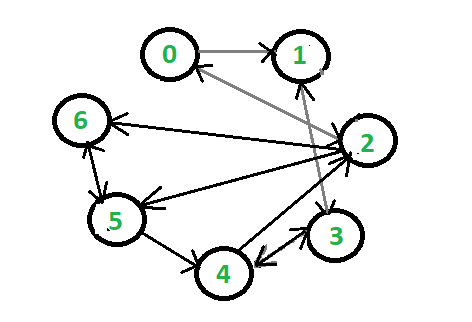
## 1. Finding in and out degrees of all vertices in a graph

### Problem

Given a directed graph, the task is to count the in and out degree of each vertex of the graph.

**Examples:**

**Input:**



**Output:**

Vertex In Out

0 1 2

1 2 1

2 2 3

3 2 2

4 2 2

5 2 2

6 2 1

### Reference

LEETCODE, GIT,GEEKSFORGEEKS

### Approach

Iterate through the adjList and the number of elements at particular index is out degree.

and number of times any vertex occurs in all list is in degree

### Solution

**public** **void** printInAndOutDegree() {

**int**[] in = **new** **int**[noOfVertices];

**int**[] out = **new** **int**[noOfVertices];

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

out[i] = adjListArray[i].size();

**for** (**int** obj : adjListArray[i]) {

in[obj] = in[obj] + 1;

}

}

System.***out***.println("v in out");

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

System.***out***.println(i + " " + in[i] + " " + out[i]);

}

}

### Time and space complexity

Time – o(ve)

Space -If adjacency list is already prepared – o(v)

## 2. Topological order of graph (Topological Sorting)

### Problem

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

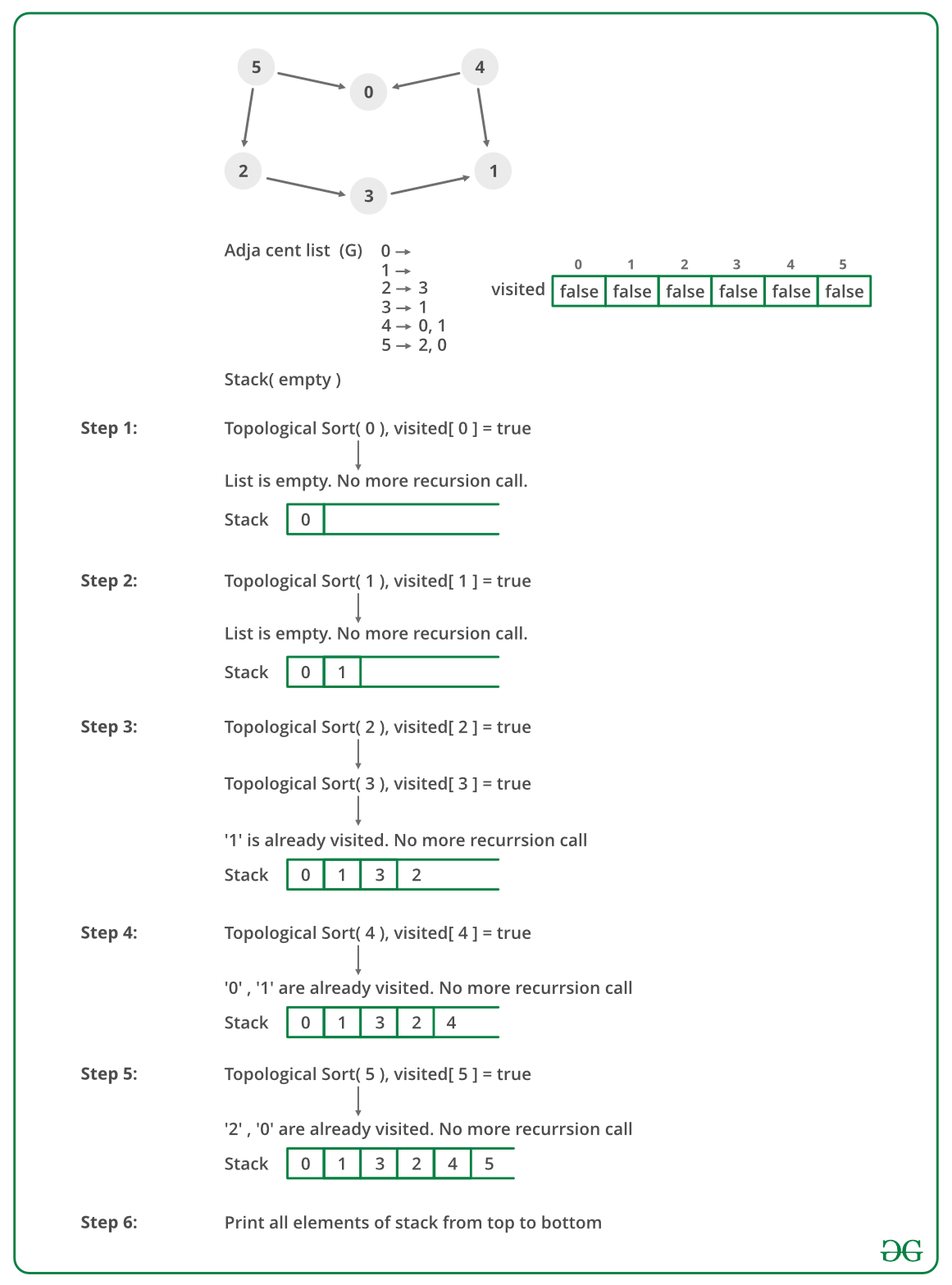
For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/graph.png)  
**Topological Sorting vs Depth First Traversal (DFS)**:  
In [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting

### Reference

LEETCODE, GIT, GEEKSFORGEEKS

### Approach

1. As it is clear by topological order that for every edge u->v, vertex u will appear before vertex v.
2. It is used in maven dependency resolution where parent should be resolved first and later child.
3. We can modify [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)to find Topological Sorting of a graph. In [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.
4. Below image is an illustration of the above approach:
5. 

### Solution

**public** **void** topologicalSort() {

System.***out***.println("Topological sort :");

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

**boolean**[] visited = **new** **boolean**[noOfVertices];

**for** (**int** i = 0; i < noOfVertices; i++) {

**if** (!visited[i]) {

topologicalSortRecursion(i, visited, stack);

}

}

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

System.***out***.print(stack.pop() + " ");

}

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

}

**private** **void** topologicalSortRecursion(**int** source, **boolean**[] visited, Deque<Integer> stack) {

visited[source] = **true**;

**for** (**int** j : adjListArray[source]) {

**if** (!visited[j]) {

topologicalSortRecursion(j, visited, stack);

}

}

stack.push(source);

}

### Time and space complexity

O(v+e) time

O(v) space

## 3. Course Schedule (detect cycle in DAG)

### Problem

There are a total of numCourses courses you have to take, labeled from 0 to numCourses-1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

**Example 1:**

**Input:** numCourses = 2, prerequisites = [[1,0]]

**Output:** true

**Explanation:** There are a total of 2 courses to take.

  To take course 1 you should have finished course 0. So it is possible.

**Example 2:**

**Input:** numCourses = 2, prerequisites = [[1,0],[0,1]]

**Output:** false

**Explanation:** There are a total of 2 courses to take.

  To take course 1 you should have finished course 0, and to take course 0 you should

  also have finished course 1. So it is impossible.

**Constraints:**

* The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices.
* You may assume that there are no duplicate edges in the input prerequisites.
* 1 <= numCourses <= 10^5

### Reference

LEETCODE, GITHUB, GeeksForGeeks

### Approach

1. It is like finding cycle in directed graph. If cycle exists it is not possible to finish the course else it is possible.
2. We can use topological sort to detect cycle in graph. Just instead of adding element in recStack we can take recStack as Boolean array. and if any given point in a recursive call a node appears twice in recStack it means cycle exists. so basically we need to keep track whether there exist duplicate element in the recStack.
3. Once all the adjacent nodes of a vertex v is visited we will set recStack as false for v. this is because v has been explored and needed to be remove from recStack.

### Solution

**class** Solution {

**private** List<Integer>[] adjArray;

**public** **boolean** canFinish(**int** numCourses, **int**[][] prerequisites) {

prepareGraph(numCourses, prerequisites);

**boolean**[] visited = **new** **boolean**[numCourses];

**boolean**[] recStack = **new** **boolean**[numCourses];

**for** (**int** i = 0; i < numCourses; i++) {

**if** (hasCycle(i, visited, recStack)) {

**return** **false**;

}

}

**return** **true**;

}

**private** **boolean** hasCycle(**int** i, **boolean**[] visited, **boolean**[] recStack) {

**if** (recStack[i]) {

**return** **true**;

}

**if** (visited[i]) {

**return** **false**;

}

recStack[i] = **true**;

visited[i] = **true**;

**for** (**int** j : adjArray[i]) {

**if** (hasCycle(j, visited, recStack)) {

**return** **true**;

}

}

recStack[i] = **false**;

**return** **false**;

}

**private** **void** prepareGraph(**int** numCourses, **int**[][] prerequisites) {

adjArray = **new** ArrayList[numCourses];

// intialize graph adjArray

**for** (**int** i = 0; i < numCourses; i++) {

adjArray[i] = **new** ArrayList<Integer>();

}

// add edges

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

adjArray[prerequisites[i][1]].add(prerequisites[i][0]);

}

}

}

\*\***NOTE** – in above code we have called hasCycle and inside it we have checked visited. But in dfs and topological sort we don’t call recursive method if node already visited. Here we have changed this logic because we wanted to check recStack first for current node and after that check visited array. If we use classic dfs logic we might fail to detect cycle in e.g. like [[1,0],[0,1]]

### Time and space complexity

* **Time Complexity:**O(V+E).  
  Time Complexity of this method is same as time complexity of DFS which is O(V+E).
* **Space Complexity:** O(V).  
  To store the visited and recursion stack O(V) space is needed.

## 4. Possible Bipartition ( check If graph is bi-partite)

### Problem

Given a set of N people (numbered 1, 2, ..., N), we would like to split everyone into two groups of **any** size.

Each person may dislike some other people, and they should not go into the same group.

Formally, if dislikes[i] = [a, b], it means it is not allowed to put the people numbered a and b into the same group.

Return true if and only if it is possible to split everyone into two groups in this way.

**Example 1:**

**Input:** N = 4, dislikes = [[1,2],[1,3],[2,4]]

**Output:** true

**Explanation**: group1 [1,4], group2 [2,3]

**Example 2:**

**Input:** N = 3, dislikes = [[1,2],[1,3],[2,3]]

**Output:** false

**Example 3:**

**Input:** N = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]

**Output:** false

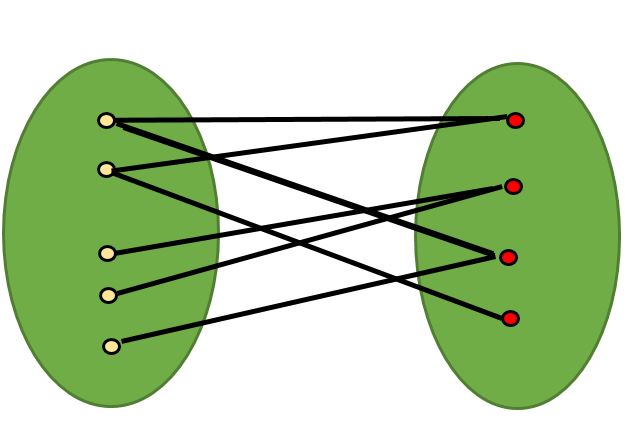
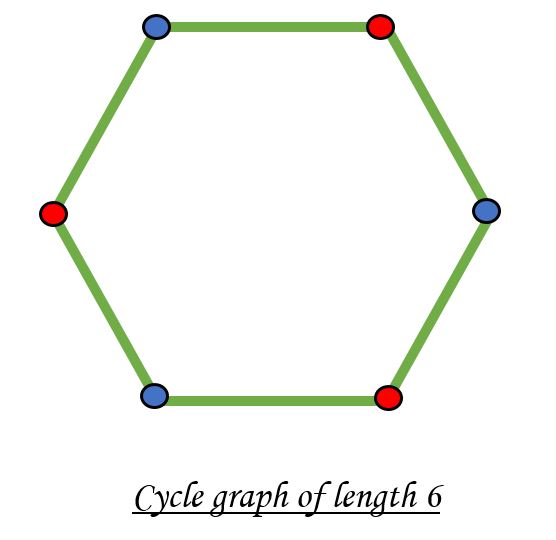
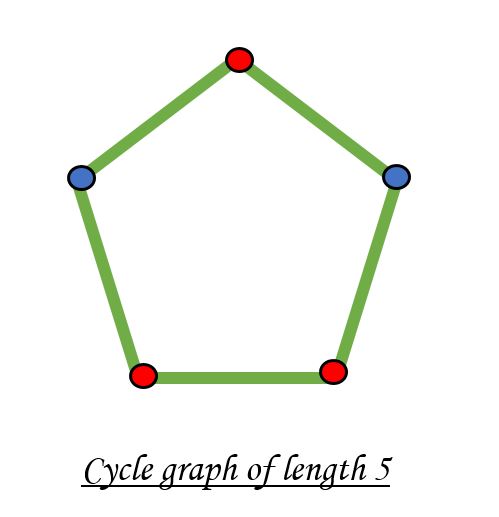
**Constraints:**

* 1 <= N <= 2000
* 0 <= dislikes.length <= 10000
* dislikes[i].length == 2
* 1 <= dislikes[i][j] <= N
* dislikes[i][0] < dislikes[i][1]
* There does not exist i != j for which dislikes[i] == dislikes[j].

### Reference

LEETCODE, GITHUB, GeeksForGeeks, leetcode

### Approach

* 1. In this question, it is asked if it is possible to divide objects into group. Or in-terms of graph. We need to check if graph is bipartite.
* A [Bipartite Graph](http://en.wikipedia.org/wiki/Bipartite_graph) is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.
* [](https://media.geeksforgeeks.org/wp-content/uploads/bipartitegraph-1.jpg)
* A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.
* [](https://media.geeksforgeeks.org/wp-content/uploads/bipartitegraphfive.sixJPG.jpg)
* It is not possible to color a cycle graph with odd cycle using two colors.  
  [](https://media.geeksforgeeks.org/wp-content/uploads/bipartitegraphfive.jpg)

**NOTE –**

* 1. **So it is clear if there exists a node whose adjust node is also having same color. The graph is not bi-partite. (i.e. by graph coloring we are checking if graph contain odd length cycle)**

**Algorithm –**

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).  
1. Assign RED color to the source vertex (putting into set U).  
2. Color all the neighbors with BLUE color (putting into set V).  
3. Color all neighbor’s neighbor with RED color (putting into set U).  
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.  
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

Int color has three options.

Color=0 means not colored yet

Color=1 means red color

Color=-1 means blue color.

Since it is possible that graph is multi connected. So we need to check for every vertex. We will call above algo for every vertex if color of that current vertex is 0.

### Solution

**class** BiPartiteGraph {

**public** **boolean** possibleBipartition(**int** n, **int**[][] dislikes) {

List<Integer>[] graph = constructGraph(n, dislikes);

**int**[] color = **new** **int**[n];

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

// if (color[i] == 0 && !dfsUtil(color, graph, i)) {

**if** (color[i] == 0 && !bfsUtil(color, graph, i)) {

**return** **false**;

}

}

**return** **true**;

}

**private** **boolean** bfsUtil(**int**[] color, List<Integer>[] graph, **int** source) {

color[source] = 1;

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

queue.add(source);

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

**int** curr = queue.poll();

**for** (**int** adj : graph[curr]) {

**if** (color[curr] == color[adj]) {

**return** **false**;

}

**if** (color[adj] == 0) {

color[adj] = -color[curr];

queue.add(adj);

}

}

}

**return** **true**;

}

**private** **boolean** dfsUtil(**int**[] color, List<Integer>[] graph, **int** curr, **int** c) {

**if** (color[curr] != 0) {

**return** color[curr] == c;

}

color[curr] = c;

**for** (**int** adj : graph[curr]) {

**if** (!dfsUtil(color, graph, adj, -c)) {

**return** **false**;

}

}

**return** **true**;

}

**private** List<Integer>[] constructGraph(**int** n, **int**[][] dislikes) {

List<Integer>[] graph = **new** ArrayList[n];

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

graph[i] = **new** ArrayList<>();

}

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

graph[dislikes[i][0] - 1].add(dislikes[i][1] - 1);

graph[dislikes[i][1] - 1].add(dislikes[i][0] - 1);

}

**return** graph;

}

}

### Time and space complexity

* **Time Complexity:**O(V+E).
* **Space** : o(V+E)

## 5. Surrounded Regions

### Problem

Given a 2D board containing 'X' and 'O' (**the letter O**), capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

**Example:**

X X X X

X O O X

X X O X

X O X X

After running your function, the board should be:

X X X X

X X X X

X X X X

X O X X

**Explanation:**

Surrounded regions should not be on the border, which means that any 'O' on the border of the board are not flipped to 'X'. Any 'O' that is not on the border and it is not connected to an 'O' on the border will be flipped to 'X'. Two cells are connected if they are adjacent cells connected horizontally or vertically.

### Reference

LEETCODE, ARRAY, GITHUB, GRAPH

### Approach

It is similar to leetcode problem to find Number of Islands in 2-D Array.

* 1. We will use dfs to traverse nodes.
  2. To make solution simple we will handle boundary checks first and visit as much O as possible. while visiting make them 'v'. so that after dfs call completes we now know that O remaining are those which needed to be converted to X and all 'v' needs to be changed back to O.

### Solution

**public** **void** captureAllRegionsExceptBoundary(**char**[][] board) {

**if** (board.length == 0 || board[0].length == 0) {

**return**;

}

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

**if** (board[0][i] == 'O') {

dfsBoundaryUtil(board, 0, i);

}

**if** (board[board.length - 1][i] == 'O') {

dfsBoundaryUtil(board, board.length - 1, i);

}

}

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

**if** (board[i][0] == 'O') {

dfsBoundaryUtil(board, i, 0);

}

**if** (board[i][board[0].length - 1] == 'O') {

dfsBoundaryUtil(board, i, board[0].length - 1);

}

}

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

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

**if** (board[i][j] == 'v') {

board[i][j] = 'O';

} **else** **if** (board[i][j] == 'O') {

board[i][j] = 'X';

}

}

}

}

**private** **void** dfsBoundaryUtil(**char**[][] board, **int** i, **int** j) {

**if** (i < 0 || i >= board.length || j < 0 || j >= board[0].length || board[i][j] != 'O') {

**return**;

}

board[i][j] = 'v';

dfsBoundaryUtil(board, i - 1, j);

dfsBoundaryUtil(board, i + 1, j);

dfsBoundaryUtil(board, i, j - 1);

dfsBoundaryUtil(board, i, j + 1);

}

### Time and space complexity

Time - O(nm)

Space – O(1).

## 6. Course Schedule II

### Problem

There are a total of n courses you have to take, labeled from 0 to n-1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

**Example 1:**

**Input:** 2, [[1,0]]

**Output:** [0,1]

**Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished

  course 0. So the correct course order is [0,1] .

**Example 2:**

**Input:** 4, [[1,0],[2,0],[3,1],[3,2]]

**Output:** [0,1,2,3] or [0,2,1,3]

**Explanation:** There are a total of 4 courses to take. To take course 3 you should have finished both

courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.

  So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3] .

**Note:**

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs).
2. You may assume that there are no duplicate edges in the input prerequisites.

### Reference

LEETCODE, ARRAY, GITHUB, GRAPH

### Approach

It is similar to leet code problem of course schedule 1 where we just need to detect cycle. In this problem if cycle does not exists return output else empty array.

We will use topological sort to find order. Just in-place of stack we will use array as we need to return array. and we will initialize count= no of vertex to make it behave like stack.

To save visited array space. I have taken recStack array as integer and 1 value represent part of call stack. 2 indicates visited completely. Classical approach with two array will also work.

### Solution

**class** CourseSchedule {

**private** List<Integer>[] adjArray;

**int** count;

**public** **int**[] findOrder(**int** numCourses, **int**[][] prerequisites) {

prepareGraph(numCourses, prerequisites);

**int**[] recStack = **new** **int**[numCourses];

**int**[] output = **new** **int**[numCourses];

count = numCourses - 1;

**for** (**int** i = 0; i < numCourses; i++) {

**if** (cycleDfs(recStack, i, output)) {

**return** **new** **int**[0];

}

}

**return** output;

}

**private** **boolean** cycleDfs(**int**[] recStack, **int** i, **int**[] res) {

**if** (recStack[i] == 1) {

**return** **true**;

}

**if** (recStack[i] == 2) {

**return** **false**;

}

recStack[i] = 1;

**for** (**int** j : adjArray[i]) {

**if** (cycleDfs(recStack, j, res)) {

**return** **true**;

}

}

res[count--] = i;

recStack[i] = 2;

**return** **false**;

}

**private** **void** prepareGraph(**int** n, **int**[][] edges) {

adjArray = **new** ArrayList[n];

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

adjArray[i] = **new** ArrayList<>();

}

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

adjArray[edges[i][1]].add(edges[i][0]);

}

}

### Time and space complexity

Time - O(V+E)

Space – O(2V).

## 7. Dijkstra Algorithm (find shortest path from source to all vertex)

### Problem

There are a total n vertex, labeled from 0 to n-1. It is given edge weight cannot be negative.

Print shortest path from source vertex passed in argument to all other vertexes.

**Example 1:**

**Input:**

n = 3, edges = [[0,1,9],[0,2,6],[0,3,5],[0,4,3],[2,1,2],[2,3,4]], src = 0

**Output:**

Dijkstra Shortest path from node :

0 -> 0 has minimum cost 0 and path is [ 0 ]

0 -> 1 has minimum cost 8 and path is [ 0 2 1 ]

0 -> 2 has minimum cost 6 and path is [ 0 2 ]

0 -> 3 has minimum cost 5 and path is [ 0 3 ]

0 -> 4 has minimum cost 3 and path is [ 0 4 ]

**Note:**

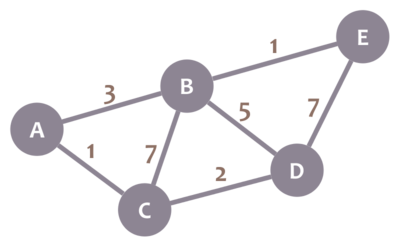
1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs).

### Reference

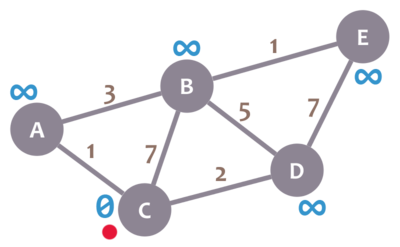
BFS, GITHUB, GRAPH,ABDUL BARI

### Approach

 let's study the algorithm with an explained example! Let's calculate the shortest path between node C and the other nodes in our graph:

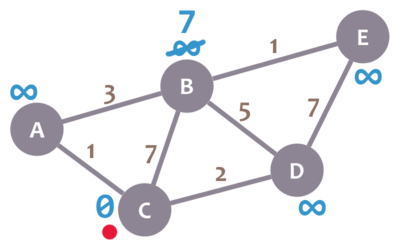


During the algorithm execution, we'll mark every node with its *minimum distance* to node C (our selected node). For node C, this distance is 0. For the rest of nodes, as we still don't know that minimum distance, it starts being infinity (∞):

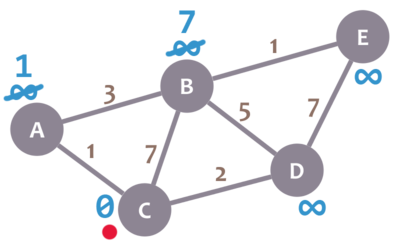


We'll also have a *current node*. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot.

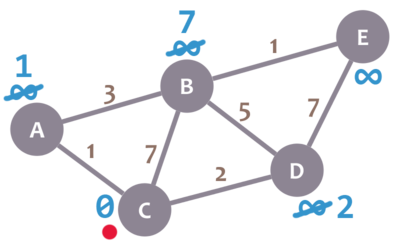
Now, we check the neighbours of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):



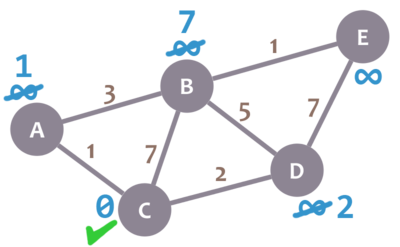
So far, so good. Now, let's check neighbour A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value:



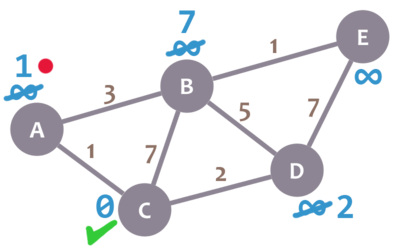
OK. Repeat the same procedure for D:



Great. We have checked all the neighbours of C. Because of that, we mark it as *visited*. Let's represent visited nodes with a green check mark:

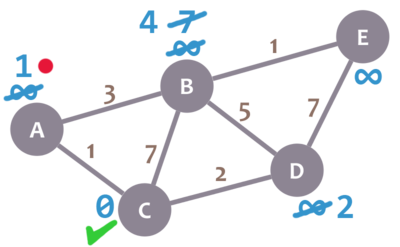


We now need to pick a new *current node*. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A. Let's mark it with the red dot:

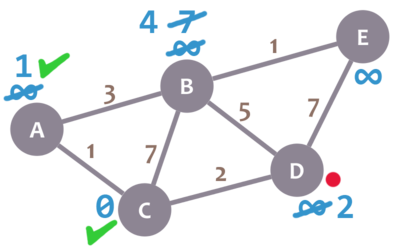


And now we repeat the algorithm. We check the neighbours of our current node, ignoring the visited nodes. This means we only check B.

For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.



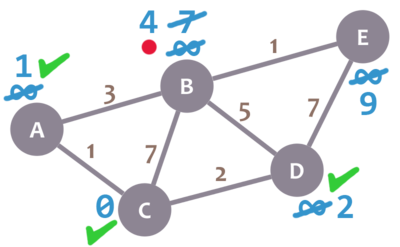
Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance.



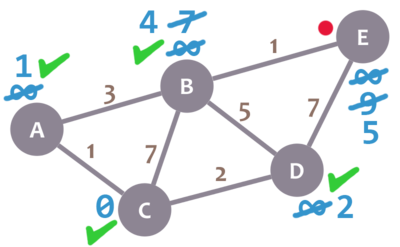
We repeat the algorithm again. This time, we check B and E.

For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4). For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).

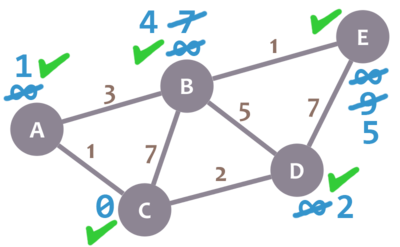
We mark D as visited and set our current node to B.



Almost there. We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.



E doesn't have any non-visited neighbours, so we don't need to check anything. We mark it as visited.



As there are not univisited nodes, we're done! The minimum distance of each node now actually represents the minimum distance from that node to node C (the node we picked as our initial node)!

Here's a description of the algorithm:

1. Mark your selected initial node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node C.
3. For each neighbour N of your current node C: add the current distance of C with the weight of the edge connecting C-N. If it's smaller than the current distance of N, set it as the new current distance of N.
4. Mark the current node C as visited.
5. If there are non-visited nodes, go to step 2.

**Implementation** –

1. Distance array is used to store the min distance for every node.
2. array of settled node is taken. which will hold true if the index of vertex is settled.
3. PriorityQueue MinHeap is used. so that we can get the minimum node at o(1) time.
4. the data stored in heap will be the distance of node on given index.(index,distance)
5. prev array is taken to hold the path.
6. AdjArray contains three fields (src,dest,distance) just to save memory (as input array already contains three fields) three fields are taken otherwise only dest and distance is needed.
7. set distance of all nodes to max\_Value and source as 0.
8. add source to MinHeap.
9. Apply bfs to use minheap in place of queue.

**STEPS –**

\* get top element u from heap and add it to the settled set.

\* for every adjacent node v of popped element u. check if that node is already present in settled set.

\* if not. perform relaxation. i.e. update distance of v if dist[u]+v.wieght<dist[v]

\* after that add that adj node v to heap with updated distance.

### Solution

**public** **int** dijikstraShortestPath(**int** n, **int**[][] flights, **int** src) {

**if** (flights.length == 0) {

**return** -1;

}

List<**int**[]>[] adjArray = prepareAdjGraph(n, flights);

PriorityQueue<**int**[]> minHeap = **new** PriorityQueue<>((a, b) -> a[1] - b[1]);

minHeap.add(**new** **int**[] { src, 0 });

**boolean**[] settled = **new** **boolean**[n];

**int**[] prev = **new** **int**[n]; // taken to store path

**int**[] dist = **new** **int**[n]; // taken to store distance

IntStream.*range*(0, dist.length).forEach(i -> dist[i] = Integer.***MAX\_VALUE***);

dist[src] = 0;

prev[src] = -1;

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

**int**[] edgeSrc = minHeap.poll();

settled[edgeSrc[0]] = **true**;

**for** (**int**[] j : adjArray[edgeSrc[0]]) {

**if** (!settled[j[1]] && j[2] + edgeSrc[1] < dist[j[1]]) {

minHeap.add(**new** **int**[] { j[1], j[2] + edgeSrc[1] });

prev[j[1]] = edgeSrc[0];

dist[j[1]] = j[2] + edgeSrc[1];

}

}

}

System.***out***.println("Dijkstra Shortest path from node :");

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

System.***out***.print(src + " -> " + i + " has minimum cost " + dist[i] + " and path is [ ");

printRoute(prev, i);

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

}

**return** -1;

}

**private** **void** printRoute(**int**[] prev, **int** i) {

**if** (i < 0) {

**return**;

}

printRoute(prev, prev[i]);

System.***out***.print(i + " ");

}

**private** List<**int**[]>[] prepareAdjGraph(**int** n, **int**[][] array) {

List<**int**[]>[] adjArray = **new** ArrayList[n];

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

adjArray[i] = **new** ArrayList<>();

}

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

adjArray[array[i][0]].add(array[i]);//to save space directly array added

}

**return** adjArray;

}

### Time and space complexity

Time - o(V+ElogV) after excluding path print

Space – O(V+E).

## Cheapest Flights Within K Stops

### Problem

There are n cities connected by m flights. Each flight starts from city u and arrives at v with a price w.

Now given all the cities and flights, together with starting city src and the destination dst, your task is to find the cheapest price from src to dst with up to k stops. If there is no such route, output -1.

**Example 1:**

**Input:**

n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]

src = 0, dst = 2, k = 1

**Output:** 200

**Explanation:**

The graph looks like this:



The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red in the picture.

**Example 2:**

**Input:**

n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]

src = 0, dst = 2, k = 0

**Output:** 500

**Explanation:**

The graph looks like this:



The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue in the picture.

### Reference

LEETCODE, ARRAY, GITHUB, GRAPH

### Approach

1. We can apply dijkstra algorithm with some modification. Like if we reach dest node from source we will stop algorithm.
2. Will add items to minHeap if that node has value less than equal to k.
3. We don’t need dist array as we are interested in one node only.
4. Since we want answer acc to k not necessary it should be minimum so we will not take settled Boolean array.
5. And we will add nodes into heap directly whether it is smaller distance or not. By this we might add same node in heap again with different distance. Heap will first try it with min value else with larger one if k variable condition did not meet.

### Solution

**public** **int** findCheapestPrice(**int** n, **int**[][] flights, **int** src, **int** dst, **int** k) {

List<**int**[]>[] adjArray = prepareAdjGraph(n, flights);

PriorityQueue<**int**[]> minHeap = **new** PriorityQueue<>((a, b) -> a[1] - b[1]);

minHeap.add(**new** **int**[] { src, 0, k + 1 });

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

**int**[] edgeSrc = minHeap.poll();

**if** (edgeSrc[0] == dst) {

**return** edgeSrc[1];

}

**if** (edgeSrc[2] > 0) {// will not consider nodes beyond k value

**for** (**int**[] j : adjArray[edgeSrc[0]]) {

minHeap.add(**new** **int**[] { j[1], j[2] + edgeSrc[1], edgeSrc[2] - 1 });

}

}

}

**return** -1;

}

**private** List<**int**[]>[] prepareAdjGraph(**int** n, **int**[][] array) {

List<**int**[]>[] adjArray = **new** ArrayList[n];

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

adjArray[i] = **new** ArrayList<>();

}

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

adjArray[array[i][0]].add(array[i]);//to save space directly added array

}

**return** adjArray;

}

### Time and space complexity

Time - O(V+ELOGV)

## Bellman Ford Algorithm (find shortest path from source to all vertex with negative)

### Problem

There are a total n vertex, labeled from 0 to n-1.

Print distance from source vertex passed in argument to all other vertexes. If cycle exists return false

**Example 1:**

**Input:**

n = 3, edges = [[0,1,9],[0,2,6],[0,3,5],[0,4,3],[2,1,2],[2,3,4]], src = 0

**Output**:

BellMan Ford Shortest path from node :

0 -> 0 has minimum cost 0

0 -> 1 has minimum cost 8

0 -> 2 has minimum cost 6

0 -> 3 has minimum cost 5

0 -> 4 has minimum cost 3

### Reference

LEETCODE, GITHUB, GRAPH, ABDUL BARI

### Approach

It is similar to dijkstra just we will relax every node n-1 times. If graph contains negative edges it will work.

In case of cycle bellman ford will not work. But yes it can detect cycle if present. Just relax nodes one more time and if we found dist reduced as compare to previous one. It means cycle exists. O(EV)

Bellman ford is slower then dijkstra and if we are sure cycle does not exists and graph is directed we can use DAG shortest path algorithm as it is the fastest one i.e. o(E+V).

In it we don’t need priorityqueue and settled vertex as we just run through all adjnodes and relax them n-1 times.

By relaxing we new sum is smaller than node current sum update it.

Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*

Example Graph

Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.  
https://media.geeksforgeeks.org/wp-content/uploads/bellmanford2.png

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).

https://media.geeksforgeeks.org/wp-content/uploads/bellmanford3.png

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.

### Solution

**public** **boolean** shortestPathPossible(**int** source) {

**int**[] dist = **new** **int**[noOfVertices];

// initialize all the distance to max value as per Algo

IntStream.*range*(0, dist.length).forEach(i -> dist[i] = Integer.***MAX\_VALUE***);

dist[source] = 0;

**for** (**int** i = 0; i < noOfVertices - 1; i++) {

**for** (GraphWeightedNode node : adjList) {

**int** sum = dist[node.getSource()] + node.getWeight();

**if** (sum < dist[node.getDest()]) {

dist[node.getDest()] = sum;

}

}

}

// check one more time for negative weight cycle Detection

**for** (GraphWeightedNode node : adjList) {

**int** sum = dist[node.getSource()] + node.getWeight();

**if** (sum < dist[node.getDest()]) {

**return** **false**;

}

}

System.***out***.println("BellMan Ford Shortest path from node :");

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

System.***out***.println(source + " -> " + i + " has minimum cost " + dist[i]);

}

**return** **true**;

}

**public** **class** GraphWeightedNode {

**private** **int** source;

**private** **int** dest;

**private** **int** weight;

}

### Time and space complexity

Time - O(VE)

## DAG Shortest/Longest Path algorithm (find shortest path from source to all vertex with negative)

### Problem

There are a total n vertex, labeled from 0 to n-1.

Print distance from source vertex passed in argument to all other vertexes.

Given Cycle is not present.

**Example 1:**

**Input:**

n = 3, edges = [[0,1,9],[0,2,6],[0,3,5],[0,4,3],[2,1,2],[2,3,4]], src = 0

**Output**:

Shortest path from node :

0 -> 0 has minimum cost 0

0 -> 1 has minimum cost 8

0 -> 2 has minimum cost 6

0 -> 3 has minimum cost 5

0 -> 4 has minimum cost 3

### Reference

LEETCODE, GITHUB, GRAPH, ABDUL BARI

### Approach

It is fastest among dijkstra and bellman ford. Limitation is it require acyclic directed graph only.

O(V+E)

TODO

### Solution

TODO

### Time and space complexity

Time - O(V+E)

## 11. Evaluate Division

### Problem

Equations are given in the format A / B = k, where A and B are variables represented as strings, and k is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return -1.0.

**Example:**  
Given a / b = 2.0, b / c = 3.0.  
queries are: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .  
return [6.0, 0.5, -1.0, 1.0, -1.0 ].

The input is: vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries , where equations.size() == values.size(), and the values are positive. This represents the equations. Return vector<double>.

According to the example above:

equations = [ ["a", "b"], ["b", "c"] ],

values = [2.0, 3.0],

queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].

The input is always valid. You may assume that evaluating the queries will result in no division by zero and there is no contradiction.

### Reference

LEETCODE, ARRAY, GITHUB, GRAPH

### Approach

1. Will use dfs approach. First prepare adjMap of <String,List<Node>> where key is the name of vertex i.e. “A” or “B” and value is the list of adjacency nodes. Since here weight is also needed .hence we store Node (dest,val) fields.
2. While preparing adjMap for edge a,b,2.0 we will two entries in map –
   1. add a as key and node (b,2.0) as value.
   2. Add b as key and node (a,1/2.0) as value.
3. Now map is created. We will iterate via queries and if map contains src. We will call dfs to give result else set -1 for that node.
4. Dfs method will set visited node true and process adjNodes. If current node reaches dest return 1. If dead end reached return -1.
5. TODO DFS explain

### Solution

**class** Solution {

**class** Node {

**double** val;

String dest;

Node(**double** val,String dest) {

**this**.val=val;

**this**.dest=dest;

}

}

**public** **double**[] calcEquation(List<List<String>> equations, **double**[] values, List<List<String>> queries) {

**double**[] output = **new** **double**[queries.size()];

Map<String,List<Node>> map = prepareGraph(equations,values);

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

String src = queries.get(i).get(0);

String dest = queries.get(i).get(1);

**if**(map.containsKey(src) && map.containsKey(dest)) {

output[i] = dfs(map,src,dest,**new** HashSet<>());

}

**if**(output[i]<=0.0) {

output[i]=-1;

}

}

**return** output;

}

**private** **double** dfs(Map<String,List<Node>> map,String src,String dest,Set<String> visited) {

**if**(src.equals(dest)) {

**return** 1.0;

}

**if**(visited.contains(src)) {

**return** -1;

}

visited.add(src);

**double** res = 1.0;

**for**(Node node: map.get(src)) {

res=node.val\*dfs(map,node.dest,dest,visited);

**if**(res>0){

**return** res;

}

}

**return** res;

}

**private** Map<String,List<Node>> prepareGraph(List<List<String>> equations, **double**[] values) {

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

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

String src = equations.get(i).get(0);

String dest = equations.get(i).get(1);

List<Node> list = map.getOrDefault(src,**new** ArrayList<>());

list.add(**new** Node(values[i],dest));

map.put(src,list);

List<Node> list2 = map.getOrDefault(dest,**new** ArrayList<>());

list2.add(**new** Node((1.0/values[i]),src));

map.put(dest,list2);

}

**return** map;

}

}

### Time and space complexity

Time – same as dfs

# Backtracking

## 1. Rat in a Maze (Maze Solver)

### Problem

A Maze is given as N\*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.  
In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

**You have to print the path if solution exists else return false.**

Gray blocks are dead ends (value = 0).

[](https://www.geeksforgeeks.org/wp-content/uploads/ratinmaze_filled11.png)

Following is binary matrix representation of the above maze.

{1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

Following is a maze with highlighted solution path.

[](https://www.geeksforgeeks.org/wp-content/uploads/ratinmaze_filled_path1.png)

Following is the solution matrix (output of program) for the above input matrx.

{1, 0, 0, 0}

{1, 1, 0, 0}

{0, 1, 0, 0}

{0, 1, 1, 1}

All enteries in solution path are marked as 1.

### Reference

LEETCODE, GITHUB, GeeksForGeeks

### Approach

* 1. We will call maze solver method recursively and if we reach n-1, n-1 return true else if we reach out of bounds or maze[i][j] == 0 return false.
  2. At any point if we reach n-1, n-1. We will stop calling recursive methods.
  3. To print path, we take extra 2-D array and will add 1 to current index I,j in solution array. After going via both paths i.e. i+1, j and I, j+1 we got false then reset sol[i][j]=0 as we need to backtrack.

### Solution

**public** **boolean** solveMaze(**int** maze[][], **int**[][] sol) {

**boolean** res = solveMazeUtil(maze, 0, 0, sol);

**if** (res) {

System.***out***.println("solution exists : " + res);

printMatrix(sol);

}

return res;

}

**private** **boolean** solveMazeUtil(**int** maze[][], **int** i, **int** j, **int**[][] sol) {

**if** (i == maze.length - 1 && j == maze.length - 1) {

sol[i][j] = 1;

**return** **true**;

}

**if** (i >= maze.length || j >= maze.length || maze[i][j] == 0) {

**return** **false**;

}

sol[i][j] = 1;//assume we are moving to a valid path

**boolean** res = solveMazeUtil(maze, i+1, j, sol)||solveMazeUtil(maze, i, j+1, sol);

**if** (!res) { // i.e. this path is not valid path

sol[i][j] = 0;

}

**return** res;

}

**private** **void** printMatrix(**int**[][] arr) {

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

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

System.***out***.print(" " + arr[i][j]);

}

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

}

}

### Time and space complexity

* **Time Complexity:** O(2^(n^2)).  
  The recursion can run upperbound 2^(n^2) times.
* **Space Complexity:** O(n^2).  
  Output matrix is required so an extra space of size n\*n is needed.

# Disjoint sets

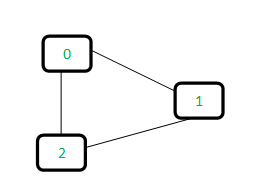
For detail clarification go abdul bari video youtube. ( In that case you can skip below theory)

A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

**Find**: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

**Union**: Join two subsets into a single subset.

We can keep track of the subsets in a 1D array, let’s call it parent[].

Let us consider the following graph:  
  
For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

0 1 2

-1 -1 -1

Now process all edges one by one.

Edge 0-1: Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

0 1 2 <----- 1 is made parent of 0 (1 is now representative of subset {0, 1})

1 -1 -1

Edge 1-2: 1 is in subset 1 and 2 is in subset 2. So, take union.

0 1 2 <----- 2 is made parent of 1 (2 is now representative of subset {0, 1, 2})

1 2 -1

Edge 0-2: 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.

How subset of 0 is same as 2?  
0->1->2 // 1 is parent of 0 and 2 is parent of 1

**(path compression techinique) ->**

To perform better we can cache parent of every node for further use. E.g. if parent of 2 is 1 and 1 is 0. We will store parent of 2 as 0. So that next time when find is called we don’t need to traverse entire tree

**(union by rank)**

Similarly while deciding PARENT IN UNION we take highest weight value as parent. And we update weight every time we update parent. By weight we means here no of childs that node has.

## 1. Detect cycle in un-directed graph

### Problem

Given a 2-D array of size n\*2 where each row has 2 fields which represent edges between vertex. Check if there exists a cycle in a graph.

### Reference

GITHUB

### Approach

1. To detect cycle we can use topological sort also.
2. Union-Find Algorithm can also be used to check whether an undirected graph contains cycle or not.
3. For every edge u,v call find method to check parent[u] and parent[v]. if they are same it means cycle exists. Else perform union. For better performance we will use union by rank and path compression.

### Solution

**public** **boolean** hasCycle(**int** numCourses, **int**[][] prerequisites) {

**int**[] parent = **new** **int**[numCourses];

// make set

**for** (**int** i = 0; i < numCourses; i++) {

parent[i] = -1;

}

// check cycle

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

**int** parentX = find(parent, prerequisites[i][0]);

**int** parentY = find(parent, prerequisites[i][1]);

**if** (parentX == parentY) {

**return** **true**;

}

union(parent, parentX, parentY);

}

**return** **false**;

}

**private** **int** find(**int**[] parent, **int** x) {

**int** res = x;

**while** (parent[x] > -1) {

x = parent[x];

}

**if** (x != res) {

parent[res] = x;

}

**return** x;

}

**private** **void** union(**int**[] parent, **int** x, **int** y) {

**if** (Math.*abs*(parent[x]) < Math.*abs*(parent[y])) {

**int** res = parent[x];

parent[x] = y;

parent[y] += res;

} **else** {

**int** res = parent[y];

parent[y] = x;

parent[x] += res;

}

}

### Time and space complexity

1. **Time Complexity:** O(logn).
2. **Space Complexity:** O(n).  
   where n is no of vertex.

# Regex

^ - indicates that regex must start with.

$ - indicates it should end.

| - used for OR.

[0-9] – means single digit in range 0 to 9.

[0-9a-fA-F] – means one digit which can be 0 to 9 or a to z or A-Z

{1,4} – repeat regex 1 to 4 times.

2[1-9] – means 2 followed by any digit from 1 to 9.

\* - means 0 to many

+ - means 1 to many

[1-9]{7} – repeat regex 7 times.

**Example -**

**import** java.util.regex.Pattern

**public** String validIPAddress(String ip) {

String ipv4 = "([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])";

String ipv6 = "([0-9a-fA-F]{1,4})";

Pattern pipv4 = Pattern.*compile*("^(" + ipv4 + "\\.){3}" + ipv4 + "$");

Pattern pipv6 = Pattern.*compile*("^(" + ipv6 + "\\:){7}" + ipv6 + "$");

**if** (pipv4.matcher(ip).matches()) {

**return** "IPv4";

} **else** **if** (pipv6.matcher(ip).matches()) {

**return** "IPv6";

}

**return** "Neither";

}