**BASIC CODING PREPARATION**

**By Jani Ahamed Habeeb Mohamed**

Table of Contents

[1. Problems 11](#_Toc159438030)

[1.1 Minimum moves to spread stones over grid 11](#_Toc159438031)

[1.2 LRU Cache 13](#_Toc159438032)

[1.3 LFU Cache 17](#_Toc159438033)

[1.4 IP Address, Subnet, Network Address, Host 21](#_Toc159438034)

[1.5 IP address generation 22](#_Toc159438035)

[1.6 Validity Of IP Address 24](#_Toc159438036)

[1.7 Packet Parser 26](#_Toc159438037)

[1.8 Store IP address 28](#_Toc159438038)

[1.9 Store MAC address 29](#_Toc159438039)

[1.10 Basic Firewall Implementation 30](#_Toc159438040)

[1.11 Rate Limiter 32](#_Toc159438041)

[1.12 Longest Prefix Match - IP 34](#_Toc159438042)

[1.13 Trie 35](#_Toc159438043)

[1.14 Calculator using Tree 38](#_Toc159438044)

[1.15 Balanced Binary Tree 39](#_Toc159438045)

[1.16 Time Needed To Inform All Employees 41](#_Toc159438046)

[1.17 Distribute Coins In Binary Tree 43](#_Toc159438047)

[1.18 Find Original Array From Doubled Array 45](#_Toc159438048)

[1.19 Sum Of SubArray Minimums 46](#_Toc159438049)

[1.20 Valid Parantheses String 49](#_Toc159438050)

[1.21 Asteroid Collission 50](#_Toc159438051)

[1.22 ReOrganize String (any two adjacent characters are not same) 52](#_Toc159438052)

[1.23 Generate All Sub Arrays 53](#_Toc159438053)

[1.24 Bit manipulations (Set/Reset/Get Bits) 55](#_Toc159438054)

[1.25 Decode Ways For String 56](#_Toc159438055)

[1.26 Multiply Two Strings 57](#_Toc159438056)

[1.27 Buildings With An Ocean View 59](#_Toc159438057)

[1.28 Bankers Algorithm 60](#_Toc159438058)

[1.29 Bucket Sort 62](#_Toc159438059)

[1.30 Merge K Sorted arrays – Heapify 63](#_Toc159438060)

[1.31 Merge K sorted arrays – Normal Way 66](#_Toc159438061)

[1.32 Linked List Palindrome 68](#_Toc159438062)

[1.33 Interleaving String 70](#_Toc159438063)

[1.34 Reverse Linked List Nodes In K Groups 72](#_Toc159438064)

[1.35 4 Sum 74](#_Toc159438065)

[1.36 Largest Rectangle In Histogram 76](#_Toc159438066)

[1.37 Minimum Path Sum 77](#_Toc159438067)

[1.38 House Robber II 79](#_Toc159438068)

[1.39 Binary Tree Vertical Order Traversal 80](#_Toc159438069)

[1.40 Course Schedule 84](#_Toc159438070)

[1.41 Word Break 87](#_Toc159438071)

[1.42 Longest Valid Parantheses 89](#_Toc159438072)

[1.43 Edit Distance (Minimum Operation to convert word1 to word2) 91](#_Toc159438073)

[1.44 SubSets II 92](#_Toc159438074)

[1.45 Nth Digit 94](#_Toc159438075)

[1.46 Minimize The Maximum Difference Of Pairs 95](#_Toc159438076)

[1.47 Reverse Words In String II 96](#_Toc159438077)

[1.48 Number Of Flowers In Full Bloom 97](#_Toc159438078)

[1.49 Restore IP addresses 100](#_Toc159438079)

[1.50 SubArray Sum Divisible By K 102](#_Toc159438080)

[1.51 Find Missing Observations 104](#_Toc159438081)

[1.52 Largest Palindromic Number 106](#_Toc159438082)

[1.53 Basic Calculator I 107](#_Toc159438083)

[1.54 Basic Calculator II 109](#_Toc159438084)

[1.55 Basic Calculator III 110](#_Toc159438085)

[1.56 Inorder Successor of Binary Search Tree 111](#_Toc159438086)

[1.57 Regular Expression Matching 113](#_Toc159438087)

[1.58 Remove All Adjacent Duplicates In String 115](#_Toc159438088)

[1.59 Fibonacci Series 116](#_Toc159438089)

[1.60 Path With Minimum Effort (Hiking heights) 118](#_Toc159438090)

[1.61 Longest Cycle In Graph 123](#_Toc159438091)

[1.62 Maximum Value Of K Coins From Piles 124](#_Toc159438092)

[1.63 Flip Equivalent Binary Trees 126](#_Toc159438093)

[1.64 Maximum Length Of a Concatenated String With Unique Characters 131](#_Toc159438094)

[1.65 Buddy Strings 132](#_Toc159438095)

[1.66 Robot Room Cleaner 134](#_Toc159438096)

[1.67 Count Unique Characters Of All Substrings Of A Given String 135](#_Toc159438097)

[1.68 Minimum Fuel Cost To Report To The Capital 136](#_Toc159438098)

[1.69 Remove Duplicate Letters(lexicographical small) 139](#_Toc159438099)

[1.70 Score of Parantheses 141](#_Toc159438100)

[1.71 Maximum Number Of Balloons 142](#_Toc159438101)

[1.72 Buy Two Chocolates 143](#_Toc159438102)

[1.73 Permutations 144](#_Toc159438103)

[1.74 Combinations 145](#_Toc159438104)

[1.75 Remove K Digits (return smallest possible integer) 148](#_Toc159438105)

[1.76 Car Fleet (arrive at destination) 149](#_Toc159438106)

[1.77 Match Sticks To Square 150](#_Toc159438107)

[1.78 132 Pattern 152](#_Toc159438108)

[1.79 Word Search II 155](#_Toc159438109)

[1.80 Merge Intervals 158](#_Toc159438110)

[1.81 Merge Sorted Array 161](#_Toc159438111)

[1.82 Serialize and Deserialize Binary Tree 163](#_Toc159438112)

[1.83 Bulb Switcher 167](#_Toc159438113)

[1.84 Sign Of The Product Of An Array 168](#_Toc159438114)

[1.85 Excel Sheet Column Title 169](#_Toc159438115)

[1.86 Remove All Adjacent Duplicates In String (K duplicate removal) 170](#_Toc159438116)

[1.87 Number Of Islands 172](#_Toc159438117)

[1.88 Copy List With Random Pointer 174](#_Toc159438118)

[1.89 Two Sum 177](#_Toc159438119)

[1.90 Flood Fill 180](#_Toc159438120)

[1.91 Maximum SubArray 184](#_Toc159438121)

[1.92 Sort Colors 185](#_Toc159438122)

[1.93 Median Of Two Sorted Arrays 187](#_Toc159438123)

[1.94 Root Equals Sum Of Children 188](#_Toc159438124)

[1.95 Generate Parantheses 189](#_Toc159438125)

[1.96 Meeting Rooms II – Minimum Rooms Required 191](#_Toc159438126)

[1.97 Reverse Linked List 193](#_Toc159438127)

[1.98 Implement Queue Using Stack 196](#_Toc159438128)

[1.99 Implement Stack Using Queue 199](#_Toc159438129)

[1.100 Trapping Rain Water 202](#_Toc159438130)

[1.101 Longest SubString Without Repeating Characters 203](#_Toc159438131)

[1.102 SubArray Sum Equals K 206](#_Toc159438132)

[1.103 Search In A Rotated Sorted Array 207](#_Toc159438133)

[1.104 Top K elements 209](#_Toc159438134)

[1.105 Top K Frequent Elements 212](#_Toc159438135)

[1.106 Rotate Image 214](#_Toc159438136)

[1.107 Gas Station 217](#_Toc159438137)

[1.108 Delete Node in BST 219](#_Toc159438138)

[1.109 Word Search 222](#_Toc159438139)

[1.110 Integer To Roman 225](#_Toc159438140)

[1.111 Merge String Alternately 228](#_Toc159438141)

[1.112 Reverse Linked List II (In between from one position to another) 229](#_Toc159438142)

[1.113 Implement Trie – Prefix Tree 232](#_Toc159438143)

[1.114 Odd Even Linked List 235](#_Toc159438144)

[1.115 Best Time To Buy And Sell Stock (Single day to buy and different day to sell once) 237](#_Toc159438145)

[1.116 Longest Consecutive Sequence 239](#_Toc159438146)

[1.117 Spiral Matrix 241](#_Toc159438147)

[1.118 Populating Next Right Pointer In Each Node 243](#_Toc159438148)

[1.119 String Compression 246](#_Toc159438149)

[1.120 Construct Binary Tree From Inorder and post Order Traversal 249](#_Toc159438150)

[1.121 Valid Sudoku 251](#_Toc159438151)

[1.122 Path Sum 255](#_Toc159438152)

[1.123 Design Add and Search Words DataStructure 258](#_Toc159438153)

[1.124 Sudoku Solver 261](#_Toc159438154)

[1.125 Max Area Of Island 264](#_Toc159438155)

[1.126 Valid Parantheses 267](#_Toc159438156)

[1.127 Merge Two Sorted Lists 268](#_Toc159438157)

[1.128 Container With Most Water 271](#_Toc159438158)

[1.129 Pascals Triangle 273](#_Toc159438159)

[1.130 Valid Anagram 275](#_Toc159438160)

[1.131 Unique Paths 276](#_Toc159438161)

[1.132 Longest Increasing Subsequence 278](#_Toc159438162)

[1.133 Group Anagrams 279](#_Toc159438163)

[1.134 Longest Palindromic SubString 284](#_Toc159438164)

[1.135 Three Sum 286](#_Toc159438165)

[1.136 Single Element In A Sorted Array 290](#_Toc159438166)

[1.137 Letter Combinations Of A Phone Number 291](#_Toc159438167)

[1.138 Find The Index Of The First Occurrence In The String (strstr – needle in haystack) 295](#_Toc159438168)

[1.139 Best Time to Buy And Sell Stock II (Buy Sell Any Times) 296](#_Toc159438169)

[1.140 Intersection Of Two Arrays 297](#_Toc159438170)

[1.141 Climbing Stairs 298](#_Toc159438171)

[1.142 Convert Sorted Array To Binary Search Tree 299](#_Toc159438172)

[1.143 Kth Largest Element In Array 301](#_Toc159438173)

[1.144 Kth Smallest Element In Array 303](#_Toc159438174)

[1.145 Binary Tree ZigZag Level Order Traversal 304](#_Toc159438175)

[1.146 Find Minimum In Rotated Sorted Array 308](#_Toc159438176)

[1.147 Binary Search 310](#_Toc159438177)

[1.148 Construct Binary Tree From PreOrder And Inorder Traversal 311](#_Toc159438178)

[1.149 Remove Nth Node From End Of List 313](#_Toc159438179)

[1.150 Jump Game 316](#_Toc159438180)

[1.151 Insert Delete GetRandom O(1) 317](#_Toc159438181)

[1.152 Daily Temperatures 320](#_Toc159438182)

[1.153 Reverse Integer 321](#_Toc159438183)

[1.154 Symmetric Tree 322](#_Toc159438184)

[1.155 Lowest Common Ancestor Of A binary tree 325](#_Toc159438185)

[1.156 Merge K Sorted Lists 328](#_Toc159438186)

[1.157 SubSets 330](#_Toc159438187)

[1.158 Can Place Flowers 332](#_Toc159438188)

[1.159 Roman To Integer 334](#_Toc159438189)

[1.160 Find The Duplicate Number 336](#_Toc159438190)

[1.161 Product Of Array Except Self 337](#_Toc159438191)

[1.162 Coin Change 338](#_Toc159438192)

[1.163 Contains Duplicate 340](#_Toc159438193)

[1.164 Search A 2D Matrix 342](#_Toc159438194)

[1.165 Validate Binary Search Tree 343](#_Toc159438195)

[1.166 Plus One 345](#_Toc159438196)

[1.167 Majority Element 347](#_Toc159438197)

[1.168 Valid Palindrome 348](#_Toc159438198)

[1.169 House Robber 349](#_Toc159438199)

[1.170 Next Greater Element 350](#_Toc159438200)

[1.171 Shuffle An Array 351](#_Toc159438201)

[1.172 MinStack 354](#_Toc159438202)

[1.173 Sum Of Two Integers 357](#_Toc159438203)

[1.174 Maximum Product SubArray 357](#_Toc159438204)

[1.175 Number Of 1 Bits 359](#_Toc159438205)

[1.176 Counting Bits 360](#_Toc159438206)

[1.177 Missing Number 361](#_Toc159438207)

[1.178 Reverse Bits 362](#_Toc159438208)

[1.179 Detect Cycle in Linked List 363](#_Toc159438209)

[1.180 Reorder List 365](#_Toc159438210)

[1.181 Set Matrix Zeros 368](#_Toc159438211)

[1.182 Insert Intervals 372](#_Toc159438212)

[1.183 Non Overlapping Intervals 376](#_Toc159438213)

[1.184 Meeting Rooms (Can attend all meetings) 378](#_Toc159438214)

[1.185 Longest Repeating Character Replacement 379](#_Toc159438215)

[1.186 Minimum Window SubString 381](#_Toc159438216)

[1.187 Palindromic Substrings 384](#_Toc159438217)

[1.188 Encode And Decode Strings 385](#_Toc159438218)

[1.189 Longest Common Subsequence 388](#_Toc159438219)

[1.190 Word Break 391](#_Toc159438220)

[1.191 Combination Sum IV 392](#_Toc159438221)

[1.192 House Robber II 392](#_Toc159438222)

[1.193 Decode Ways 394](#_Toc159438223)

[1.194 Find Median From Data Stream 395](#_Toc159438224)

[1.195 Maximum Depth Of Binary Tree 397](#_Toc159438225)

[1.196 Same Tree 398](#_Toc159438226)

[1.197 Invert Binary Tree 400](#_Toc159438227)

[1.198 Binary Tree Maximum Path Sum 402](#_Toc159438228)

[1.199 Binary Tree Level Order Traversal 405](#_Toc159438229)

[1.200 Kth Smallest Element In BST 408](#_Toc159438230)

[1.201 SubTree Of Another Tree 411](#_Toc159438231)

[1.202 Word Search II 414](#_Toc159438232)

[1.203 Clone Graph 416](#_Toc159438233)

[1.204 Course Schedule 421](#_Toc159438234)

[1.205 Pacific Atlantic Water Flow 423](#_Toc159438235)

[1.206 Alien Dictionary 426](#_Toc159438236)

[1.207 Graph Valid Tree 428](#_Toc159438237)

[1.208 Number Of Connected Components In an Undirected Graph 430](#_Toc159438238)

[1.209 Longest Prefix Match – Trie (Store and Query IP address efficiently) 431](#_Toc159438239)

[1.210 UDP Socket Communication Client and Server 435](#_Toc159438240)

[1.211 TCP Socket Communication Client and Server 438](#_Toc159438241)

[1.212 Shared Memory 439](#_Toc159438242)

[1.213 Message Queues 440](#_Toc159438243)

[1.214 Dequeue 440](#_Toc159438244)

[1.215 Regex Search Target String 444](#_Toc159438245)

[1.216 Quick Sort 446](#_Toc159438246)

[1.217 Find repeating elements from two linked lists 447](#_Toc159438247)

[1.218 Merge Sort 449](#_Toc159438248)

[1.219 Bloom Filter 449](#_Toc159438249)

[1.220 Hash Table 451](#_Toc159438250)

[1.221 Endianess 453](#_Toc159438251)

[1.222 MALLOC Implementation 454](#_Toc159438252)

[1.223 Single Linked List 457](#_Toc159438253)

[1.224 Double Linked List 460](#_Toc159438254)

[1.225 Stack 464](#_Toc159438255)

[1.226 Queue 466](#_Toc159438256)

[1.227 Binary Search Tree 467](#_Toc159438257)

[1.228 Trie Complete Code 471](#_Toc159438258)

[1.229 Graph Shortest Path 474](#_Toc159438259)

[1.230 Semaphore 477](#_Toc159438260)

[1.231 Mutex 479](#_Toc159438261)

[1.232 Blocking Queue 480](#_Toc159438262)

[1.233 String to Integer(atoi) 482](#_Toc159438263)

[1.234 Add Two Numbers 483](#_Toc159438264)

[1.235 Next Permutation 484](#_Toc159438265)

[1.236 First Missing Positive 486](#_Toc159438266)

[1.237 Permutations 487](#_Toc159438267)

[1.238 Remove Duplicates From Sorted List 488](#_Toc159438268)

[1.239 Binary Tree InOrder Traversal 490](#_Toc159438269)

[1.240 Binary Tree ZigZag Level Order Traversal 491](#_Toc159438270)

[1.241 Flatten Binary Tree To Linked List 492](#_Toc159438271)

[1.242 Binary Tree Post Order Traversal 494](#_Toc159438272)

[1.243 Binary Tree Pre Order Traversal 495](#_Toc159438273)

[1.244 Excel Sheet Column Number 496](#_Toc159438274)

[1.245 Ugly Number 496](#_Toc159438275)

[1.246 Game Of Life 497](#_Toc159438276)

[1.247 Minimum Number Of Arrows To Burst Balloons 498](#_Toc159438277)

[1.248 Exclusive Time Of Functions 499](#_Toc159438278)

[1.249 Top K Frequent Words 500](#_Toc159438279)

[1.250 Reverse String In Place 500](#_Toc159438280)

[1.251 Bitwise AND Range of Numbers 501](#_Toc159438281)

[1.252 Walking Robot Simulation 502](#_Toc159438282)

[1.253 Cousins In Binary Tree 503](#_Toc159438283)

[1.254 Maximum Equal Frequency 505](#_Toc159438284)

[1.255 Packet Parser Multithreading 506](#_Toc159438285)

[1.256 Thread Pool 510](#_Toc159438286)

[1.257 Memory Pool 512](#_Toc159438287)

[1.258 DPDK 514](#_Toc159438288)

[1.259 Attack Packet Detection With PCAP 518](#_Toc159438289)

[1.260 Attack Packet Context Generation - HTTP 519](#_Toc159438290)

[1.261 Breadth First Search(BFS) Of Binary Tree 521](#_Toc159438291)

[1.262 Depth First Search(DFS) Of Binary Tree 522](#_Toc159438292)

[1.263 Breadth First Search(BFS) Of Graph 523](#_Toc159438293)

[1.264 Depth First Search(DFS) Of Graph 525](#_Toc159438294)

[1.265 Shortest Path In Graph 527](#_Toc159438295)

[1.266 Sliding Window Maximum(Maximum element in a window of size k in linear time) 529](#_Toc159438296)

[1.267 Throughput Load 531](#_Toc159438297)

[1.268 Middle Element Of Linked List 532](#_Toc159438298)

[1.269 Build Process Tree 533](#_Toc159438299)

[1.270 Construct Tree From Data 535](#_Toc159438300)

[1.271 Peak Element 536](#_Toc159438301)

[1.272 T2 subtree of T1 537](#_Toc159438302)

[1.273 Minimum Swaps Required To Convert Min Heap to BST 538](#_Toc159438303)

[1.274 Check If Two Files Are Identical 539](#_Toc159438304)

[1.275 Git Commit History 540](#_Toc159438305)

[1.276 Recursive Links In Web page 541](#_Toc159438306)

[1.277 Left Most Nodes 544](#_Toc159438307)

[1.278 Recursive Print Linked List 545](#_Toc159438308)

[1.279 Minimum Distance Between Two Elements 546](#_Toc159438309)

[1.280 Unique Elements 546](#_Toc159438310)

[1.281 Square A Sorted Array 547](#_Toc159438311)

[1.282 NTP Logger 550](#_Toc159438312)

[1.283 Job Scheduler 551](#_Toc159438313)

[1.284 Course Schedule II 553](#_Toc159438314)

[1.285 Cycle/Loop In Linked List 555](#_Toc159438315)

[1.286 Version Sorting 557](#_Toc159438316)

[1.287 Reverse Linked List Using Stack 557](#_Toc159438317)

[1.288 Hexadecimal to decimal 559](#_Toc159438318)

[1.289 Decimal To Hexadecimal 560](#_Toc159438319)

[1.290 Absolute Path From Relative Path 561](#_Toc159438320)

[1.291 LockLess programming 562](#_Toc159438321)

[1.292 IOCTL 562](#_Toc159438322)

[1.293 I2C 565](#_Toc159438323)

[1.294 Large File Word Frequency 567](#_Toc159438324)

[1.295 Compressed String 569](#_Toc159438325)

[1.296 Remove Node from Linked List 569](#_Toc159438326)

[1.297 DOS attack detection from file 572](#_Toc159438327)

[1.298 TCAM 573](#_Toc159438328)

[1.299 Spin Locks 575](#_Toc159438329)

[1.300 Synchronization Primitives 580](#_Toc159438330)

[1.301 RPC (Remote Procedure Call) 581](#_Toc159438331)

[1.301.1.1.1 Step1  : Specify the protocol for client server communication 581](#_Toc159438332)

[1.301.1.1.2 Step 2 – Develop the client program 584](#_Toc159438333)

[1.301.1.1.3 Step 3 – Develop the server program 587](#_Toc159438334)

[1.302 Ether Types 596](#_Toc159438335)

[1.303 Maximum Length Of A Concatenated String With Unique Characters 597](#_Toc159438336)

# Problems

## Minimum moves to spread stones over grid

You are given a **0-indexed** 2D integer matrix grid of size 3 \* 3, representing the number of stones in each cell. The grid contains exactly 9 stones, and there can be **multiple** stones in a single cell.

In one move, you can move a single stone from its current cell to any other cell if the two cells share a side.

Return the ***minimum number of moves*** required to place one stone in each cell.

**Example 1:**

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

**Output:** 3

**Explanation:** One possible sequence of moves to place one stone in each cell is:

1- Move one stone from cell (2,1) to cell (2,2).

2- Move one stone from cell (2,2) to cell (1,2).

3- Move one stone from cell (1,2) to cell (0,2).

In total, it takes 3 moves to place one stone in each cell of the grid.

It can be shown that 3 is the minimum number of moves required to place one stone in each cell.

**Example 2:**

**Input:** grid = [[1,3,0],[1,0,0],[1,0,3]]

**Output:** 4

**Explanation:** One possible sequence of moves to place one stone in each cell is:

1- Move one stone from cell (0,1) to cell (0,2).

2- Move one stone from cell (0,1) to cell (1,1).

3- Move one stone from cell (2,2) to cell (1,2).

4- Move one stone from cell (2,2) to cell (2,1).

In total, it takes 4 moves to place one stone in each cell of the grid.

It can be shown that 4 is the minimum number of moves required to place one stone in each cell.

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX\_SIZE 3

int min(int a, int b) {

    return a < b ? a : b;

}

int minimumMoves(int\*\* grid, int gridSize, int\* gridColSize) {

    int result = INT\_MAX;

    int distance = 0;

    for (int i = 0; i < MAX\_SIZE; i++) {

        for (int j = 0; j < MAX\_SIZE; j++) {

            if (grid[i][j]) continue;

            for (int a = 0; a < MAX\_SIZE; a++) {

                for (int b = 0; b < MAX\_SIZE; b++) {

                    if (grid[a][b] < 2) continue;

                    int distance = abs(i-a) + abs(j-b);

                    grid[a][b]--;

                    grid[i][j]++;

                    distance += minimumMoves(grid, gridSize, gridColSize);

                    grid[a][b]++;

                    grid[i][j]--;

                    result = min(distance,result);

                }

            }

        }

    }

    return (result == INT\_MAX)?0:result;

}

int main() {

    // Example grid

    int gridSize = MAX\_SIZE;

    int gridColSize[MAX\_SIZE] = {MAX\_SIZE, MAX\_SIZE, MAX\_SIZE};

    int \*\*grid = (int \*\*)malloc(MAX\_SIZE \* sizeof(int \*));

    for (int i = 0; i < MAX\_SIZE; i++) {

        grid[i] = (int \*)malloc(MAX\_SIZE \* sizeof(int));

    }

    // Initialize the grid with sample values

    // Example: {1, 0, 2}

    //          {0, 2, 0}

    //          {1, 0, 0}

    grid[0][0] = 1;

    grid[0][1] = 1;

    grid[0][2] = 0;

    grid[1][0] = 1;

    grid[1][1] = 1;

    grid[1][2] = 1;

    grid[2][0] = 1;

    grid[2][1] = 2;

    grid[2][2] = 1;

    // Call the function and print the result

    int moves = minimumMoves(grid, gridSize, gridColSize);

    printf("Minimum moves required: %d\n", moves);

    grid[0][0] = 1;

    grid[0][1] = 3;

    grid[0][2] = 0;

    grid[1][0] = 1;

    grid[1][1] = 0;

    grid[1][2] = 0;

    grid[2][0] = 1;

    grid[2][1] = 0;

    grid[2][2] = 3;

    // Call the function and print the result

    moves = minimumMoves(grid, gridSize, gridColSize);

    printf("Minimum moves required: %d\n", moves);

    // Free memory allocated for the grid

    for (int i = 0; i < MAX\_SIZE; i++) {

        free(grid[i]);

    }

    free(grid);

    return 0;

}

Output: 3 and 4

1- Move one stone from cell (2,1) to cell (2,2).

2- Move one stone from cell (2,2) to cell (1,2).

3- Move one stone from cell (1,2) to cell (0,2).

1- Move one stone from cell (0,1) to cell (0,2).

2- Move one stone from cell (0,1) to cell (1,1).

3- Move one stone from cell (2,2) to cell (1,2).

4- Move one stone from cell (2,2) to cell (2,1).

Time/Space Complexity:   
*O*(*3^n*)/O(n)

## LRU Cache

Design a data structure that follows the constraints of a [**Least Recently Used (LRU) cache**](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU).

Implement the LRUCache class:

* LRUCache(int capacity) Initialize the LRU cache with **positive** size capacity.
* int get(int key) Return the value of the key if the key exists, otherwise return -1.
* void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, **evict** the least recently used key.

The functions get and put must each run in O(1) average time complexity.

**Example 1:**

**Input**

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]

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

**Output**

[null, null, null, 1, null, -1, null, -1, 3, 4]

**Explanation**

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1); // cache is {1=1}

lRUCache.put(2, 2); // cache is {1=1, 2=2}

lRUCache.get(1); // return 1

lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}

lRUCache.get(2); // returns -1 (not found)

lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}

lRUCache.get(1); // return -1 (not found)

lRUCache.get(3); // return 3

lRUCache.get(4); // return 4

**C :**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_KEYS 10000

// Structure of a Node in the LRUCache

typedef struct Node {

    int key;

    int value;

    struct Node\* next;

    struct Node\* prev;

} Node;

// Structure of the LRUCache

typedef struct {

    int capacity;

    int size;

    Node\* head;

    Node\* tail;

    Node cache[MAX\_KEYS]; // Static cache array for quick lookups

} LRUCache;

// Function to create an LRUCache

LRUCache\* lRUCacheCreate(int capacity) {

    LRUCache\* obj = (LRUCache\*)malloc(sizeof(LRUCache));

    obj->capacity = capacity;

    obj->size = 0;

    obj->head = NULL;

    obj->tail = NULL;

    // Initialize the cache array

    for (int i = 0; i < MAX\_KEYS; ++i) {

        obj->cache[i].key = -1; // Mark unused entries with -1

        obj->cache[i].next = NULL;

        obj->cache[i].prev = NULL;

    }

    return obj;

}

// Function to get the value of a key from the LRUCache

int lRUCacheGet(LRUCache\* obj, int key) {

    if (obj->cache[key].key == -1) {

        return -1; // Key not found

    }

    // Move the accessed node to the front

    Node\* current = &obj->cache[key];

    if (current != obj->head) {

        if (current == obj->tail) {

            obj->tail = current->prev;

        } else {

            current->next->prev = current->prev;

        }

        current->prev->next = current->next;

        current->next = obj->head;

        current->prev = NULL;

        obj->head->prev = current;

        obj->head = current;

    }

    return current->value;

}

// Function to put a key-value pair into the LRUCache

void lRUCachePut(LRUCache\* obj, int key, int value) {

    if (obj->cache[key].key != -1) {

        // Key exists, update the value and move the node to the front

        obj->cache[key].value = value;

        lRUCacheGet(obj, key); // Move the node to the front

    } else {

        // Key doesn't exist, create a new node and add it to the front

        Node\* newNode = &obj->cache[key];

        newNode->key = key;

        newNode->value = value;

        newNode->next = obj->head;

        newNode->prev = NULL;

        if (obj->head != NULL) {

            obj->head->prev = newNode;

        }

        obj->head = newNode;

        if (obj->tail == NULL) {

            obj->tail = newNode;

        }

        // If the cache is full, remove the least recently used element

        if (obj->size == obj->capacity) {

            Node\* tailPrev = obj->tail->prev;

            obj->cache[obj->tail->key].key = -1; // Mark the entry as unused

            obj->tail = tailPrev;

            if (obj->tail != NULL) {

                obj->tail->next = NULL;

            }

        } else {

            obj->size++;

        }

    }

}

// Function to free the memory used by the LRUCache

void lRUCacheFree(LRUCache\* obj) {

    // No dynamic memory to free in this implementation

    free(obj);

}

// Main function for testing the LRUCache implementation

int main() {

    // Create an LRUCache with a capacity of 2

    LRUCache\* cache = lRUCacheCreate(2);

    // Put some key-value pairs into the cache

    lRUCachePut(cache, 1, 1);

    lRUCachePut(cache, 2, 2);

    // Retrieve and print the value associated with key 1

    printf("Value for key 1: %d\n", lRUCacheGet(cache, 1));

    // Put a new key-value pair into the cache, causing eviction of the least recently used item (key 2)

    lRUCachePut(cache, 3, 3);

    // Attempt to retrieve the value associated with evicted key 2

    printf("Value for key 2 after eviction: %d\n", lRUCacheGet(cache, 2)); // Should print -1

    lRUCachePut(cache, 4, 4);

    printf("Value for key 1 after eviction: %d\n", lRUCacheGet(cache, 1));

    printf("Value for key 3 after eviction: %d\n", lRUCacheGet(cache, 3));

    printf("Value for key 3 after eviction: %d\n", lRUCacheGet(cache, 4));

    // Free the memory used by the LRUCache

    lRUCacheFree(cache);

    return 0;

}

Output:

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1); // cache is {1=1}

lRUCache.put(2, 2); // cache is {1=1, 2=2}

lRUCache.get(1); // return 1

lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}

lRUCache.get(2); // returns -1 (not found)

lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}

lRUCache.get(1); // return -1 (not found)

lRUCache.get(3); // return 3

lRUCache.get(4); // return 4

Time/Space Complexity: O(1)/O(K)

## LFU Cache

#include <stdlib.h>

#include <stdio.h>

// Define the Node struct

typedef struct Node {

    int key, val, cnt;

    struct Node \*next, \*prev;

} Node;

// Define the List struct

typedef struct List {

    int len;

    Node \*head, \*tail;

} List;

// Define the LFUCache struct

typedef struct LFUCache {

    int minFreq, currSize, cacheCapacity;

    struct Node\* addrOfKey[100000]; // Assuming the key range is within 10,0000

    struct List\* freqListMap[100000]; // Assuming the key range is within 10,0000

} LFUCache;

// Create a new Node

Node\* createNode(int key, int value) {

    Node\* newNode = (Node\*)malloc(sizeof(Node));

    newNode->key = key;

    newNode->val = value;

    newNode->cnt = 1;

    newNode->next = NULL;

    newNode->prev = NULL;

    return newNode;

}

// Initialize a new List

List\* createList() {

    List\* newList = (List\*)malloc(sizeof(List));

    newList->head = createNode(0, 0);

    newList->tail = createNode(0, 0);

    newList->head->next = newList->tail;

    newList->tail->prev = newList->head;

    newList->len = 0;

    return newList;

}

// Add a new Node to the List

void addNode(List\* list, Node\* newNode) {

    Node\* temp = list->head->next;

    newNode->next = temp;

    newNode->prev = list->head;

    list->head->next = newNode;

    temp->prev = newNode;

    list->len++;

}

// Delete a Node from the List

void deleteNode(List\* list, Node\* delNode) {

    Node\* delPrev = delNode->prev;

    Node\* delNext = delNode->next;

    delPrev->next = delNext;

    delNext->prev = delPrev;

    list->len--;

}

// LFUCache constructor

LFUCache\* lFUCacheCreate(int capacity) {

    LFUCache\* obj = (LFUCache\*)malloc(sizeof(LFUCache));

    obj->cacheCapacity = capacity;

    obj->minFreq = 0;

    obj->currSize = 0;

    for (int i = 0; i < 100000; i++) {

        obj->addrOfKey[i] = NULL;

        obj->freqListMap[i] = createList();

    }

    return obj;

}

// Get the value associated with the given key

int lFUCacheGet(LFUCache\* obj, int key) {

    if (obj->addrOfKey[key] == NULL) {

        return -1;

    }

    Node\* resNode = obj->addrOfKey[key];

    int res = resNode->val;

    int freq = resNode->cnt;

    // Update the frequency map

    deleteNode(obj->freqListMap[freq], resNode);

    if (obj->minFreq == freq && obj->freqListMap[freq]->len == 0) {

        obj->minFreq++;

    }

    resNode->cnt++;

    addNode(obj->freqListMap[freq + 1], resNode);

    obj->addrOfKey[key] = resNode;

    return res;

}

// Put a key-value pair into the LFUCache

void lFUCachePut(LFUCache\* obj, int key, int value) {

    if (obj->cacheCapacity == 0) {

        return;

    }

    if (obj->addrOfKey[key] != NULL) {

        Node\* existingNode = obj->addrOfKey[key];

        existingNode->val = value;

        int freq = existingNode->cnt;

        deleteNode(obj->freqListMap[freq], existingNode);

        if (obj->minFreq == freq && obj->freqListMap[freq]->len == 0) {

            obj->minFreq++;

        }

        existingNode->cnt++;

        addNode(obj->freqListMap[freq + 1], existingNode);

        obj->addrOfKey[key] = existingNode;

    } else {

        if (obj->currSize == obj->cacheCapacity) {

            List\* list = obj->freqListMap[obj->minFreq];

            Node\* prevNode = list->tail->prev;

            int keyToBeErased = prevNode->key;

            obj->addrOfKey[keyToBeErased] = NULL;

            deleteNode(list, prevNode);

            obj->currSize--;

        }

        obj->currSize++;

        obj->minFreq = 1;

        Node\* newNode = createNode(key, value);

        addNode(obj->freqListMap[obj->minFreq], newNode);

        obj->addrOfKey[key] = newNode;

    }

}

// Free the memory used by the LFUCache

void lFUCacheFree(LFUCache\* obj) {

    for (int i = 0; i < 100000; i++) {

        if (obj->freqListMap[i] != NULL) {

            free(obj->freqListMap[i]->head);

            free(obj->freqListMap[i]->tail);

            free(obj->freqListMap[i]);

        }

    }

    free(obj);

}

int main() {

    LFUCache\* cache = lFUCacheCreate(2);

    lFUCachePut(cache, 1, 1);

    lFUCachePut(cache, 2, 2);

    printf("%d\n", lFUCacheGet(cache, 1)); // returns 1

    lFUCachePut(cache, 3, 3);              // evicts key 2

    printf("%d\n", lFUCacheGet(cache, 2)); // returns -1 (not found)

    printf("%d\n", lFUCacheGet(cache, 3)); // returns 3.

    lFUCachePut(cache, 4, 4);              // evicts key 1.

    printf("%d\n", lFUCacheGet(cache, 1)); // returns -1 (not found)

    printf("%d\n", lFUCacheGet(cache, 3)); // returns 3

    printf("%d\n", lFUCacheGet(cache, 4)); // returns 4

    lFUCacheFree(cache);

    return 0;

}

["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]

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

**Output**

[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

**Explanation**

// cnt(x) = the use counter for key x

// cache=[] will show the last used order for tiebreakers (leftmost element is most recent)

LFUCache lfu = new LFUCache(2);

lfu.put(1, 1); // cache=[1,\_], cnt(1)=1

lfu.put(2, 2); // cache=[2,1], cnt(2)=1, cnt(1)=1

lfu.get(1); // return 1

// cache=[1,2], cnt(2)=1, cnt(1)=2

lfu.put(3, 3); // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.

  // cache=[3,1], cnt(3)=1, cnt(1)=2

lfu.get(2); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,1], cnt(3)=2, cnt(1)=2

lfu.put(4, 4); // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.

// cache=[4,3], cnt(4)=1, cnt(3)=2

lfu.get(1); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,4], cnt(4)=1, cnt(3)=3

lfu.get(4); // return 4

// cache=[4,3], cnt(4)=2, cnt(3)=3

Time/Space Complexity: O(n)/O(1)

## IP Address, Subnet, Network Address, Host

#include <stdio.h>

#include <stdint.h>

#include <netinet/in.h>

#include <arpa/inet.h>

// Function to calculate the network address

struct in\_addr calculateNetworkAddress(struct in\_addr ipAddr, struct in\_addr netmask) {

    struct in\_addr networkAddr;

    networkAddr.s\_addr = ipAddr.s\_addr & netmask.s\_addr;

    return networkAddr;

}

// Function to calculate the broadcast address

struct in\_addr calculateBroadcastAddress(struct in\_addr networkAddr, struct in\_addr netmask) {

    struct in\_addr broadcastAddr;

    printf("%d : %u",~netmask.s\_addr,networkAddr.s\_addr);

    broadcastAddr.s\_addr = networkAddr.s\_addr | ~netmask.s\_addr;

    return broadcastAddr;

}

// Function to calculate the number of usable hosts

uint32\_t calculateNumUsableHosts(uint32\_t numHosts) {

    return numHosts - 2;

}

int main() {

    // IP address and subnet mask

    struct in\_addr ipAddr, netmask;

    // Input IP address

    printf("Enter IP address in dotted decimal notation (e.g., 192.168.1.10): ");

    char ipAddrStr[INET\_ADDRSTRLEN];

    fgets(ipAddrStr, INET\_ADDRSTRLEN, stdin);

    inet\_aton(ipAddrStr, &ipAddr);

    // Input subnet mask

    printf("Enter subnet mask in dotted decimal notation (e.g., 255.255.255.248): ");

    char netmaskStr[INET\_ADDRSTRLEN];

    fgets(netmaskStr, INET\_ADDRSTRLEN, stdin);

    inet\_aton(netmaskStr, &netmask);

    // Calculate network address

    struct in\_addr networkAddr = calculateNetworkAddress(ipAddr, netmask);

    // Calculate broadcast address

    struct in\_addr broadcastAddr = calculateBroadcastAddress(networkAddr, netmask);

    // Calculate number of hosts

    uint32\_t numHosts = ntohl(~netmask.s\_addr) + 1;

    // Calculate number of usable hosts

    uint32\_t numUsableHosts = calculateNumUsableHosts(numHosts);

    // Print the details

    printf("\nIP Address:\t%s\n", inet\_ntoa(ipAddr));

    printf("Network Address:\t%s\n", inet\_ntoa(networkAddr));

    printf("Usable Host IP Range:\t%s - %s\n", inet\_ntoa(networkAddr), inet\_ntoa(broadcastAddr));

    printf("Broadcast Address:\t%s\n", inet\_ntoa(broadcastAddr));

    printf("Total Number of Hosts:\t%u\n", numHosts);

    printf("Number of Usable Hosts:\t%u\n", numUsableHosts);

    printf("Subnet Mask:\t%s\n", inet\_ntoa(netmask));

    printf("IP Class:\tC\n");

    return 0;

}

## IP address generation

#include <stdio.h>

// Function to generate IP addresses in a given network with multiple subnets

void generateIPAddresses(char network[], int numSubnets, int hostsPerSubnet) {

    int i, j;

    // Loop through each subnet

    for (i = 0; i < numSubnets; i++) {

        // Loop through the range of valid host addresses in each subnet

        for (j = 1; j <= hostsPerSubnet; j++) {

            // Print the generated IP addresses

            printf("%s.%d.%d\n", network, i + 1, j);

            // Check if the desired number of addresses is reached

            if (j == hostsPerSubnet) {

                break;

            }

        }

    }

}

int main() {

    char network[] = "192.168";    // Example network address

    int numSubnets = 3;            // Number of subnets

    int hostsPerSubnet = 254;      // Number of hosts per subnet (excluding network and broadcast addresses)

    // Generate IP addresses across multiple subnets

    generateIPAddresses(network, numSubnets, hostsPerSubnet);

    return 0;

}

#include <stdio.h>

#include <stdint.h>

// Function to convert a 32-bit unsigned integer to dotted-decimal IP format

void printIPAddress(uint32\_t ipAddress) {

    printf("%u.%u.%u.%u", (ipAddress >> 24) & 0xFF, (ipAddress >> 16) & 0xFF, (ipAddress >> 8) & 0xFF, ipAddress & 0xFF);

}

// Function to generate and print IP addresses within a given subnet

void generateIPAddresses(uint32\_t ipAddress, uint32\_t subnetMask, int numAddresses) {

    // Calculate the network address

    uint32\_t networkAddress = ipAddress & subnetMask;

    // Calculate the number of hosts in the subnet (excluding network and broadcast addresses)

    int hostsInSubnet = (1 << (32 - \_\_builtin\_popcount(subnetMask)));

    // Print the generated IP addresses

    printf("Network Address: ");

    printIPAddress(networkAddress);

    printf("\nNumber of Hosts in Subnet: %d\n", hostsInSubnet);

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

        if (i >= hostsInSubnet) {

            printf("Exhausted available hosts in subnet.\n");

            break;

        }

        uint32\_t generatedIP = networkAddress + i;

        printf("Generated IP Address %d: ", i);

        printIPAddress(generatedIP);

        printf("\n");

    }

}

int main() {

    uint32\_t ipAddress = 0xC0A80101;  // Example IP address: 192.168.1.1 (in hexadecimal)

    uint32\_t subnetMask = 0xFFFFFF00;  // Example subnet mask: 255.255.255.0 (in hexadecimal)

    int numAddresses = 300;  // Number of IP addresses to generate

    generateIPAddresses(ipAddress, subnetMask, numAddresses);

    return 0;

}

## Validity Of IP Address

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function to check if a string is a valid IP address

int isValidIPAddress(char \*ipAddress) {

    int num, dots = 0;

    int len = strlen(ipAddress);

    // Check for empty string

    if (len == 0) {

        return 0;

    }

    // Count the number of dots in the IP address

    for (int i = 0; i < len; i++) {

        if (ipAddress[i] == '.') {

            dots++;

        }

    }

    // An IP address should have exactly three dots

    if (dots != 3) {

        return 0;

    }

    // Tokenize the string based on dots

    char \*token = strtok(ipAddress, ".");

    // Check each token

    while (token != NULL) {

        // Convert the token to an integer

        num = atoi(token);

        // Check if the integer is between 0 and 255 (inclusive)

        if (num < 0 || num > 255) {

            return 0;

        }

        // Check for leading zeros in each token

        if (num != 0 && token[0] == '0') {

            return 0;

        }

        // Move to the next token

        token = strtok(NULL, ".");

        // If token is NULL and there should be more tokens, return 0

        if (token == NULL && dots != 0) {

            return 0;

        }

    }

    // If everything is fine, return 1 (valid IP address)

    return 1;

}

int main() {

    char ipAddress[20];  // Assuming a maximum length of 20 characters for the IP address

    // Get the input from the user

    printf("Enter an IP address: ");

    scanf("%s", ipAddress);

    // Check the validity of the IP address

    if (isValidIPAddress(ipAddress)) {

        printf("The entered IP address (%s) is valid.\n", ipAddress);

    } else {

        printf("The entered IP address (%s) is not valid.\n", ipAddress);

    }

    return 0;

}

## Packet Parser

#include <stdio.h>

#include <stdint.h>

#include <arpa/inet.h>

// Ethernet header structure

struct EthernetHeader {

    uint8\_t destMAC[6];

    uint8\_t srcMAC[6];

    uint16\_t etherType;

};

// IPv4 header structure

struct IPv4Header {

    uint8\_t versionIHL;

    uint8\_t dscpECN;

    uint16\_t totalLength;

    uint16\_t identification;

    uint16\_t flagsFragmentOffset;

    uint8\_t timeToLive;

    uint8\_t protocol;

    uint16\_t headerChecksum;

    uint32\_t srcIP;

    uint32\_t destIP;

};

// TCP header structure

struct TCPHeader {

    uint16\_t srcPort;

    uint16\_t destPort;

    uint32\_t sequenceNumber;

    uint32\_t ackNumber;

    uint8\_t dataOffsetReserved;

    uint8\_t flags;

    uint16\_t windowSize;

    uint16\_t checksum;

    uint16\_t urgentPointer;

};

void processEthernetFrame(const uint8\_t\* frame, uint16\_t frameLength);

void processIPv4Packet(const uint8\_t\* packet, uint16\_t packetLength);

void processTCPHeader(const uint8\_t\* tcpHeader, uint16\_t tcpHeaderLength);

void processApplicationData(const uint8\_t\* data, uint16\_t dataLength);

void printMACAddress(const uint8\_t\* mac) {

    printf("%02x:%02x:%02x:%02x:%02x:%02x", mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);

}

int main() {

    // Example Ethernet frame containing IPv4, TCP, and application data

    uint8\_t frame[] = {

        // Ethernet header

        0x00, 0x11, 0x22, 0x33, 0x44, 0x55, // Source MAC

        0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, // Destination MAC

        0x08, 0x00,                         // EtherType (IPv4)

        // IPv4 header

        0x45, 0x00, 0x00, 0x3C, 0x12, 0x34, 0x40, 0x00, 0x40, 0x06, 0x00, 0x00,

        0xC0, 0xA8, 0x01, 0x01,             // Source IP: 192.168.1.1

        0xC0, 0xA8, 0x01, 0x02,             // Destination IP: 192.168.1.2

        // TCP header

        0x00, 0x50, 0x00, 0x37, 0xB8, 0xA0, 0x00, 0x00, 0x00, 0x00, 0xA0, 0x02,

        0xFA, 0xF0, 0x00, 0x00, 0x02, 0x04, 0x05, 0xB4, 0x01, 0x03, 0x03, 0x06,

        // Application data (HTTP request)

        0x47, 0x45, 0x54, 0x20, 0x2F, 0x20, 0x48, 0x54, 0x54, 0x50, 0x2F, 0x31,

        0x2E, 0x31, 0x0D, 0x0A, 0x48, 0x6F, 0x73, 0x74, 0x3A, 0x20, 0x77, 0x77,

        // ... (HTTP request continues)

    };

    processEthernetFrame(frame, sizeof(frame));

    return 0;

}

void processEthernetFrame(const uint8\_t\* frame, uint16\_t frameLength) {

    if (frameLength < sizeof(struct EthernetHeader)) {

        printf("Invalid Ethernet frame\n");

        return;

    }

    struct EthernetHeader\* ethHeader = (struct EthernetHeader\*)frame;

    printf("Ethernet Frame - Source MAC: ");

    printMACAddress(ethHeader->srcMAC);

    printf(", Destination MAC: ");

    printMACAddress(ethHeader->destMAC);

    printf(", EtherType: %04x\n", ntohs(ethHeader->etherType));

    // Assuming Ethernet frame contains IPv4 packet

    if (ntohs(ethHeader->etherType) == 0x0800) {

        processIPv4Packet(frame + sizeof(struct EthernetHeader), frameLength - sizeof(struct EthernetHeader));

    } else {

        printf("Unsupported EtherType: %04x\n", ntohs(ethHeader->etherType));

    }

}

void processIPv4Packet(const uint8\_t\* packet, uint16\_t packetLength) {

    if (packetLength < sizeof(struct IPv4Header)) {

        printf("Invalid IPv4 packet\n");

        return;

    }

    struct IPv4Header\* ipv4Header = (struct IPv4Header\*)packet;

    printf("IPv4 Packet - Source IP: %s, Destination IP: %s\n",

           inet\_ntoa(\*(struct in\_addr\*)&ipv4Header->srcIP), inet\_ntoa(\*(struct in\_addr\*)&ipv4Header->destIP));

    // Assuming IPv4 packet contains TCP header

    processTCPHeader(packet + (ipv4Header->versionIHL & 0x0F) \* 4, packetLength - (ipv4Header->versionIHL & 0x0F) \* 4);

}

void processTCPHeader(const uint8\_t\* tcpHeader, uint16\_t tcpHeaderLength) {

    if (tcpHeaderLength < sizeof(struct TCPHeader)) {

        printf("Invalid TCP header\n");

        return;

    }

    struct TCPHeader\* tcpHeaderStruct = (struct TCPHeader\*)tcpHeader;

    printf("TCP Header - Source Port: %u, Destination Port: %u\n",

           ntohs(tcpHeaderStruct->srcPort), ntohs(tcpHeaderStruct->destPort));

    // Assuming TCP header contains application data

    processApplicationData(tcpHeader + (tcpHeaderStruct->dataOffsetReserved >> 4) \* 4, tcpHeaderLength - (tcpHeaderStruct->dataOffsetReserved >> 4) \* 4);

}

void processApplicationData(const uint8\_t\* data, uint16\_t dataLength) {

    // Add code to process application-specific data

    printf("Application Data (%u bytes):\n", dataLength);

    for (uint16\_t i = 0; i < dataLength; ++i) {

        printf("%c", data[i]);

    }

    printf("\n");

}

## Store IP address

#include<stdio.h>

typedef struct ip\_address {

    unsigned char ip[4];

} ip\_addr\_t;

void store\_ip(char \*input\_ip, ip\_addr\_t \*ip) {

    sscanf(input\_ip, "%hhu.%hhu.%hhu.%hhu", &ip->ip[0], &ip->ip[1], &ip->ip[2], &ip->ip[3]);

    printf("\nStored ip is : %hhu.%hhu.%hhu.%hhu",ip->ip[0], ip->ip[1], ip->ip[2], ip->ip[3]);

}

void retrieve\_ip(char \*output\_ip, ip\_addr\_t \*ip) {

    sprintf(output\_ip, "%hhu.%hhu.%hhu.%hhu",ip->ip[0], ip->ip[1], ip->ip[2], ip->ip[3]);

}

void main() {

    char input\_ip[15];

    char output\_ip[15];

    ip\_addr\_t stored\_ip;

    printf("\nEnter an ip address : ");

    scanf("%s", input\_ip);

    printf("\nInput ip address is : %s",input\_ip);

    store\_ip(&input\_ip, &stored\_ip);

    retrieve\_ip(&output\_ip, &stored\_ip);

    printf("\nRetrieved ip address is : %s",output\_ip);

}

## Store MAC address

#include<stdio.h>

typedef struct mac\_address {

    unsigned char mac[6];

} mac\_addr\_t;

void store\_mac(char \*input\_mac, mac\_addr\_t \*mac) {

    printf("\naddress of mac %p",&mac);

    printf("\naddress of mac->mac[0] %p",&mac->mac[0]);

    sscanf(input\_mac, "%hhx:%hhx:%hhx:%hhx:%hhx:%hhx", &mac->mac[0], &mac->mac[1], &mac->mac[2], &mac->mac[3], &mac->mac[4], &mac->mac[5]);

    printf("\nStored mac is : %02x:%02x:%02x:%02x:%02x:%02x",mac->mac[0], mac->mac[1], mac->mac[2], mac->mac[3], mac->mac[4], mac->mac[5]);

}

void retrieve\_mac(char \*output\_mac, mac\_addr\_t \*mac) {

    sprintf(output\_mac, "%hhx:%hhx:%hhx:%hhx:%hhx:%hhx",mac->mac[0], mac->mac[1], mac->mac[2], mac->mac[3], mac->mac[4], mac->mac[5]);

}

void main() {

    char input\_mac[18];

    char output\_mac[18];

    mac\_addr\_t stored\_mac;

    printf("\nEnter a mac address : ");

    scanf("%s", input\_mac);

    printf("\nInput mac address is : %s",input\_mac);

    store\_mac(&input\_mac, &stored\_mac);

    printf("\naddress of stored mac %p",&stored\_mac);

    printf("\naddress of stored mac->mac[0] %p",&stored\_mac.mac[0]);

    retrieve\_mac(&output\_mac, &stored\_mac);

    printf("\nRetrieved mac address is : %s",output\_mac);

}

## Basic Firewall Implementation

#include <stdio.h>

#include <pcap.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_PACKET\_SIZE 1500

// IP address structure

typedef struct {

    unsigned char octet1;

    unsigned char octet2;

    unsigned char octet3;

    unsigned char octet4;

} IPAddress;

// Rule structure

typedef struct {

    IPAddress source\_ip;

    IPAddress destination\_ip;

    int source\_port;

    int destination\_port;

    int allow; // 1 for allow, 0 for deny

} FirewallRule;

// Function to check if a packet matches a firewall rule

int packet\_matches\_rule(const unsigned char\* packet, const FirewallRule\* rule) {

    // Assuming Ethernet frames

    int ethernet\_header\_length = 14;

    // Assuming IPv4 packets

    int ip\_header\_length = (packet[ethernet\_header\_length] & 0x0F) \* 4;

    // Extract source and destination IP addresses

    IPAddress source\_ip = {

        packet[ethernet\_header\_length + 12],

        packet[ethernet\_header\_length + 13],

        packet[ethernet\_header\_length + 14],

        packet[ethernet\_header\_length + 15]

    };

    IPAddress destination\_ip = {

        packet[ethernet\_header\_length + ip\_header\_length + 12],

        packet[ethernet\_header\_length + ip\_header\_length + 13],

        packet[ethernet\_header\_length + ip\_header\_length + 14],

        packet[ethernet\_header\_length + ip\_header\_length + 15]

    };

    // Assuming TCP packets

    int tcp\_header\_length = ((packet[ethernet\_header\_length + ip\_header\_length + 12] & 0xF0) >> 4) \* 4;

    // Extract source and destination ports

    int source\_port = (packet[ethernet\_header\_length + ip\_header\_length + 20] << 8) | packet[ethernet\_header\_length + ip\_header\_length + 21];

    int destination\_port = (packet[ethernet\_header\_length + ip\_header\_length + 22] << 8) | packet[ethernet\_header\_length + ip\_header\_length + 23];

    // Check if the packet matches the rule

    return (memcmp(&source\_ip, &rule->source\_ip, sizeof(IPAddress)) == 0 &&

            memcmp(&destination\_ip, &rule->destination\_ip, sizeof(IPAddress)) == 0 &&

            source\_port == rule->source\_port &&

            destination\_port == rule->destination\_port);

}

// Function to process each packet

void packet\_handler(unsigned char\* user, const struct pcap\_pkthdr\* pkthdr, const unsigned char\* packet) {

    FirewallRule\* rule = (FirewallRule\*)user;

    // Check if the packet matches the rule

    if (packet\_matches\_rule(packet, rule)) {

        if (rule->allow) {

            printf("Allowed packet\n");

        } else {

            printf("Blocked packet\n");

        }

    } else {

        printf("Passing through\n");

    }

}

int main() {

    char errbuf[PCAP\_ERRBUF\_SIZE];

    pcap\_t\* handle;

    // Open the network interface for packet capture

    handle = pcap\_open\_live("your\_network\_interface", MAX\_PACKET\_SIZE, 1, 1000, errbuf);

    if (handle == NULL) {

        fprintf(stderr, "Couldn't open device: %s\n", errbuf);

        return 2;

    }

    // Set a packet filter to capture all traffic

    struct bpf\_program fp;

    char filter\_exp[] = "tcp or udp or icmp";

    if (pcap\_compile(handle, &fp, filter\_exp, 0, PCAP\_NETMASK\_UNKNOWN) == -1) {

        fprintf(stderr, "Couldn't parse filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

        return 2;

    }

    if (pcap\_setfilter(handle, &fp) == -1) {

        fprintf(stderr, "Couldn't install filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

        return 2;

    }

    // Define a firewall rule (Example: Allow TCP traffic from source IP 192.168.1.2 to destination IP 192.168.1.1 on port 80)

    FirewallRule rule;

    rule.source\_ip = (IPAddress){192, 168, 1, 2};

    rule.destination\_ip = (IPAddress){192, 168, 1, 1};

    rule.source\_port = 0; // Any source port

    rule.destination\_port = 80;

    rule.allow = 1; // Allow the specified traffic

    // Start capturing packets

    pcap\_loop(handle, 0, packet\_handler, (unsigned char\*)&rule);

    // Close the handle

    pcap\_close(handle);

    return 0;

}

## Rate Limiter

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define TOKEN\_RATE\_LIMIT 5

#define BUCKET\_CAPACITY 10

struct TokenBucket {

    int tokens;

    pthread\_mutex\_t mutex;

    pthread\_cond\_t tokenAdded;

};

void initTokenBucket(struct TokenBucket\* bucket) {

    bucket->tokens = BUCKET\_CAPACITY;

    pthread\_mutex\_init(&bucket->mutex, NULL);

    pthread\_cond\_init(&bucket->tokenAdded, NULL);

}

void\* tokenRefill(void\* tokenBucket) {

    struct TokenBucket\* bucket = (struct TokenBucket\*)tokenBucket;

    while (1) {

        usleep(1000000 / TOKEN\_RATE\_LIMIT);

        pthread\_mutex\_lock(&bucket->mutex);

        if (bucket->tokens < BUCKET\_CAPACITY) {

            bucket->tokens++;

            printf("Token added: %d\n", bucket->tokens);

            // Signal that a token has been added

            pthread\_cond\_signal(&bucket->tokenAdded);

        }

        pthread\_mutex\_unlock(&bucket->mutex);

    }

}

void processRequest(struct TokenBucket\* bucket, int requestId) {

    pthread\_mutex\_lock(&bucket->mutex);

    // Wait until a token is available

    while (bucket->tokens <= 0) {

        printf("Request %d waiting for token...\n", requestId);

        pthread\_cond\_wait(&bucket->tokenAdded, &bucket->mutex);

    }

    // Token is available, process the request

    bucket->tokens--;

    pthread\_mutex\_unlock(&bucket->mutex);

    printf("Request %d processed\n", requestId);

}

int main() {

    struct TokenBucket tokenBucket;

    initTokenBucket(&tokenBucket);

    pthread\_t refillThread;

    pthread\_create(&refillThread, NULL, tokenRefill, (void\*)&tokenBucket);

    // Simulate requests

    for (int i = 1; i <= 20; i++) {

        processRequest(&tokenBucket, i);

        usleep(200000);

    }

    // Cancel the token refill thread

    pthread\_cancel(refillThread);

    pthread\_join(refillThread, NULL);

    return 0;

}

## Longest Prefix Match - IP

#include <stdio.h>

#include <stdint.h>

#include <arpa/inet.h>

typedef struct {

    char network[20];

    int subnet;

} CIDR;

void findLongestPrefixMatch(CIDR cidrList[], int n, char\* inputIP) {

    uint32\_t input = ntohl(inet\_addr(inputIP));  // Convert input IP to integer, consider network byte order

    int maxPrefixLength = -1;

    char\* matchingCIDR = NULL;

    for (int i = 0; i < n; ++i) {

        uint32\_t network = ntohl(inet\_addr(cidrList[i].network));  // Convert network IP to integer

        uint32\_t subnetMask = ~((1u << (32 - cidrList[i].subnet)) - 1);

uint32\_t subnetMask = (0xFFFFFFFFu << (32 - cidrList[i].subnet));

        if ((input & subnetMask) == (network & subnetMask)) {

            if (cidrList[i].subnet > maxPrefixLength) {

                maxPrefixLength = cidrList[i].subnet;

                matchingCIDR = cidrList[i].network;

            }

        }

    }

    if (matchingCIDR != NULL) {

        printf("Longest Prefix Match: %s/%d\n", matchingCIDR, maxPrefixLength);

    } else {

        printf("No match found.\n");

    }

}

int main() {

    CIDR cidrList[] = {{"192.168.1.0", 24}, {"10.0.0.0", 16}, {"172.16.0.0", 20}};

    char inputIP[] = "192.168.1.15";

    findLongestPrefixMatch(cidrList, sizeof(cidrList) / sizeof(cidrList[0]), inputIP);

    return 0;

}

## Trie

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h> // Include for bool data type

// Define TrieNode structure

typedef struct TrieNode {

    struct TrieNode\* children[26];

    bool isEndOfWord;

} TrieNode;

// Define Trie structure

typedef struct {

    TrieNode\* root;

} Trie;

// Function to create a new TrieNode

TrieNode\* createNode() {

    TrieNode\* newNode = (TrieNode\*)malloc(sizeof(TrieNode));

    for (int i = 0; i < 26; i++) {

        newNode->children[i] = NULL;

    }

    newNode->isEndOfWord = false;

    return newNode;

}

// Initialize the Trie

Trie\* trieCreate() {

    Trie\* trie = (Trie\*)malloc(sizeof(Trie));

    trie->root = createNode();

    return trie;

}

// Insert a word into the Trie

void trieInsert(Trie\* obj, char\* word) {

    TrieNode\* node = obj->root;

    while (\*word) {

        int index = \*word - 'a';

        if (node->children[index] == NULL) {

            node->children[index] = createNode();

        }

        node = node->children[index];

        word++;

    }

    node->isEndOfWord = true;

}

// Search for a word in the Trie

bool trieSearch(Trie\* obj, char\* word) {

    TrieNode\* node = obj->root;

    while (\*word) {

        int index = \*word - 'a';

        if (node->children[index] == NULL) {

            return false;

        }

        node = node->children[index];

        word++;

    }

    return node->isEndOfWord;

}

// Check if a prefix exists in the Trie

bool trieStartsWith(Trie\* obj, char\* prefix) {

    TrieNode\* node = obj->root;

    while (\*prefix) {

        int index = \*prefix - 'a';

        if (node->children[index] == NULL) {

            return false;

        }

        node = node->children[index];

        prefix++;

    }

    return true;

}

// Free the Trie memory

void trieFree(Trie\* obj) {

    // Use recursive helper function to free Trie nodes

    void freeNode(TrieNode\* node) {

        if (node == NULL) {

            return;

        }

        for (int i = 0; i < 26; i++) {

            freeNode(node->children[i]);

        }

        free(node);

    }

    freeNode(obj->root);

    free(obj);

}

// Function to print all words stored in the Trie

void printTrieHelper(TrieNode\* node, char\* buffer, int depth) {

// Base case: If node is NULL, return

if (node == NULL) {

return;

}

// If current node is end of a word, print the word

if (node->isEndOfWord) {

buffer[depth] = '\0'; // Null-terminate the buffer

printf("%s\n", buffer);

}

// Recursively traverse children nodes

for (int i = 0; i < 26; i++) {

if (node->children[i] != NULL) {

buffer[depth] = 'a' + i; // Add current character to buffer

printTrieHelper(node->children[i], buffer, depth + 1);

}

}

}

void printTrie(Trie\* trie) {

char buffer[100]; // Assuming maximum word length is 100

printTrieHelper(trie->root, buffer, 0);

}

int main() {

    Trie\* trie = trieCreate();

    // Example usage

    trieInsert(trie, "apple");

    printf("Search for 'apple': %s\n", trieSearch(trie, "apple") ? "Found" : "Not found");

    printf("Search for 'app': %s\n", trieSearch(trie, "app") ? "Found" : "Not found");

    printf("Starts with 'app': %s\n", trieStartsWith(trie, "app") ? "True" : "False");

    trieInsert(trie, "app");

    printf("Search for 'app' after insertion: %s\n", trieSearch(trie, "app") ? "Found" : "Not found");

    // Free memory

    trieFree(trie);

    return 0;

}

## Calculator using Tree

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <ctype.h>

// Define structure for expression tree node

typedef struct TreeNode {

    char data;

    struct TreeNode\* left;

    struct TreeNode\* right;

} TreeNode;

// Function to create a new node

TreeNode\* createNode(char data) {

    TreeNode\* newNode = (TreeNode\*)malloc(sizeof(TreeNode));

    newNode->data = data;

    newNode->left = newNode->right = NULL;

    return newNode;

}

// Function to check if a character is an operator

bool isOperator(char c) {

    return (c == '+' || c == '-' || c == '\*' || c == '/');

}

// Function to build expression tree from postfix expression

TreeNode\* buildExpressionTree(char postfix[]) {

    TreeNode\* stack[100]; // Assuming the expression won't exceed 100 characters

    int top = -1;

    for (int i = 0; postfix[i] != '\0'; i++) {

        TreeNode\* newNode = createNode(postfix[i]);

        if (!isOperator(postfix[i])) {

            stack[++top] = newNode;

        } else {

            newNode->right = stack[top--];

            newNode->left = stack[top--];

            stack[++top] = newNode;

        }

    }

    return stack[top];

}

// Function to evaluate expression tree

int evaluate(TreeNode\* root) {

    if (root == NULL) {

        return 0;

    }

    if (!isOperator(root->data)) {

        return root->data - '0'; // Convert char to int

    }

    int leftValue = evaluate(root->left);

    int rightValue = evaluate(root->right);

    switch (root->data) {

        case '+': return leftValue + rightValue;

        case '-': return leftValue - rightValue;

        case '\*': return leftValue \* rightValue;

        case '/': return leftValue / rightValue;

        default: return 0;

    }

}

int main() {

    char postfix[100];

    printf("Enter postfix expression: ");

    scanf("%s", postfix);

    TreeNode\* root = buildExpressionTree(postfix);

    int result = evaluate(root);

    printf("Result: %d\n", result);

    return 0;

}

## Balanced Binary Tree

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h> // Include the bool type

// Definition for a binary tree node.

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Helper function to calculate the height of a subtree

int getHeight(struct TreeNode\* root) {

    if (root == NULL) {

        return 0;

    }

    int leftHeight = getHeight(root->left);

    int rightHeight = getHeight(root->right);

    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);

}

// Main function to check if the binary tree is height-balanced

bool isBalanced(struct TreeNode\* root) {

    if (root == NULL) {

        return true; // An empty tree is height-balanced

    }

    int leftHeight = getHeight(root->left);

    int rightHeight = getHeight(root->right);

    // Check if the height difference of left and right subtrees is more than 1

    if (abs(leftHeight - rightHeight) > 1) {

        return false;

    }

    // Recursively check if both left and right subtrees are height-balanced

    return isBalanced(root->left) && isBalanced(root->right);

}

// Function to create a new tree node

struct TreeNode\* createNode(int val) {

    struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    newNode->val = val;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

// Function to free memory allocated for the tree

void freeTree(struct TreeNode\* root) {

    if (root == NULL) {

        return;

    }

    freeTree(root->left);

    freeTree(root->right);

    free(root);

}

int main() {

    // Create a sample binary tree

    struct TreeNode\* root = createNode(1);

    root->left = createNode(2);

    root->right = createNode(3);

    root->left->left = createNode(4);

    root->left->right = createNode(5);

    root->right->left = createNode(6);

    root->right->right = createNode(7);

    // Check if the binary tree is height-balanced

    if (isBalanced(root)) {

        printf("The binary tree is height-balanced.\n");

    } else {

        printf("The binary tree is not height-balanced.\n");

    }

    freeTree(root);

      // Create a sample binary tree

    root = createNode(1);

    root->left = createNode(2);

    root->right = createNode(2);

    root->left->left = createNode(3);

    root->left->right = createNode(3);

    root->left->right->left = createNode(6);

    root->left->right->right = createNode(7);

    // Check if the binary tree is height-balanced

    if (isBalanced(root)) {

        printf("The binary tree is height-balanced.\n");

    } else {

        printf("The binary tree is not height-balanced.\n");

    }

    // Free memory allocated for the binary tree

    freeTree(root);

    return 0;

}

Time/Space Complexity: O(n^2)/O(n)

## Time Needed To Inform All Employees

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

typedef struct Node {

    int val;

    struct Node\* next;

} Node;

typedef struct {

    Node\* head;

} List;

List\* createList() {

    List\* list = (List\*)malloc(sizeof(List));

    list->head = NULL;

    return list;

}

void printList(List\* list) {

    Node\* current = list->head;

    while (current != NULL) {

        printf("%d ", current->val);

        current = current->next;

    }

    printf("\n");

}

void addToList(List\* list, int val) {

    Node\* newNode = (Node\*)malloc(sizeof(Node));

    newNode->val = val;

    newNode->next = list->head;

    list->head = newNode;

}

int helper(int\* informTime, List\*\* list, int curr) {

    printf("\ncurr : %d",curr);

    if (list[curr] == NULL) return 0;

    int min = 0;

    Node\* current = list[curr]->head;

    while (current != NULL) {

        min = fmax(helper(informTime, list, current->val), min);

        current = current->next;

    }

    return min + informTime[curr];

}

int numOfMinutes(int n, int headID, int\* manager, int managerSize, int\* informTime, int informTimeSize) {

    List\*\* list = (List\*\*)malloc(n \* sizeof(List\*));

    for (int i = 0; i < n; i++) {

        list[i] = createList();

    }

    for (int i = 0; i < n; i++) {

        if (manager[i] != -1) {

            addToList(list[manager[i]], i);

        }

    }

    for (int i = 0; i < n; i++) {

        printf("\ni = %d",i);

        printList(list[i]);

    }

    int result = helper(informTime, list, headID);

    for (int i = 0; i < n; i++) {

        free(list[i]);

    }

    free(list);

    return result;

}

int main() {

    int n = 6;

    int headID = 2;

    int manager[] = {2, 2, -1, 2, 2, 2};

    int informTime[] = {0, 0, 1, 0, 0, 0};

    int result = numOfMinutes(n, headID, manager, sizeof(manager) / sizeof(manager[0]), informTime, sizeof(informTime) / sizeof(informTime[0]));

    printf("Time taken for all employees to receive the information: %d minutes\n", result);

    return 0;

}

Output: 1 minute

Time/Space Complexity : O(n^2)/O(n)

## Distribute Coins In Binary Tree

#include <stdio.h>

#include <stdlib.h>

// Definition for a binary tree node.

struct TreeNode {

    int val;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

// Function to create a new TreeNode

struct TreeNode\* newNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

// Helper function to perform DFS and calculate the number of moves required

int helper(struct TreeNode\* root, int\* moves) {

    if (root == NULL)

        return 0;

    // Recursively traverse left and right subtrees

    int coinsLeft = helper(root->left, moves);

    int coinsRight = helper(root->right, moves);

    // Calculate total coins for the current subtree

    int coins = coinsLeft + coinsRight;

    // Adjust coins based on the current node's value

    if (root->val == 0)

        coins -= 1;

    else if (root->val == 1)

        coins += 0;

    else

        coins += root->val - 1;

    // Update the number of moves required

    \*moves += abs(coins);

    return coins;

}

// Function to distribute coins in the binary tree and return the total number of moves required

int distributeCoins(struct TreeNode\* root) {

    int moves = 0;

    helper(root, &moves);

    return moves;

}

// Main function for testing

int main() {

    // Create a sample binary tree

    struct TreeNode\* root = newNode(3);

    root->left = newNode(0);

    root->right = newNode(0);

    //root->left->right = newNode(3);

    // Distribute coins and calculate the number of moves required

    int moves = distributeCoins(root);

    printf("Total moves required to distribute coins: %d\n", moves);

    // Free dynamically allocated memory for the binary tree

    // (Not included in the original implementation as it doesn't involve dynamic memory allocation)

    free(root->left->right);

    free(root->left);

    free(root->right);

    free(root);

    return 0;

}

Output: 2

Time/Space Complexity: O(n)/O(n) or O(log n)

## Find Original Array From Doubled Array

#include <stdio.h>

#include <stdlib.h>

int\* findOriginalArray(int\* changed, int changedSize, int\* returnSize) {

    \*returnSize = 0;

    int n = changedSize;

    if (n % 2 == 1) {

        return NULL;

    }

    int\* ans = (int\*)malloc((n / 2) \* sizeof(int));

    int idx = 0;

    int\* freq = (int\*)calloc(100005, sizeof(int));

    for (int i = 0; i < n; i++) {

        freq[changed[i]]++;

    }

    for (int num = 100000; num >= 0; num--) {

        while (freq[num]-- > 0) {

            if ((num & 1) == 1) {

                free(ans);

                free(freq);

                return NULL;

            }

            int div = num / 2;

            if (freq[div] == 0) {

                free(ans);

                free(freq);

                return NULL;

            }

            ans[idx++] = div;

            freq[div]--;

        }

    }

    \*returnSize = idx;

    return ans;

}

int main() {

    int changed[] = {1, 3, 4, 2, 6, 8};

    int changedSize = sizeof(changed) / sizeof(changed[0]);

    int returnSize;

    int\* originalArray = findOriginalArray(changed, changedSize, &returnSize);

    if (originalArray == NULL) {

        printf("No original array found!\n");

    } else {

        printf("Original array: ");

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

            printf("%d ", originalArray[i]);

        }

        printf("\n");

        // Free dynamically allocated memory for the original array

        free(originalArray);

    }

    return 0;

}

Output: 4,3,1 or 1,3,4

Time/Space Complexity :O(n)/O(1)

## Sum Of SubArray Minimums

#include <stdio.h>

#include <stdlib.h>

int ind = 0;

int \*ssize;

// Function to count the number of subarrays for given array size

int countSubarrays(int size) {

    return (size \* (size + 1)) / 2;

}

// Function to store all subarrays of an array

int storeSubarrays(int arr[], int size) {

    int result = 0; // Variable to store the cumulative sum of minimums

    for (int i = 0; i < size; ++i) {

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

            int min = arr[i]; // Assume the first element as minimum

            // Find the actual minimum element in the subarray

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

                if (arr[k] < min) {

                    min = arr[k];

                }

            }

            // Add the minimum element to the result variable

            result = (result + min);

        }

    }

    return result;

}

int main() {

    int arr[] = {3, 1, 2, 4};

    int len = sizeof(arr) / sizeof(int);

    // Count the number of subarrays

    int subarraysSize = countSubarrays(len);

    // Calculate the sum of minimums of all subarrays

    int result = storeSubarrays(arr, len);

    // Print the result

    printf("Sum of minimums of all subarrays: %d\n", result);

    return 0;

}

=================================

#include <stdio.h>

#include <stdlib.h>

#define MOD 1000000007

int ind = 0;

int \*ssize;

int sumOfMinimums(int\*\* subarrays, int numSubarrays) {

    int result = 0; // Variable to store the cumulative sum of minimums

    // Calculate the sum of minimums for each subarray and add to the result variable

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

        int min = subarrays[i][0]; // Assume the first element as minimum

        // Find the actual minimum element in the subarray

        for (int j = 1; j < ssize[i]; ++j) {

            if (subarrays[i][j] < min) {

                min = subarrays[i][j];

            }

        }

        printf("\nMin is : %d",min);

        // Add the minimum element to the result variable

        result += min;

    }

    return result;

}

// Function to count the number of subarrays for given array size

int countSubarrays(int size) {

    return (size \* (size + 1)) / 2;

}

// Function to allocate memory for subarrays array

int\*\* allocateSubarrays(int subarraysSize) {

    int\*\* subarrays = (int\*\*)malloc(subarraysSize \* sizeof(int\*)); // Allocate memory for subarrays

    return subarrays;

}

// Function to store all subarrays of an array

int storeSubarrays(int arr[], int size, int\*\* subarrays) {

    int subarraysIndex = 0;

    for (int i = 0; i < size; ++i) {

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

            int subarraySize = j - i + 1;

            ssize[ind++] = subarraySize;

            printf("\nSize %d",subarraySize);

            subarrays[subarraysIndex] = (int\*)malloc(subarraySize \* sizeof(int)); // Allocate memory for each subarray

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

                subarrays[subarraysIndex][k - i] = arr[k];

            }

            subarraysIndex++;

        }

    }

    return subarraysIndex;

}

int main() {

    int arr[] = {3, 1, 2, 4};

    int len = sizeof(arr) / sizeof(int);

    // Count the number of subarrays

    int subarraysSize = countSubarrays(len);

    // Allocate memory for subarrays

    int\*\* subarrays = allocateSubarrays(subarraysSize);

    ssize = (int \*) malloc (subarraysSize\*sizeof(int));

    // Store all subarrays

    int numSubarrays = storeSubarrays(arr, len, subarrays);

    printf("\nResult is : %d\n",sumOfMinimums(subarrays,numSubarrays));

    // Print the stored subarrays

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

        int subarraySize = i + 1;

        printf("Subarray %d: ", i + 1);

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

            printf("%d ", subarrays[i][j]);

        }

        printf("\n");

        // Free memory for each subarray

        free(subarrays[i]);

    }

    // Free allocated memory for subarrays array

    free(subarrays);

    return 0;

}

Time/Space Complexity: O(n^2)/O(n)

## Valid Parantheses String

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

bool checkValidString(char\* s) {

    int minBalance = 0; // Minimum possible balance of open left parentheses

    int maxBalance = 0; // Maximum possible balance of open left parentheses

    int n = strlen(s);

    for (int i = 0; i < n; i++) {

        if (s[i] == '(') {

            minBalance++;

            maxBalance++;

        } else if (s[i] == ')') {

            minBalance = (minBalance > 0) ? minBalance - 1 : 0;

            maxBalance--;

            if (maxBalance < 0) {

                return false; // More right parentheses ')' than left '('

            }

        } else { // When s[i] == '\*'

            minBalance = (minBalance > 0) ? minBalance - 1 : 0; // Treat '\*' as ')'

            maxBalance++; // Treat '\*' as '('

        }

    }

    return minBalance == 0;

}

int main() {

    char\* s1 = "()"; // Expected output: true

    char\* s2 = "(\*)"; // Expected output: true

    char\* s3 = "(\*))"; // Expected output: true

    char\* s4 = "((\*\*))"; // Expected output: true

    char\* s5 = "(()\*"; // Expected output: true

    char\* s6 = "())\*"; // Expected output: false

    printf("%s\n", checkValidString(s1) ? "true" : "false");

    printf("%s\n", checkValidString(s2) ? "true" : "false");

    printf("%s\n", checkValidString(s3) ? "true" : "false");

    printf("%s\n", checkValidString(s4) ? "true" : "false");

    printf("%s\n", checkValidString(s5) ? "true" : "false");

    printf("%s\n", checkValidString(s6) ? "true" : "false");

    return 0;

}

Time/Space Complexity:O(n)/O(1)

## Asteroid Collission

#include <stdio.h>

#include <stdlib.h>

int\* asteroidCollision(int\* asteroids, int asteroidsSize, int\* returnSize) {

    int\* stack = (int\*)malloc(asteroidsSize \* sizeof(int)); // Stack to simulate asteroids moving

    int top = -1; // Initialize top of the stack

    \*returnSize = 0; // Initialize the return size

    // Simulate asteroids moving and collisions

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

        int current = asteroids[i];

        // If stack is empty or current asteroid moves to right (positive)

        if (top == -1 || current > 0) {

            stack[++top] = current;

            (\*returnSize)++;

        } else {

            // Resolve collisions

            while (top >= 0 && stack[top] > 0 && stack[top] < -current) {

                top--; // Top asteroid explodes

                (\*returnSize)--;

            }

            // Check if current asteroid survives collision

            if (top >= 0 && stack[top] == -current) {

                top--; // Both asteroids explode

                (\*returnSize)--;

            } else if (top == -1 || stack[top] < 0) {

                stack[++top] = current; // Current asteroid survives collision

                (\*returnSize)++;

            }

        }

    }

    return stack;

}

int main() {

    // Test cases

    int asteroids1[] = {5, 10, -5};

    int size1;

    int\* result1 = asteroidCollision(asteroids1, 3, &size1);

    printf("Output for test case 1: ");

    for (int i = 0; i < size1; ++i) {

        printf("%d ", result1[i]);

    }

    printf("\n");

    free(result1);

    int asteroids2[] = {8, -8};

    int size2;

    int\* result2 = asteroidCollision(asteroids2, 2, &size2);

    printf("Output for test case 2: ");

    for (int i = 0; i < size2; ++i) {

        printf("%d ", result2[i]);

    }

    printf("\n");

    free(result2);

    int asteroids3[] = {10, 2, -5};

    int size3;

    int\* result3 = asteroidCollision(asteroids3, 3, &size3);

    printf("Output for test case 3: ");

    for (int i = 0; i < size3; ++i) {

        printf("%d ", result3[i]);

    }

    printf("\n");

    free(result3);

    return 0;

}

Time/Space Complexity: O(n)/O(n)

## ReOrganize String (any two adjacent characters are not same)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* reorganizeString(char\* s) {

    int charCounts[26] = {0}; // Array to store the frequency of each character

    int maxCount = 0, letter = 0;

    int len = strlen(s);

    // Count the frequency of each character

    for (int i = 0; i < len; i++) {

        charCounts[s[i] - 'a']++;

    }

    // Find the character with the highest frequency

    for (int i = 0; i < 26; i++) {

        if (charCounts[i] > maxCount) {

            maxCount = charCounts[i];

            letter = i;

        }

    }

    // If the most frequent character exceeds half the length of the string + 1, it's impossible to rearrange

    if (maxCount > (len + 1) / 2) {

        return "";

    }

    // Create an array to store the result

    char\* ans = (char\*)malloc((len + 1) \* sizeof(char));

    ans[len] = '\0';

    int index = 0;

    // Place the most frequent letter

    while (charCounts[letter] != 0) {

        ans[index] = letter + 'a';

        index += 2;

        charCounts[letter]--;

    }

    // Place the rest of the letters in any order

    for (int i = 0; i < 26; i++) {

        while (charCounts[i] > 0) {

            if (index >= len) {

                index = 1;

            }

            ans[index] = i + 'a';

            index += 2;

            charCounts[i]--;

        }

    }

    return ans;

}

int main() {

    char s1[] = "aab";

    char\* result1 = reorganizeString(s1);

    printf("Output for test case 1: %s\n", result1);

    free(result1);

    char s2[] = "aaab";

    char\* result2 = reorganizeString(s2);

    printf("Output for test case 2: %s\n", result2);

    free(result2);

    return 0;

}

Time/Space Complexity: O(n)/O(1)

## Generate All Sub Arrays

#include <stdio.h>

#include <stdlib.h>

int ind = 0;

int \*ssize;

// Function to count the number of subarrays for given array size

int countSubarrays(int size) {

    return (size \* (size + 1)) / 2;

}

// Function to allocate memory for subarrays array

int\*\* allocateSubarrays(int subarraysSize) {

    int\*\* subarrays = (int\*\*)malloc(subarraysSize \* sizeof(int\*)); // Allocate memory for subarrays

    return subarrays;

}

// Function to store all subarrays of an array

int storeSubarrays(int arr[], int size, int\*\* subarrays) {

    int subarraysIndex = 0;

    for (int i = 0; i < size; ++i) {

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

            int subarraySize = j - i + 1;

            ssize[ind++] = subarraySize;

            printf("\nSize %d",subarraySize);

            subarrays[subarraysIndex] = (int\*)malloc(subarraySize \* sizeof(int)); // Allocate memory for each subarray

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

                subarrays[subarraysIndex][k - i] = arr[k];

            }

            subarraysIndex++;

        }

    }

    return subarraysIndex;

}

int main() {

    int arr[] = {3, 1, 2, 4};

    int len = sizeof(arr) / sizeof(int);

    // Count the number of subarrays

    int subarraysSize = countSubarrays(len);

    // Allocate memory for subarrays

    int\*\* subarrays = allocateSubarrays(subarraysSize);

    ssize = (int \*) malloc (subarraysSize\*sizeof(int));

    // Store all subarrays

    int numSubarrays = storeSubarrays(arr, len, subarrays);

    // Print the stored subarrays

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

        int subarraySize = i + 1;

        printf("Subarray %d: ", i + 1);

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

            printf("%d ", subarrays[i][j]);

        }

        printf("\n");

        // Free memory for each subarray

        free(subarrays[i]);

    }

    // Free allocated memory for subarrays array

    free(subarrays);

    return 0;

}

Time/Space Complexity : O(n^3)/O(n^2)

## Bit manipulations (Set/Reset/Get Bits)

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

char result[32] = "\0";

int count(int a) {

    int c = 0;

    for (int j = 0; j < 32; j++) {

        if (a & 1)

            c++;

        a >>= 1;

    }

    return c;

}

char \*bits(int a) {

    memset(result,0,sizeof(result));

    for(int i = 31; i >=0; i--) {

        if (a & (1 << i)) {

            strcat(result,"1");

        } else {

            strcat(result,"0");

        }

    }

    return result;

}

char \*swap\_nibble(int a) {

    int lmask = 0xf0;

    int rmask = 0x0f;

    int left = a & lmask;

    int right = a & rmask;

    memset(result,0,sizeof(result));

    int res = right << 4;

    res |= (left >> 4);

    return bits(res);

}

char \*swap\_nibbles(int a) {

    memset(result, 0, sizeof(result));

    // Swap the nibbles using bitwise operations

    int res = ((a & 0x0F) << 4) | ((a & 0xF0) >> 4);

    return bits(res);

}

int set\_bit(int a, int b) {

    int c = 1 << b;

    a |= c;

    return a;

}

int get\_bit(int a, int b) {

    int d = a;

    return (d >> b)&1;

}

int main(int argc, char \*argv[]) {

    int i = atoi(argv[1]);

    printf("\nBit representation is : %s",bits(i));

    printf("\nNo of 1 bits : %d",count(i));

    printf("\nSwapped Nibble : %s",swap\_nibble(i));

    printf("\nSwapped Nibbles : %s",swap\_nibbles(i));

    printf("\nBit representation is : %s",bits(0x432));

    printf("\nBit representation is : %d",get\_bit(0x432,13));

    int r = set\_bit(0x432,13);

    printf("\nBit representation is : %s",bits(r));

    printf("\nBit got is : %d",get\_bit(r,13));

}

## Decode Ways For String

#include <stdio.h>

#include <string.h>

int numDecodings(char\* s) {

    int n = strlen(s);

    int dp[n + 1];

    dp[0] = 1; // Base case: empty string has one decoding

    dp[1] = (s[0] == '0') ? 0 : 1; // First character

    for (int i = 2; i <= n; i++) {

        // One-digit decoding

        if (s[i - 1] != '0') {

            dp[i] = dp[i - 1];

        } else {

            dp[i] = 0;

        }

        // Two-digit decoding

        int twoDigit = (s[i - 2] - '0') \* 10 + (s[i - 1] - '0');

        if (twoDigit >= 10 && twoDigit <= 26) {

            dp[i] += dp[i - 2];

        }

    }

    return dp[n];

}

int main() {

    char s1[] = "12";

    printf("Output for test case 1: %d\n", numDecodings(s1));

    char s2[] = "226";

    printf("Output for test case 2: %d\n", numDecodings(s2));

    char s3[] = "06";

    printf("Output for test case 3: %d\n", numDecodings(s3));

    return 0;

}

Time/Space Complexity: O(n)/O(n)

## Multiply Two Strings

// Function to calculate the length of a string

int strLength(char\* str) {

    int length = 0;

    while (str[length] != '\0') {

        length++;

    }

    return length;

}

// Function to convert character digit to integer

int charToInt(char c) {

    return c - '0';

}

// Function to convert integer to character digit

char intToChar(int num) {

    return num + '0';

}

// Function to multiply two numbers represented as strings

char\* multiply(char\* num1, char\* num2) {

    int len1 = strLength(num1);

    int len2 = strLength(num2);

    int len3 = len1 + len2;

    // Allocate memory for the result string

    char\* result = (char\*)malloc((len3 + 1) \* sizeof(char));

    result[len3] = '\0';

    // Initialize result with '0'

    for (int i = 0; i < len3; i++) {

        result[i] = '0';

    }

    // Perform multiplication digit by digit

    for (int i = len1 - 1; i >= 0; i--) {

        int carry = 0;

        for (int j = len2 - 1; j >= 0; j--) {

            int digit = charToInt(num1[i]) \* charToInt(num2[j]) + charToInt(result[i + j + 1]) + carry;

            carry = digit / 10;

            result[i + j + 1] = intToChar(digit % 10);

        }

        if (carry > 0) {

            result[i] = intToChar(charToInt(result[i]) + carry);

        }

    }

    // Trim leading zeros if any

    int startIdx = 0;

    while (result[startIdx] == '0' && startIdx < len3 - 1) {

        startIdx++;

    }

    // Shift the result string to remove leading zeros

    if (startIdx > 0) {

        for (int i = 0; i < len3 - startIdx; i++) {

            result[i] = result[i + startIdx];

        }

        result[len3 - startIdx] = '\0';

    }

    return result;

}

int main() {

    char\* num1 = "123";

    char\* num2 = "456";

    char\* result = multiply(num1, num2);

    printf("Multiplication of %s and %s is: %s\n", num1, num2, result);

    // Free memory allocated for result

    free(result);

    return 0;

}

Space/Time Complexity:O(n2)/O(n)

## Buildings With An Ocean View

#include <stdio.h>

#include <stdlib.h>

int\* findBuildings(int\* heights, int heightsSize, int\* returnSize) {

    // Allocate memory for the result array

    int\* result = (int\*)malloc(heightsSize \* sizeof(int));

    \*returnSize = 0;

    // Initialize the maximum height seen so far

    int maxHeight = 0;

    // Traverse the buildings from right to left

    for (int i = heightsSize - 1; i >= 0; i--) {

        // Check if the current building has an ocean view

        if (heights[i] > maxHeight) {

            // Update the maximum height seen so far

            maxHeight = heights[i];

            // Store the index of the building in the result array

            result[(\*returnSize)++] = i;

        }

    }

    // Reverse the result array to have indices in increasing order

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

        int temp = result[i];

        result[i] = result[\*returnSize - i - 1];

        result[\*returnSize - i - 1] = temp;

    }

    return result;

}

int main() {

    int heights[] = {4, 2, 3, 1};

    int heightsSize = sizeof(heights) / sizeof(heights[0]);

    int returnSize;

    int\* result = findBuildings(heights, heightsSize, &returnSize);

    printf("Buildings with ocean view: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    // Free the dynamically allocated memory

    free(result);

    return 0;

}

Time/Space Complexity:O(n)/O(n)

## Bankers Algorithm

// Banker's Algorithm

#include <stdio.h>

int main()

{

    // P0, P1, P2, P3, P4 are the Process names here

    int n, m, i, j, k;

    n = 5; // Number of processes

    m = 3; // Number of resources

    int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix

                        { 2, 0, 0 }, // P1

                        { 3, 0, 2 }, // P2

                        { 2, 1, 1 }, // P3

                        { 0, 0, 2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix

                    { 3, 2, 2 }, // P1

                    { 9, 0, 2 }, // P2

                    { 2, 2, 2 }, // P3

                    { 4, 3, 3 } }; // P4

    int avail[3] = { 3, 3, 2 }; // Available Resources

    int f[n], ans[n], ind = 0;

    for (k = 0; k < n; k++) {

        f[k] = 0;

    }

    int need[n][m];

    for (i = 0; i < n; i++) {

        for (j = 0; j < m; j++)

            need[i][j] = max[i][j] - alloc[i][j];

    }

    int y = 0;

    for (k = 0; k < 5; k++) {

        for (i = 0; i < n; i++) {

            if (f[i] == 0) {

                int flag = 0;

                for (j = 0; j < m; j++) {

                    if (need[i][j] > avail[j]){

                        flag = 1;

                        break;

                    }

                }

                if (flag == 0) {

                    ans[ind++] = i;

                    for (y = 0; y < m; y++)

                        avail[y] += alloc[i][y];

                    f[i] = 1;

                }

            }

        }

    }

    int flag = 1;

    for(int i=0;i<n;i++)

    {

    if(f[i]==0)

    {

        flag=0;

        printf("The following system is not safe");

        break;

    }

    }

    if(flag==1)

    {

    printf("Following is the SAFE Sequence\n");

    for (i = 0; i < n - 1; i++)

        printf(" P%d ->", ans[i]);

    printf(" P%d", ans[n - 1]);

    }

    return (0);

    // This code is contributed by Deep Baldha (CandyZack)

}

## Bucket Sort

#include <stdio.h>

#include<limits.h>

//Function to find maximum element of the array

int max\_element(int array[], int size)

{

    // Initializing max variable to minimum value so that it can be updated

    // when we encounter any element which is greater than it.

    int max = INT\_MIN;

    for (int i = 0; i < size; i++)

    {

        //Updating max when array[i] is greater than max

        if (array[i] > max)

        max = array[i];

    }

    //return the max element

    return max;

}

//Implementing bucket sort

void Bucket\_Sort(int array[], int size)

{

    //Finding max element of array which we will use to create buckets

    int max = max\_element(array, size);

    // Creating buckets

    int bucket[max+1];

    //Initializing buckets to zero

    for (int i = 0; i <= max; i++)

    bucket[i] = 0;

    // Pushing elements in their corresponding buckets

    for (int i = 0; i < size; i++)

    bucket[array[i]]++;

    // Merging buckets effectively

    int j=0;   // j is a variable which points at the index we are updating

    for (int i = 0; i <= max; i++)

    {

        // Running while loop until there is an element in the bucket

        while (bucket[i] > 0)

        {

            // Updating array and increment j

            array[j++] = i;

            // Decreasing count of bucket element

            bucket[i]--;

        }

    }

}

/\* The main() begins \*/

int main()

{

    int array[100], i, num;

    printf("Enter the size of array: ");

    scanf("%d", &num);

    printf("Enter the %d elements to be sorted:\n",num);

    for (i = 0; i < num; i++)

        scanf("%d", &array[i]);

    printf("\nThe array of elements before sorting: \n");

    for (i = 0; i < num; i++)

        printf("%d ", array[i]);

    printf("\nThe array of elements after sorting: \n");

    // Calling bucket sort function

    Bucket\_Sort(array, num);

    for (i = 0; i < num; i++)

        printf("%d ", array[i]);

    printf("\n");

    return 0;

}

## Merge K Sorted arrays – Heapify

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define N 4

// Structure to represent a node in the min heap

struct MinHeapNode {

    int arrayIndex;      // Index of the array in arrays

    int elementIndex;    // Index of the current element in the array

    int value;           // Value of the current element

};

// Function prototypes

void heapify(struct MinHeapNode minHeap[], int size);

void bubbleDown(struct MinHeapNode minHeap[], int index, int size);

int findMinChildIndex(struct MinHeapNode minHeap[], int leftIndex, int rightIndex, int size);

void mergeKSorted(int arrays[][N], int rows, int cols);

// Function to merge K sorted arrays

void mergeKSorted(int arrays[][N], int rows, int cols) {

    int result[rows \* cols];  // Result array to store merged sorted elements

    struct MinHeapNode minHeap[rows]; // Min heap to store one element from each array

    // Initialize the min heap with the first element from each array

    for (int i = 0; i < rows; i++) {

        minHeap[i].arrayIndex = i;

        minHeap[i].elementIndex = 0;

        minHeap[i].value = arrays[i][0];

    }

    // Heapify the min heap

    heapify(minHeap, rows);

    // Merge the sorted arrays

    int resultIndex = 0;

    while (1) {

        // Extract the minimum element from the min heap

        struct MinHeapNode top = minHeap[0];

        result[resultIndex++] = top.value;

        // Increment the element index and update the value

        top.elementIndex++;

        if (top.elementIndex < cols) {

            top.value = arrays[top.arrayIndex][top.elementIndex];

        } else {

            top.value = INT\_MAX; // Set to infinity

        }

        // Update the top element in the min heap

        minHeap[0] = top;

        // Perform bubble down operation

        bubbleDown(minHeap, 0, rows);

        // Check if all arrays have been processed

        if (minHeap[0].value == INT\_MAX) {

            break;

        }

    }

    // Print the merged array

    printf("Merged array is: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

}

// Function to perform bubble down operation in min heap

void bubbleDown(struct MinHeapNode minHeap[], int index, int size) {

    int currentIndex = index;

    while (1) {

        int leftIndex = 2 \* currentIndex + 1;

        int rightIndex = 2 \* currentIndex + 2;

        // Find the index of the smaller child

        int minChildIndex = findMinChildIndex(minHeap, leftIndex, rightIndex, size);

        // If there are no children or current element is smaller than its smallest child, break

        if (minChildIndex == -1 || minHeap[currentIndex].value <= minHeap[minChildIndex].value) {

            break;

        }

        // Swap the current node with its smallest child

        struct MinHeapNode temp = minHeap[currentIndex];

        minHeap[currentIndex] = minHeap[minChildIndex];

        minHeap[minChildIndex] = temp;

        // Update currentIndex for next iteration

        currentIndex = minChildIndex;

    }

}

// Function to find the index of the smallest child of a node in min heap

int findMinChildIndex(struct MinHeapNode minHeap[], int leftIndex, int rightIndex, int size) {

    if (leftIndex >= size) {

        return -1; // No children

    }

    if (rightIndex >= size) {

        return leftIndex; // Only left child exists

    }

    // Return index of the smallest child

    return minHeap[leftIndex].value < minHeap[rightIndex].value ? leftIndex : rightIndex;

}

// Function to heapify the min heap

void heapify(struct MinHeapNode minHeap[], int size) {

    // Start from the last non-leaf node and perform bubble down operation

    for (int i = (size - 1) / 2; i >= 0; i--) {

        bubbleDown(minHeap, i, size);

    }

}

int main() {

    int arrs[][N] = {

        {5, 6, 8, 16},

        {3, 7, 12, 13},

        {1, 10, 11, 15},

        {2, 4, 9, 14}

    };

    int rows = sizeof(arrs) / sizeof(arrs[0]);

    int cols = sizeof(arrs[0]) / sizeof(arrs[0][0]);

    mergeKSorted(arrs, rows, cols);

    return 0;

}

## Merge K sorted arrays – Normal Way

#include <stdio.h>

#define N 4

// Merge arr1[0..N1-1] and arr2[0..N2-1] into arr3[0..N1+N2-1]

void mergeArrays(int arr1[], int arr2[], int N1, int N2, int arr3[]) {

    int i = 0, j = 0, k = 0;

    // Traverse both arrays

    while (i < N1 && j < N2) {

        // Check if current element of first array is smaller than current element of second array

        // If yes, store first array element and increment first array index. Otherwise do the same with second array

        if (arr1[i] < arr2[j])

            arr3[k++] = arr1[i++];

        else

            arr3[k++] = arr2[j++];

    }

    // Store remaining elements of first array

    while (i < N1)

        arr3[k++] = arr1[i++];

    // Store remaining elements of second array

    while (j < N2)

        arr3[k++] = arr2[j++];

}

// This function takes an array of arrays as an argument where all arrays are assumed to be sorted.

// It merges them together and prints the final sorted output.

void mergeKArrays(int arr[][N], int i, int j, int output[]) {

    // If one array is in range

    if (i == j) {

        for (int p = 0; p < N; p++)

            output[p] = arr[i][p];

        return;

    }

    // If only two arrays are left then merge them

    if (j - i == 1) {

        mergeArrays(arr[i], arr[j], N, N, output);

        return;

    }

    // Output arrays

    int out1[N \* (((i + j) / 2) - i + 1)], out2[N \* (j - ((i + j) / 2))];

    // Divide the array into halves

    mergeKArrays(arr, i, (i + j) / 2, out1);

    mergeKArrays(arr, (i + j) / 2 + 1, j, out2);

    // Merge the output array

    mergeArrays(out1, out2, N \* (((i + j) / 2) - i + 1), N \* (j - ((i + j) / 2)), output);

}

// A utility function to print array elements

void printArray(int arr[], int size) {

    for (int i = 0; i < size; i++)

        printf("%d ", arr[i]);

}

// Driver's code

int main() {

    int arr[][N] = {{2, 6, 12, 34}, {1, 9, 20, 1000}, {23, 34, 90, 2000}};

    int K = sizeof(arr) / sizeof(arr[0]);

    int output[N \* K];

    printf("\n%d %d %d\n",sizeof(arr),sizeof(arr[0]),K);

    mergeKArrays(arr, 0, K - 1, output);

    // Print the merged array

    printf("Merged array is: ");

    printArray(output, N \* K);

    return 0;

}

## Linked List Palindrome

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct ListNode {

    int val;

    struct ListNode \*next;

};

struct ListNode \*endoffirsthalf(struct ListNode \*head) {

    struct ListNode \*slow = head;

    struct ListNode \*fast = head;

    while (fast->next != NULL && fast->next->next != NULL) {

        fast = fast->next->next;

        slow = slow->next;

    }

    return slow;

}

struct ListNode \*reverselist(struct ListNode \*head) {

    struct ListNode \*curr = head;

    struct ListNode \*prev = NULL;

    struct ListNode \*next = NULL;

    while (curr != NULL) {

        next = curr->next;

        curr->next = prev;

        prev = curr;

        curr = next;

    }

    return prev;

}

bool isPalindrome(struct ListNode\* head){

    if (head == NULL) return true;

    struct ListNode \*firsthalfend = endoffirsthalf(head);

    struct ListNode \*secondhalfstart = reverselist(firsthalfend->next);

    struct ListNode\* one = head;

    struct ListNode\* two = secondhalfstart;

    while (two != NULL) {

        if (two->val == one->val) {

            two = two->next;

            one = one->next;

        } else {

            return false;

        }

    }

    firsthalfend->next = reverselist(secondhalfstart);

    return true;

}

// Function to create a new node with the given value

struct ListNode\* newNode(int val) {

    struct ListNode\* node = (struct ListNode\*)malloc(sizeof(struct ListNode));

    node->val = val;

    node->next = NULL;

    return node;

}

// Function to insert a new node at the end of the list

void insertAtEnd(struct ListNode\*\* head\_ref, int val) {

    struct ListNode\* new\_node = newNode(val);

    if (\*head\_ref == NULL) {

        \*head\_ref = new\_node;

        return;

    }

    struct ListNode\* last = \*head\_ref;

    while (last->next != NULL) {

        last = last->next;

    }

    last->next = new\_node;

}

// Function to print the elements of the linked list

void printList(struct ListNode\* node) {

    while (node != NULL) {

        printf("%d -> ", node->val);

        node = node->next;

    }

    printf("NULL\n");

}

int main() {

    struct ListNode\* head = NULL;

    // Insert elements into the linked list

    insertAtEnd(&head, 1);

    insertAtEnd(&head, 2);

    insertAtEnd(&head, 3);

    insertAtEnd(&head, 2);

    insertAtEnd(&head, 1);

    printf("Original list: ");

    printList(head);

    // Check if the linked list is a palindrome

    if (isPalindrome(head)) {

        printf("The linked list is a palindrome.\n");

    } else {

        printf("The linked list is not a palindrome.\n");

    }

    return 0;

}

## Interleaving String

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define a helper function with memoization

bool isInterleaveHelper(char \*s1, char \*s2, char \*s3, int p1, int p2, int p3, bool \*\*memo) {

    // Base case: If both s1 and s2 have reached the end

    if (s1[p1] == '\0' && s2[p2] == '\0') {

        return s3[p3] == '\0';

    }

    // If the result is already memoized, return it

    if (memo[p1][p2]) {

        return false;

    }

    // If the current character of s1 matches the current character of s3

    if (s1[p1] == s3[p3]) {

        // Recursively check if remaining characters of s1 and s3 can interleave with s2 and s3

        if (isInterleaveHelper(s1, s2, s3, p1 + 1, p2, p3 + 1, memo)) {

            return true;

        }

    }

    // If the current character of s2 matches the current character of s3

    if (s2[p2] == s3[p3]) {

        // Recursively check if remaining characters of s2 and s3 can interleave with s1 and s3

        if (isInterleaveHelper(s1, s2, s3, p1, p2 + 1, p3 + 1, memo)) {

            return true;

        }

    }

    // Memoize the result to avoid redundant computations

    memo[p1][p2] = true;

    return false;

}

bool isInterleave(char \*s1, char \*s2, char \*s3) {

    int len1 = strlen(s1), len2 = strlen(s2), len3 = strlen(s3);

    // If the lengths of s1 and s2 do not sum up to the length of s3, it's impossible to interleave them

    if (len1 + len2 != len3) {

        return false;

    }

    // Initialize memoization table

    bool \*\*memo = (bool \*\*)malloc((len1 + 1) \* sizeof(bool \*));

    for (int i = 0; i <= len1; i++) {

        memo[i] = (bool \*)calloc(len2 + 1, sizeof(bool));

    }

    // Call the helper function recursively

    bool result = isInterleaveHelper(s1, s2, s3, 0, 0, 0, memo);

    // Free memory allocated for memoization table

    for (int i = 0; i <= len1; i++) {

        free(memo[i]);

    }

    free(memo);

    return result;

}

int main() {

    char s1[] = "aabcc";

    char s2[] = "dbbca";

    char s3[] = "aadbbcbcac";

    if (isInterleave(s1, s2, s3)) {

        printf("%s and %s can interleave to form %s\n", s1, s2, s3);

    } else {

        printf("%s and %s cannot interleave to form %s\n", s1, s2, s3);

    }

    return 0;

}

Time/Space Complexity:O(m\*n)/O(m\*n)

## Reverse Linked List Nodes In K Groups

#include <stdio.h>

#include <stdlib.h>

// Define the ListNode structure

struct ListNode {

    int val;

    struct ListNode \*next;

};

// Define the helper function to reverse a sublist of the linked list

struct ListNode\* reverseList(struct ListNode\* head, struct ListNode\* tail) {

    struct ListNode\* prev = NULL;

    struct ListNode\* curr = head;

    while (prev != tail) {

        struct ListNode\* nextTemp = curr->next;

        curr->next = prev;

        prev = curr;

        curr = nextTemp;

    }

    return prev;

}

// Define the function to reverse linked list in groups of k nodes

struct ListNode\* reverseKGroup(struct ListNode\* head, int k) {

    struct ListNode\* dummy = (struct ListNode\*)malloc(sizeof(struct ListNode));

    dummy->val = 0;

    dummy->next = head;

    struct ListNode\* prevGroupTail = dummy;

    while (head) {

        struct ListNode\* groupHead = head;

        struct ListNode\* groupTail = groupHead;

        int count = 1;

        while (groupTail && count < k) {

            groupTail = groupTail->next;

            count++;

        }

        if (!groupTail) break;

        struct ListNode\* nextGroupHead = groupTail->next;

        struct ListNode\* reversedGroupHead = reverseList(groupHead, groupTail);

        prevGroupTail->next = reversedGroupHead;

        groupHead->next = nextGroupHead;

        prevGroupTail = groupHead;

        head = nextGroupHead;

    }

    struct ListNode\* result = dummy->next;

    free(dummy);

    return result;

}

// Function to create a new node

struct ListNode\* createNode(int val) {

    struct ListNode\* newNode = (struct ListNode\*)malloc(sizeof(struct ListNode));

    newNode->val = val;

    newNode->next = NULL;

    return newNode;

}

// Function to print the linked list

void printList(struct ListNode\* head) {

    while (head) {

        printf("%d -> ", head->val);

        head = head->next;

    }

    printf("NULL\n");

}

int main() {

    // Create a linked list: 1 -> 2 -> 3 -> 4 -> 5 -> NULL

    struct ListNode\* head = createNode(1);

    head->next = createNode(2);

    head->next->next = createNode(3);

    head->next->next->next = createNode(4);

    head->next->next->next->next = createNode(5);

    printf("Original linked list: ");

    printList(head);

    // Reverse the linked list in groups of 2 nodes

    int k = 2;

    struct ListNode\* reversedHead = reverseKGroup(head, k);

    printf("Reversed linked list in groups of %d nodes: ", k);

    printList(reversedHead);

    // Free the memory allocated for the linked lists

    struct ListNode\* temp;

    while (reversedHead) {

        temp = reversedHead;

        reversedHead = reversedHead->next;

        free(temp);

    }

    return 0;

}

**Input:** head = [1,2,3,4,5], k = 2

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

**Input:** head = [1,2,3,4,5], k = 3

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

Time/Space Complexity: O(n)/O(1)

## 4 Sum

#include <stdio.h>

#include <stdlib.h>

// Function to compare integers (used for qsort)

int compareIntegers(const void \*a, const void \*b) {

    return (\*(int \*)a - \*(int \*)b);

}

// Function to return an array of arrays of size \*returnSize

int \*\*fourSum(int \*nums, int numsSize, int target, int \*returnSize, int \*\*returnColumnSizes) {

    // Sort the array in non-decreasing order

    qsort(nums, numsSize, sizeof(int), compareIntegers);

    // Initialize variables

    int \*\*result = NULL;

    \*returnSize = 0;

    // Iterate over each element up to numsSize - 4

    for (int i = 0; i < numsSize - 3; i++) {

        // Skip duplicates

        if (i > 0 && nums[i] == nums[i - 1]) continue;

        for (int j = i + 1; j < numsSize - 2; j++) {

            // Skip duplicates

            if (j > i + 1 && nums[j] == nums[j - 1]) continue;

            int left = j + 1;

            int right = numsSize - 1;

            while (left < right) {

                int sum = nums[i] + nums[j] + nums[left] + nums[right];

                if (sum == target) {

                    // Add the quadruplet to the result

                    (\*returnSize)++;

                    result = (int \*\*)realloc(result, (\*returnSize) \* sizeof(int \*));

                    result[\*returnSize - 1] = (int \*)malloc(4 \* sizeof(int));

                    result[\*returnSize - 1][0] = nums[i];

                    result[\*returnSize - 1][1] = nums[j];

                    result[\*returnSize - 1][2] = nums[left];

                    result[\*returnSize - 1][3] = nums[right];

                    // Skip duplicates

                    while (left < right && nums[left] == nums[left + 1]) left++;

                    while (left < right && nums[right] == nums[right - 1]) right--;

                    left++;

                    right--;

                } else if (sum < target) {

                    left++;

                } else {

                    right--;

                }

            }

        }

    }

    // Set returnColumnSizes

    \*returnColumnSizes = (int \*)malloc((\*returnSize) \* sizeof(int));

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

        (\*returnColumnSizes)[i] = 4;

    }

    return result;

}

// Function to free the memory allocated for the result

void freeResult(int \*\*result, int returnSize) {

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

        free(result[i]);

    }

    free(result);

}

// Function to print the result

void printResult(int \*\*result, int returnSize, int \*returnColumnSizes) {

    printf("[\n");

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

        printf("  [");

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

            printf("%d", result[i][j]);

            if (j < returnColumnSizes[i] - 1) printf(",");

        }

        printf("]");

        if (i < returnSize - 1) printf(",");

        printf("\n");

    }

    printf("]\n");

}

int main() {

    int nums1[] = {1, 0, -1, 0, -2, 2};

    int target1 = 0;

    int numsSize1 = sizeof(nums1) / sizeof(nums1[0]);

    int returnSize1;

    int \*returnColumnSizes1;

    int \*\*result1 = fourSum(nums1, numsSize1, target1, &returnSize1, &returnColumnSizes1);

    printf("Example 1:\n");

    printResult(result1, returnSize1, returnColumnSizes1);

    freeResult(result1, returnSize1);

    free(returnColumnSizes1);

    int nums2[] = {2, 2, 2, 2, 2};

    int target2 = 8;

    int numsSize2 = sizeof(nums2) / sizeof(nums2[0]);

    int returnSize2;

    int \*returnColumnSizes2;

    int \*\*result2 = fourSum(nums2, numsSize2, target2, &returnSize2, &returnColumnSizes2);

    printf("Example 2:\n");

    printResult(result2, returnSize2, returnColumnSizes2);

    freeResult(result2, returnSize2);

    free(returnColumnSizes2);

    return 0;

}

Time/Space Complexity:O(n3)/O(1)

## Largest Rectangle In Histogram

#include <stdio.h>

#include <stdlib.h>

int largestRectangleArea(int\* heights, int heightsSize) {

    // Initialize a stack to store the indices of bars

    int\* stack = (int\*)malloc((heightsSize + 1) \* sizeof(int));

    int top = -1;

    int maxArea = 0;

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

        // Get the current bar's height

        int currentHeight = (i == heightsSize) ? 0 : heights[i];

        // While the stack is not empty and the current bar's height is less than the height of the bar at the top of the stack

        while (top >= 0 && heights[stack[top]] > currentHeight) {

            // Pop the index of the bar from the stack

            int heightIndex = stack[top--];

            // Calculate the width of the rectangle

            int width = (top == -1) ? i : i - stack[top] - 1;

            // Calculate the area of the rectangle

            int area = heights[heightIndex] \* width;

            // Update the maximum area

            if (area > maxArea) {

                maxArea = area;

            }

        }

        // Push the current bar's index onto the stack

        stack[++top] = i;

    }

    // Free the allocated memory for the stack

    free(stack);

    return maxArea;

}

int main() {

    int heights[] = {2, 1, 5, 6, 2, 3};

    int heightsSize = sizeof(heights) / sizeof(heights[0]);

    int maxArea = largestRectangleArea(heights, heightsSize);

    printf("Largest Rectangle Area: %d\n", maxArea);

    return 0;

}

Time/Space Complexity:O(n)/O(n)

## Minimum Path Sum

#include <stdio.h>

#include <stdlib.h>

int min(int a, int b) {

    return (a < b) ? a : b;

}

int minPathSum(int\*\* grid, int gridSize, int\* gridColSize) {

    for (int i = gridSize - 1; i >= 0; i--) {

        for (int j = \*gridColSize - 1; j >= 0; j--) {

            if (i == gridSize - 1 && j != \*gridColSize - 1)

                grid[i][j] += grid[i][j + 1];

            else if (j == \*gridColSize - 1 && i != gridSize - 1)

                grid[i][j] += grid[i + 1][j];

            else if (j != \*gridColSize - 1 && i != gridSize - 1)

                grid[i][j] += min(grid[i + 1][j], grid[i][j + 1]);

        }

    }

    return grid[0][0];

}

int main() {

    int gridSize = 3;

    int gridColSize = 3;

    int\*\* grid = (int\*\*)malloc(gridSize \* sizeof(int\*));

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

        grid[i] = (int\*)malloc(gridColSize \* sizeof(int));

    }

    // Initialize grid values

    grid[0][0] = 1;

    grid[0][1] = 3;

    grid[0][2] = 1;

    grid[1][0] = 1;

    grid[1][1] = 5;

    grid[1][2] = 1;

    grid[2][0] = 4;

    grid[2][1] = 2;

    grid[2][2] = 1;

    // Print original grid

    printf("Original grid:\n");

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

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

            printf("%d ", grid[i][j]);

        }

        printf("\n");

    }

    // Calculate and print the minimum path sum

    int minSum = minPathSum(grid, gridSize, &gridColSize);

    printf("\nMinimum path sum: %d\n", minSum);

    // Free allocated memory for grid

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

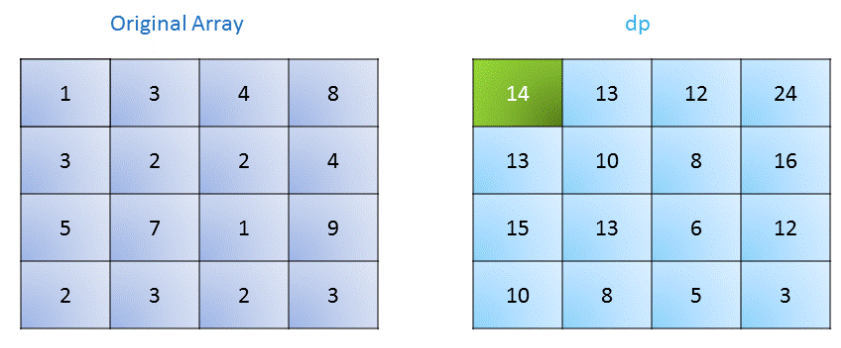
        free(grid[i]);

    }

    free(grid);

    return 0;

}



Space/Time Complexity: O(n\*m)/O(1)

## House Robber II

#include <stdio.h>

int max(int a, int b) {

    return (a > b) ? a : b;

}

int rob\_simple(int\* nums, int start, int end) {

    int t1 = 0;

    int t2 = 0;

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

        int temp = t1;

        int current = nums[i];

        t1 = max(current + t2, t1);

        t2 = temp;

    }

    return t1;

}

int rob(int\* nums, int numsSize) {

    if (numsSize == 0)

        return 0;

    if (numsSize == 1)

        return nums[0];

    int max1 = rob\_simple(nums, 0, numsSize - 2);

    int max2 = rob\_simple(nums, 1, numsSize - 1);

    return max(max1, max2);

}

int main() {

    int nums[] = {2, 7, 9, 3, 1};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int result = rob(nums, numsSize);

    printf("Maximum amount that can be robbed: %d\n", result);

    return 0;

}

Time/Space Complexity:O(n)/O(1)

## Binary Tree Vertical Order Traversal

#include <stdlib.h>

#include <stdio.h>

// Definition for a binary tree node.

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Definition for a queue node.

struct QueueNode {

    struct TreeNode \*node;

    int hd; // Horizontal distance

    struct QueueNode \*next;

};

// Definition for a queue.

struct Queue {

    struct QueueNode \*front, \*rear;

};

// Function to create a new queue node.

struct QueueNode\* newQueueNode(struct TreeNode \*node, int hd) {

    struct QueueNode\* temp = (struct QueueNode\*)malloc(sizeof(struct QueueNode));

    temp->node = node;

    temp->hd = hd;

    temp->next = NULL;

    return temp;

}

// Function to create a new queue.

struct Queue\* createQueue() {

    struct Queue\* q = (struct Queue\*)malloc(sizeof(struct Queue));

    q->front = q->rear = NULL;

    return q;

}

// Function to enqueue a node to the rear of the queue.

void enqueue(struct Queue\* q, struct TreeNode \*node, int hd) {

    struct QueueNode\* temp = newQueueNode(node, hd);

    if (q->rear == NULL) {

        q->front = q->rear = temp;

        return;

    }

    q->rear->next = temp;

    q->rear = temp;

}

// Function to dequeue a node from the front of the queue.

struct QueueNode\* dequeue(struct Queue\* q) {

    if (q->front == NULL) return NULL;

    struct QueueNode\* temp = q->front;

    q->front = q->front->next;

    if (q->front == NULL) q->rear = NULL;

    return temp;

}

// Function to perform vertical order traversal of the binary tree.

int\*\* verticalOrder(struct TreeNode\* root, int\* returnSize, int\*\* returnColumnSizes) {

    if (root == NULL) {

        \*returnSize = 0;

        return NULL;

    }

    // Initialize the variables and data structures

    int minHd = 0, maxHd = 0;

    struct Queue\* q = createQueue();

    enqueue(q, root, 0);

    // Traverse the tree and populate the map

    while (q->front != NULL) {

        struct QueueNode\* current = dequeue(q);

        int hd = current->hd;

        struct TreeNode\* node = current->node;

        if (hd < minHd) minHd = hd;

        if (hd > maxHd) maxHd = hd;

        // Enqueue left child

        if (node->left != NULL) enqueue(q, node->left, hd - 1);

        // Enqueue right child

        if (node->right != NULL) enqueue(q, node->right, hd + 1);

        free(current);

    }

    // Calculate the number of columns

    \*returnSize = maxHd - minHd + 1;

    // Allocate memory for the result arrays

    int\*\* result = (int\*\*)malloc((\*returnSize) \* sizeof(int\*));

    \*returnColumnSizes = (int\*)malloc((\*returnSize) \* sizeof(int));

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

        result[i] = (int\*)malloc(1000 \* sizeof(int)); // Assuming maximum 1000 nodes in each column

        (\*returnColumnSizes)[i] = 0;

    }

    // Reset the queue and enqueue the root node with its horizontal distance

    q = createQueue();

    enqueue(q, root, 0);

    // Traverse the tree again and fill the result arrays

    while (q->front != NULL) {

        struct QueueNode\* current = dequeue(q);

        int hd = current->hd;

        struct TreeNode\* node = current->node;

        int index = hd - minHd;

        // Add the node's value to the corresponding column in the result

        result[index][(\*returnColumnSizes)[index]++] = node->val;

        // Enqueue left child

        if (node->left != NULL) enqueue(q, node->left, hd - 1);

        // Enqueue right child

        if (node->right != NULL) enqueue(q, node->right, hd + 1);

        free(current);

    }

    return result;

}

// Function to create a new binary tree node.

struct TreeNode\* newTreeNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

// Sample main function for testing

int main() {

    // Create a sample binary tree

    struct TreeNode\* root = newTreeNode(3);

    root->left = newTreeNode(9);

    root->right = newTreeNode(20);

    root->right->left = newTreeNode(15);

    root->right->right = newTreeNode(7);

    int returnSize, \*returnColumnSizes;

    int\*\* result = verticalOrder(root, &returnSize, &returnColumnSizes);

    // Print the result

    printf("[");

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

        printf("[");

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

            printf("%d", result[i][j]);

            if (j < returnColumnSizes[i] - 1) printf(",");

        }

        printf("]");

        if (i < returnSize - 1) printf(",");

    }

    printf("]\n");

    // Free memory allocated for result arrays

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

        free(result[i]);

    }

    free(result);

    free(returnColumnSizes);

    // Free memory allocated for the binary tree

    // (Note: Actual implementation may vary depending on the tree structure)

    free(root->left);

    free(root->right);

    free(root);

    return 0;

}

A screenshot of a graph

Description automatically generated

Time/Space Complexity:O(n)/O(n)

## Course Schedule

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

// Definition for a queue node

struct QueueNode {

    int val;

    struct QueueNode\* next;

};

// Definition for a queue

struct Queue {

    struct QueueNode\* front;

    struct QueueNode\* rear;

};

// Function to create a new queue node

struct QueueNode\* newQueueNode(int val) {

    struct QueueNode\* node = (struct QueueNode\*)malloc(sizeof(struct QueueNode));

    node->val = val;

    node->next = NULL;

    return node;

}

// Function to create a new queue

struct Queue\* createQueue() {

    struct Queue\* q = (struct Queue\*)malloc(sizeof(struct Queue));

    q->front = q->rear = NULL;

    return q;

}

// Function to enqueue a node to the rear of the queue

void enqueue(struct Queue\* q, int val) {

    struct QueueNode\* newNode = newQueueNode(val);

    if (q->rear == NULL) {

        q->front = q->rear = newNode;

        return;

    }

    q->rear->next = newNode;

    q->rear = newNode;

}

// Function to dequeue a node from the front of the queue

int dequeue(struct Queue\* q) {

    if (q->front == NULL) return -1;

    struct QueueNode\* temp = q->front;

    int val = temp->val;

    q->front = q->front->next;

    if (q->front == NULL) q->rear = NULL;

    free(temp);

    return val;

}

// Function to check if it's possible to finish all courses

bool canFinish(int numCourses, int\*\* prerequisites, int prerequisitesSize, int\* prerequisitesColSize) {

    int\* indegree = (int\*)calloc(numCourses, sizeof(int));

    int\*\* adj = (int\*\*)malloc(numCourses \* sizeof(int\*));

    // Initialize adjacency list and indegree array

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

        adj[i] = (int\*)malloc(numCourses \* sizeof(int));

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

            adj[i][j] = 0;

        }

    }

    // Populate adjacency list and indegree array

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

        int course = prerequisites[i][0];

        int prereq = prerequisites[i][1];

        if (!adj[prereq][course]) {

            adj[prereq][course] = 1;

            indegree[course]++;

        }

    }

    // Perform topological sort using BFS

    struct Queue\* q = createQueue();

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

        if (indegree[i] == 0) {

            enqueue(q, i);

        }

    }

    int nodesVisited = 0;

    while (q->front) {

        int course = dequeue(q);

        nodesVisited++;

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

            if (adj[course][i]) {

                indegree[i]--;

                if (indegree[i] == 0) {

                    enqueue(q, i);

                }

            }

        }

    }

    // Check if all courses can be finished

    bool canFinish = (nodesVisited == numCourses);

    // Free memory

    free(indegree);

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

        free(adj[i]);

    }

    free(adj);

    free(q);

    return canFinish;

}

int main() {

    int numCourses = 2;

    int prerequisitesSize = 2;

    int prerequisitesColSize[] = {2, 2};

    int\*\* prerequisites = (int\*\*)malloc(prerequisitesSize \* sizeof(int\*));

    prerequisites[0] = (int\*)malloc(2 \* sizeof(int));

    prerequisites[1] = (int\*)malloc(2 \* sizeof(int));

    prerequisites[0][0] = 1;

    prerequisites[0][1] = 0;

    prerequisites[1][0] = 0;

    prerequisites[1][1] = 1;

    bool result = canFinish(numCourses, prerequisites, prerequisitesSize, prerequisitesColSize);

    printf("Result: %s\n", result ? "true" : "false");

    // Free allocated memory

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

        free(prerequisites[i]);

    }

    free(prerequisites);

    return 0;

}

Time/Space Complexity:O(1)/O(n)

## Word Break

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#define ALPHABET\_SIZE 26

struct TrieNode {

    struct TrieNode\* children[ALPHABET\_SIZE];

    bool isEndOfWord;

};

// Function to initialize a new Trie node

struct TrieNode\* createNode() {

    struct TrieNode\* node = (struct TrieNode\*)malloc(sizeof(struct TrieNode));

    for (int i = 0; i < ALPHABET\_SIZE; ++i) {

        node->children[i] = NULL;

    }

    node->isEndOfWord = false;

    return node;

}

// Function to insert a word into the Trie

void insert(struct TrieNode\* root, char\* word) {

    struct TrieNode\* curr = root;

    while (\*word) {

        int index = \*word - 'a';

        if (!curr->children[index]) {

            curr->children[index] = createNode();

        }

        curr = curr->children[index];

        ++word;

    }

    curr->isEndOfWord = true;

}

// Function to perform DFS traversal to check if the input string can be segmented

bool dfs(struct TrieNode\* root, char\* s) {

    struct TrieNode\* curr = root;

    while (\*s) {

        int index = \*s - 'a';

        if (!curr->children[index]) {

            return false; // Cannot find the current character in the trie

        }

        curr = curr->children[index];

        ++s;

        if (curr->isEndOfWord) {

            if (\*s == '\0') {

                return true; // Reach the end of the input string and find a complete word

            }

            if (dfs(root, s)) {

                return true; // Check if the remaining substring can be segmented

            }

        }

    }

    return false;

}

// Function to free the memory used by the Trie

void freeTrie(struct TrieNode\* root) {

    if (!root) {

        return;

    }

    for (int i = 0; i < ALPHABET\_SIZE; ++i) {

        freeTrie(root->children[i]);

    }

    free(root);

}

// Function to check if the input string can be segmented into words from the dictionary

bool wordBreak(char\* s, char\*\* wordDict, int wordDictSize) {

    struct TrieNode\* root = createNode();

    // Build the trie

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

        insert(root, wordDict[i]);

    }

    // Perform DFS traversal

    bool result = dfs(root, s);

    // Free the memory used by the Trie

    freeTrie(root);

    return result;

}

int main() {

    char\* s = "leetcode";

    char\* wordDict[] = {"leet", "code"};

    int wordDictSize = 2;

    bool result = wordBreak(s, wordDict, wordDictSize);

    if (result) {

        printf("The string \"%s\" can be segmented into words from the dictionary.\n", s);

    } else {

        printf("The string \"%s\" cannot be segmented into words from the dictionary.\n", s);

    }

    return 0;

}

Time/Space Complexity: O(n)/O(1)

## Longest Valid Parantheses

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_STACK\_SIZE 10000 // Maximum size of the stack

// Stack structure

typedef struct {

    int\* arr;

    int top;

} Stack;

// Function to initialize the stack

Stack\* createStack() {

    Stack\* stack = (Stack\*)malloc(sizeof(Stack));

    stack->arr = (int\*)malloc(MAX\_STACK\_SIZE \* sizeof(int));

    stack->top = -1;

    return stack;

}

// Function to check if the stack is empty

int isEmpty(Stack\* stack) {

    return stack->top == -1;

}

// Function to push an element onto the stack

void push(Stack\* stack, int val) {

    stack->arr[++stack->top] = val;

}

// Function to pop an element from the stack

int pop(Stack\* stack) {

    return stack->arr[stack->top--];

}

int longestValidParentheses(char\* s) {

    int maxans = 0;

    Stack\* stack = createStack();

    push(stack, -1); // Push -1 as a base index onto the stack

    for (int i = 0; s[i] != '\0'; i++) {

        if (s[i] == '(') {

            push(stack, i);

        } else { // s[i] == ')'

            pop(stack);

            if (isEmpty(stack)) {

                push(stack, i); // Update the base index

            } else {

                maxans = (i - stack->arr[stack->top] > maxans) ? i - stack->arr[stack->top] : maxans;

            }

        }

    }

    free(stack->arr);

    free(stack);

    return maxans;

}

int main() {

    // Test cases

    char\* s1 = "(()";

    printf("Input: %s\n", s1);

    printf("Output: %d\n", longestValidParentheses(s1)); // Output: 2

    char\* s2 = ")()())";

    printf("Input: %s\n", s2);

    printf("Output: %d\n", longestValidParentheses(s2)); // Output: 4

    char\* s3 = "";

    printf("Input: %s\n", s3);

    printf("Output: %d\n", longestValidParentheses(s3)); // Output: 0

    return 0;

}

Time/Space Complexity:O(n)/O(n)

## Edit Distance (Minimum Operation to convert word1 to word2)

#include <stdio.h>

#include <string.h>

// Function to find the minimum of three integers

int min(int a, int b, int c) {

    int min = (a < b) ? a : b;

    return (min < c) ? min : c;

}

// Function to find the minimum edit distance between two strings

int minDistance(char \*word1, char \*word2) {

    int m = strlen(word1);

    int n = strlen(word2);

    if (m == 0) return n;

    if (n == 0) return m;

    int md[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {

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

            if (i == 0) {

                md[i][j] = j;

            } else if (j == 0) {

                md[i][j] = i;

            } else if (word1[i - 1] == word2[j - 1]) {

                md[i][j] = md[i - 1][j - 1];

            } else {

                md[i][j] = 1 + min(md[i - 1][j - 1], md[i - 1][j], md[i][j - 1]);

            }

        }

    }

    return md[m][n];

}

int main() {

    char word1[] = "horse";

    char word2[] = "ros";

    printf("Minimum edit distance between '%s' and '%s' is: %d\n", word1, word2, minDistance(word1, word2));

    return 0;

}

**Input:** word1 = "horse", word2 = "ros"

**Output:** 3

**Explanation:**

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

Time/Space Complexity: O(m\*n)/O(m\*n)

## SubSets II

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Helper function to compare integers for qsort

int compareIntegers(const void \*a, const void \*b) {

    return (\*(int \*)a - \*(int \*)b);

}

// Helper function for backtracking to generate subsets

void backtrack(int \*nums, int numsSize, int start, int \*temp, int tempSize, int \*\*result, int \*returnSize, int \*\*returnColumnSizes) {

    // Allocate memory for the new subset and copy the elements

    int \*subset = (int \*)malloc(tempSize \* sizeof(int));

    memcpy(subset, temp, tempSize \* sizeof(int));

    // Add the subset to the result array

    result[\*returnSize] = subset;

    (\*returnColumnSizes)[\*returnSize] = tempSize;

    (\*returnSize)++;

    // Explore all possible subsets

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

        // Skip duplicates

        if (i > start && nums[i] == nums[i - 1]) continue;

        // Add current element to the temporary subset

        temp[tempSize] = nums[i];

        // Recursively generate subsets starting from the next index

        backtrack(nums, numsSize, i + 1, temp, tempSize + 1, result, returnSize, returnColumnSizes);

    }

}

int\*\* subsetsWithDup(int\* nums, int numsSize, int\* returnSize, int\*\* returnColumnSizes) {

    // Sort the input array to handle duplicates efficiently

    qsort(nums, numsSize, sizeof(int), compareIntegers);

    // Allocate memory for the result array and column sizes array

    int capacity = 10000; // Initial capacity, can be adjusted

    int \*\*result = (int \*\*)malloc(capacity \* sizeof(int \*));

    \*returnColumnSizes = (int \*)malloc(capacity \* sizeof(int));

    \*returnSize = 0;

    // Temporary array to store subsets during backtracking

    int \*temp = (int \*)malloc(numsSize \* sizeof(int));

    // Use backtracking to generate subsets

    backtrack(nums, numsSize, 0, temp, 0, result, returnSize, returnColumnSizes);

    // Free memory for the temporary array

    free(temp);

    return result;

}

int main() {

    // Example input

    int nums[] = {1, 2, 2};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Variables to store the result

    int returnSize;

    int \*returnColumnSizes;

    // Generate subsets

    int \*\*result = subsetsWithDup(nums, numsSize, &returnSize, &returnColumnSizes);

    // Print the result

    printf("[");

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

        printf("[");

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

            printf("%d", result[i][j]);

            if (j < returnColumnSizes[i] - 1) printf(", ");

        }

        printf("]");

        if (i < returnSize - 1) printf(", ");

    }

    printf("]\n");

    // Free dynamically allocated memory

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

        free(result[i]);

    }

    free(result);

    free(returnColumnSizes);

    return 0;

}

Time/Space Complexity:O(2^n)/O(n\*2^n)

## Nth Digit

#include <stdio.h>

int findNthDigit(int n) {

    long base = 9;

    int digits = 1;

    // Step 1: Determine the length of the numbers at each level

    while (n > base \* digits) {

        n -= base \* digits;

        base \*= 10;

        digits++;

    }

    // Step 2: Calculate the actual number where the nth digit belongs

    int number = 1;

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

        number \*= 10;

    }

    number += (n - 1) / digits;

    // Step 3: Find the position of the digit within that number

    int position = (n - 1) % digits;

    // Step 4: Return the digit at the specified position

    for (int i = digits - 1; i > position; i--) {

        number /= 10;

    }

    return number % 10;

}

int main() {

    // Example usage

    int n = 11;

    int nthDigit = findNthDigit(n);

    // Print the result

    printf("The %dth digit is: %d\n", n, nthDigit);

    return 0;

}

Output: The 11th digit of the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... is a 0, which is part of the number 10.

Time/Space Complexity:O(log n)/O(1)

## Minimize The Maximum Difference Of Pairs

#include <stdio.h>

#include <stdlib.h>

// Function to compare integers for qsort

int compare(const void\* a, const void\* b) {

    return (\*(int\*)a - \*(int\*)b);

}

// Function to find the number of valid pairs by greedy approach

int countValidPairs(int\* nums, int size, int threshold) {

    int index = 0, count = 0;

    while (index < size - 1) {

        // If a valid pair is found, skip both numbers.

        if (nums[index + 1] - nums[index] <= threshold) {

            count++;

            index++;

        }

        index++;

    }

    return count;

}

// Function to minimize the maximum difference between pairs

int minimizeMax(int\* nums, int size, int p) {

    // Sort the array

    qsort(nums, size, sizeof(int), compare);

    int left = 0, right = nums[size - 1] - nums[0];

    while (left < right) {

        int mid = left + (right - left) / 2;

        // If there are enough pairs, look for a smaller threshold.

        // Otherwise, look for a larger threshold.

        if (countValidPairs(nums, size, mid) >= p) {

            right = mid;

        } else {

            left = mid + 1;

        }

    }

    return left;

}

int main() {

    // Example usage

    int nums[] = {10, 1, 2, 7, 1, 3};

    int size = sizeof(nums) / sizeof(nums[0]);

    int p = 2;

    // Find the minimum maximum difference between pairs

    int result = minimizeMax(nums, size, p);

    printf("The minimum maximum difference between pairs: %d\n", result);

    return 0;

}

The first pair is formed from the indices 1 and 4, and the second pair is formed from the indices 2 and 5.

The maximum difference is max(|nums[1] - nums[4]|, |nums[2] - nums[5]|) = max(0, 1) = 1. Therefore, we return 1

Output: 1

Time/Space Complexity: O(n log n)/O(1)

## Reverse Words In String II

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function to reverse a portion of the array

void reverse(char\* s, int start, int end) {

    while (start < end) {

        char temp = s[start];

        s[start] = s[end];

        s[end] = temp;

        start++;

        end--;

    }

}

// Function to reverse the order of words in-place

void reverseWords(char\* s, int sSize) {

    // Step 1: Reverse the entire array

    reverse(s, 0, sSize - 1);

    // Step 2: Reverse each individual word

    int start = 0;

    for (int i = 0; i < sSize; i++) {

        if (s[i] == ' ' || i == sSize - 1) {

            // If space is encountered or at the end, reverse the word

            if (i == sSize - 1)

                reverse(s, start, i);

            else

                reverse(s, start, i - 1);

            start = i + 1;

        }

    }

}

int main() {

    // Example usage

    char s[] = "the sky is blue";

    int sSize = strlen(s);

    // Reverse the order of words in the string

    reverseWords(s, sSize);

    // Print the result

    printf("Reversed string: %s\n", s);

    return 0;

}

Output: Blue is Sky The

Time/Space Complexity: O(n)/O(1)

## Number Of Flowers In Full Bloom

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

#include <stdio.h>

#include <stdlib.h>

// Utility function to perform binary search

int compareInt(const void\* a, const void\* b) {

    return (\*(int\*)a - \*(int\*)b);

}

int binarySearch(int\* arr, int size, int target) {

    int left = 0, right = size - 1;

    while (left <= right) {

        int mid = left + (right - left) / 2;

        if (arr[mid] <= target) {

            left = mid + 1;

        } else {

            right = mid - 1;

        }

    }

    return left;

}

int\* fullBloomFlowers(int\*\* flowers, int flowersSize, int\* flowersColSize, int\* people, int peopleSize, int\* returnSize) {

    // Initialize arrays for start and end times

    int\* starts = (int\*)malloc(flowersSize \* sizeof(int));

    int\* ends = (int\*)malloc(flowersSize \* sizeof(int));

    // Populate start and end arrays

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

        starts[i] = flowers[i][0];

        ends[i] = flowers[i][1] + 1;

    }

    // Sort the start and end arrays

    qsort(starts, flowersSize, sizeof(int), compareInt);

    qsort(ends, flowersSize, sizeof(int), compareInt);

    // Initialize result array

    int\* result = (int\*)malloc(peopleSize \* sizeof(int));

    // Process each person's arrival time

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

        int startCount = binarySearch(starts, flowersSize, people[i]);

        int endCount = binarySearch(ends, flowersSize, people[i]);

        result[i] = startCount - endCount;

    }

    // Set the return size

    \*returnSize = peopleSize;

    // Free allocated memory

    free(starts);

    free(ends);

    return result;

}

int main() {

    // Example usage

    int flowersSize = 4;

    int\* flowersColSize = (int\*)malloc(flowersSize \* sizeof(int));

    int\*\* flowers = (int\*\*)malloc(flowersSize \* sizeof(int\*));

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

        flowersColSize[i] = 2;

        flowers[i] = (int\*)malloc(flowersColSize[i] \* sizeof(int));

    }

    flowers[0][0] = 1; flowers[0][1] = 6;

    flowers[1][0] = 3; flowers[1][1] = 7;

    flowers[2][0] = 9; flowers[2][1] = 12;

    flowers[3][0] = 4; flowers[3][1] = 13;

    int people[] = {2, 3, 7, 11};

    int peopleSize = sizeof(people) / sizeof(people[0]);

    int returnSize;

    // Call the function and get the result

    int\* result = fullBloomFlowers(flowers, flowersSize, flowersColSize, people, peopleSize, &returnSize);

    // Print the result

    printf("Result: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    // Free allocated memory

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

        free(flowers[i]);

    }

    free(flowersColSize);

    free(flowers);

    free(result);

    return 0;

}

Output:

A diagram of a number of objects

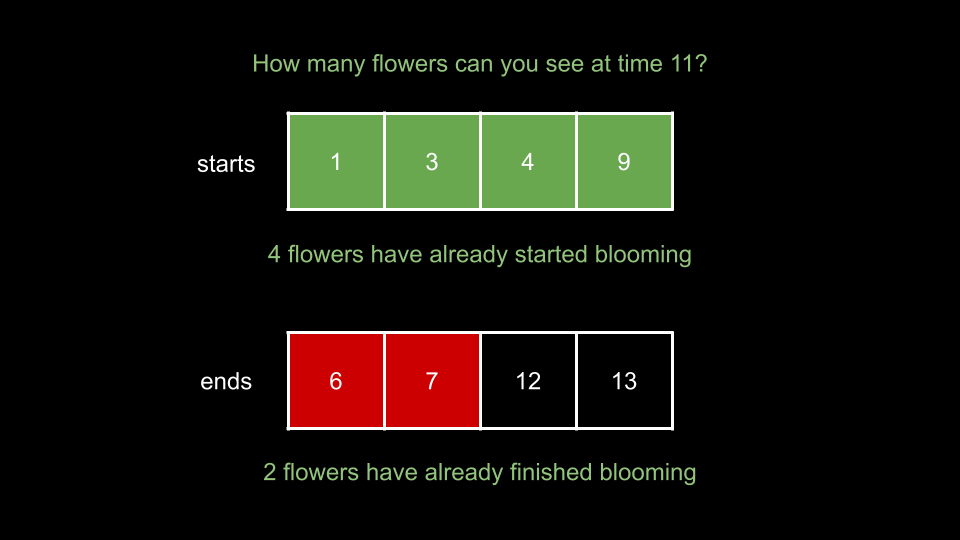
Description automatically generated

**Input:** flowers = [[1,6],[3,7],[9,12],[4,13]], people = [2,3,7,11]

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

**Explanation:** The figure above shows the times when the flowers are in full bloom and when the people arrive.

For each person, we return the number of flowers in full bloom during their arrival.



Time/Space Complexity: O(m+n(log n))/O(m + n)

## Restore IP addresses

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct {

    char\* data;

    int length;

} String;

void rec(const char\* s, int i, char\*\* address, int addressSize, char\*\*\* addresses, int\* returnSize);

int isValid(const char\* s, int start, int length);

void rec(const char\* s, int i, char\*\* address, int addressSize, char\*\*\* addresses, int\* returnSize) {

    if (addressSize == 4) {

        if (s[i] == '\0') {

            // If we have reached the end of the input string, add the current address to the list of valid addresses

            char\* current = (char\*)malloc(16 \* sizeof(char));  // Maximum IP address length is 15

            snprintf(current, 16, "%s.%s.%s.%s", address[0], address[1], address[2], address[3]);

            (\*returnSize)++;

            \*addresses = (char\*\*)realloc(\*addresses, (\*returnSize) \* sizeof(char\*));

            (\*addresses)[\*returnSize - 1] = current;

        }

    } else {

        // Try all possible next segments for the current address

        for (int j = i + 1; j <= i + 3 && s[j - 1] != '\0'; j++) {

            char\* nextInt = (char\*)malloc((j - i + 1) \* sizeof(char));

            strncpy(nextInt, s + i, j - i);

            nextInt[j - i] = '\0';

            // Check if the next segment is valid (between 0 and 255, and not starting with 0 unless it is 0)

            if (isValid(nextInt, 0, j - i)) {

                // Add the next segment to the current address

                address[addressSize] = nextInt;

                // Recursively call the function to generate the next segment

                rec(s, j, address, addressSize + 1, addresses, returnSize);

            }

        }

    }

}

int isValid(const char\* s, int start, int length) {

    int num = 0;

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

        num = num \* 10 + (s[start + i] - '0');

    }

    return (num <= 255) && (length == 1 || (s[start] != '0'));

}

char\*\* restoreIpAddresses(char\* s, int\* returnSize) {

    \*returnSize = 0;

    char\*\* addresses = NULL;

    char\* address[4] = {"", "", "", ""};

    rec(s, 0, address, 0, &addresses, returnSize);

    return addresses;

}

int main() {

    char\* s = "25525511135";

    int returnSize;

    char\*\* addresses = restoreIpAddresses(s, &returnSize);

    // Print the restored IP addresses

    printf("Restored IP Addresses:\n");

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

        printf("%s\n", addresses[i]);

        free(addresses[i]); // Free memory allocated for each address

    }

    free(addresses); // Free memory allocated for the array of addresses

    return 0;

}

Output:["255.255.11.135","255.255.111.35"]

Time/Space Complexity:O( 3 ^ 4) / O(3 ^ 4) – 3 choices for 4 segments

## SubArray Sum Divisible By K

#include <stdio.h>

#include <stdlib.h>

int subarraysDivByK(int\* nums, int numsSize, int k) {

    int n = numsSize;

    int prefixMod = 0, result = 0;

    // There are k mod groups 0...k-1.

    int\* modGroups = (int\*)calloc(k, sizeof(int));

    modGroups[0] = 1;

    for (int i = 0; i < n; ++i) {

        // Take modulo twice to avoid negative remainders.

        prefixMod = (prefixMod + nums[i] % k + k) % k;

        // Add the count of subarrays that have the same remainder as the current

        result += modGroups[prefixMod]++;

    }

    free(modGroups);

    return result;

}

int main() {

    int nums[] = {4, 5, 0, -2, -3, 1};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int k = 5;

    int result = subarraysDivByK(nums, numsSize, k);

    printf("Number of subarrays divisible by %d: %d\n", k, result);

    return 0;

}

The problem is based on the concept of using prefix sums to compute the total number of subarrays that are divisible by k. A prefix sum array for nums is another array prefixSum of the same size as nums, such that the value of prefixSum[i] is the sum of all elements of the nums array from index 0 to index i, i.e., nums[0] + nums[1] + nums[2] + . . . + nums[i].

The sum of the subarray i + 1 to j (inclusive) is computed by prefixSum[j] - prefixSum[i]. Using this, we can count the number of pairs that exist for every pair (i, j) where i < j and (prefixSum[j] - prefix[i]) % k = 0. There are n \* (n - 1) / 2 pairs for an array of length n (pick any two from n). As a result, while this will provide the correct answer for every test case, it will take O(n2)O(n^2)*O*(*n*2) time, indicating that the time limit has been exceeded (TLE).

The character % is the modulo operator.

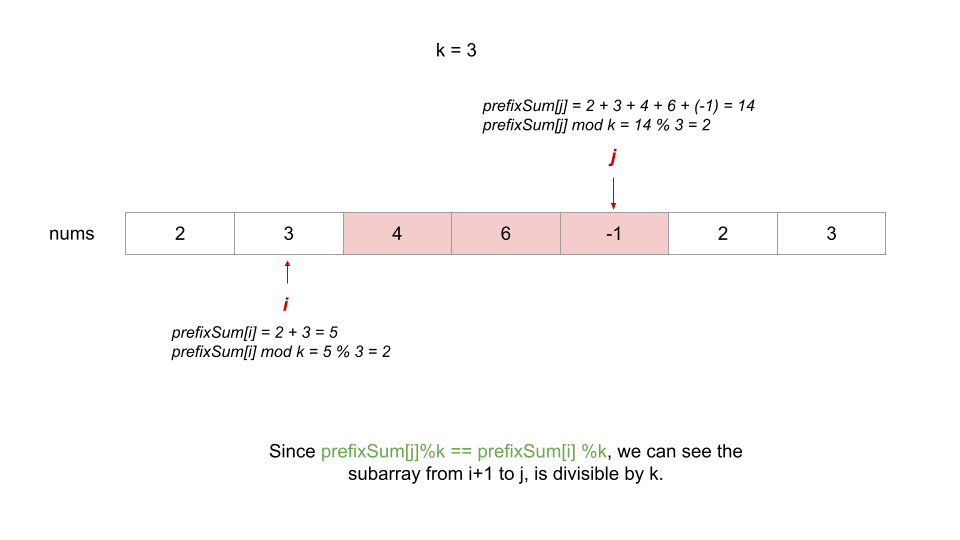
Let's try to use the information with respect to the remainders of every prefix sum and try to optimize the above approach.

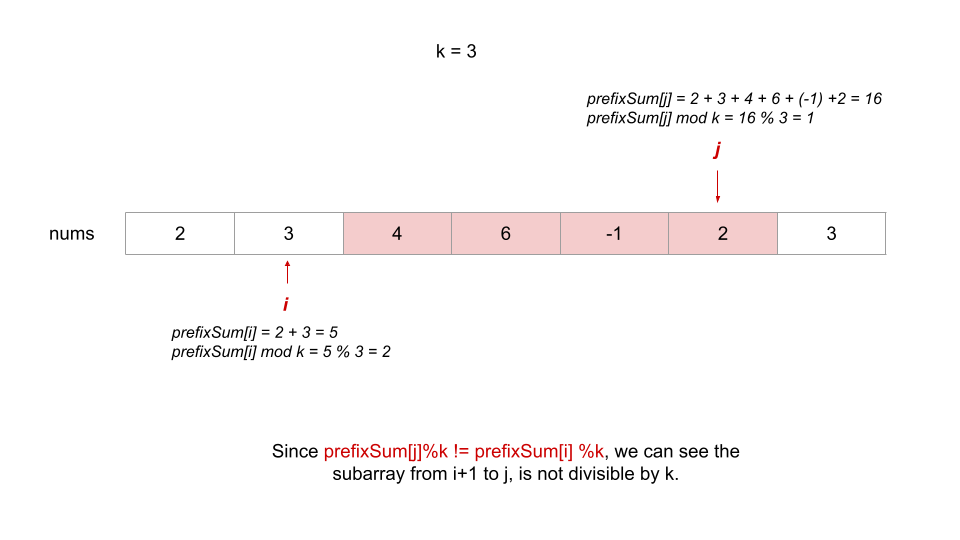
As stated previously, our task is to determine the number of pairs (i, j) where i < j and (prefixSum[j] - prefix[i]) % k = 0. This equality can only be true if prefixSum[i] % k = prefixSum[j] % k. We will demonstrate this property.

We can express any number as number = divisor × quotient + remainder. For example, 13 when divided by 3 can be written as 13 = 3 \* 4 + 1. So we can express:  
a) prefixSum[i] as prefixSum[i] = A \* k + R0 where A is the quotient and R0 is the remainder when divided by k.  
b) Similarly, prefixSum[j] = B \* k + R1 where B is the quotient and R1 is the remainder when divided by k.

We can write, prefixSum[j] - prefixSum[i] = k \* (B - A) + (R1 - R0). The first term (k \* (B - A)) is divisible by k, so for the entire expression to be divisible by k, R1 - R0 must also be divisible by k. This gives us an equation R1 - R0 = C \* k, where C is some integer. Rearranging it yields R1 = C \* k + R0. Because the values of R0 and R1 will be between 0 and k - 1, R1 cannot be greater than k. So the only possible value for C is 0, leading to R0 = R1, which proves the above property. If C > 0, then the RHS would be at least k, but as stated the LHS (R1) is between 0 and k - 1.

Here are two visual examples showing the calculations:





Output: There are 7 subarrays with a sum divisible by k = 5:

[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

Time/Space complexity : O(n)/O(k)

## Find Missing Observations

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

#include <stdio.h>

#include <stdlib.h>

int\* missingRolls(int\* rolls, int rollsSize, int mean, int n, int\* returnSize) {

    int orig\_sum = 0;

    \*returnSize = 0;

    int \*result = (int \*)malloc(sizeof(int) \* n);

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

        orig\_sum += rolls[i];

    }

    printf("\nOrig\_sum = %d", orig\_sum);

    int expected\_sum = (mean \* (n + rollsSize));

    printf("\nExpected\_sum = %d", expected\_sum);

    int missing\_sum = expected\_sum - orig\_sum;

    printf("\nMissing\_sum = %d", missing\_sum);

    int missing\_value = missing\_sum / n;

    int remainder = missing\_sum % n;

    printf("\nMissing Value = %d", missing\_value);

    if ((missing\_value > 6) || (missing\_sum != 0 && missing\_value == 0) || (missing\_sum < 0) || ((missing\_value == 6) && (remainder != 0))) return result;

    for (int j = 0; j < n; j++) {

        result[j] = missing\_value;

        (\*returnSize)++;

    }

    for (int j = 0; j < n; j++) {

        while (remainder != 0) {

            if (missing\_value < 6) {

                if (result[j] < 6) {

                    result[j]++;

                    remainder -= 1;

                } else {

                    break;

                }

            } else {

                continue;

            }

        }

    }

    return result;

}

int main() {

    int rolls[] = {3, 2, 4, 3};

    int rollsSize = sizeof(rolls) / sizeof(rolls[0]);

    int mean = 4;

    int n = 2;

    int returnSize;

    int\* result = missingRolls(rolls, rollsSize, mean, n, &returnSize);

    printf("Missing Rolls: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    free(result); // Don't forget to free allocated memory

    return 0;

}

Output:[6,6]

**Explanation:** The mean of all n + m rolls is (3 + 2 + 4 + 3 + 6 + 6) / 6 = 4.

Time/Space Complexity:O(n)/O(1)

## Largest Palindromic Number

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* largestPalindromic(char\* num) {

    int cnt[10] = {0};

    int n = strlen(num);

    // Count occurrences of each digit

    for (int i = 0; i < n; i++) {

        cnt[num[i] - '0']++;

    }

    char\* lp = (char\*)malloc((n + 1) \* sizeof(char));

    char\* rp = (char\*)malloc((n + 1) \* sizeof(char));

    int lpIndex = 0;

    int rpIndex = 0;

    // Build left and right partitions

    for (int i = 0; i < n; i++) {

        for (int j = 9; j >= 0; j--) {

            if (cnt[j] > 1 && (j > 0 || lpIndex > 0)) {

                lp[lpIndex++] = '0' + j;

                rp[rpIndex++] = '0' + j;

                cnt[j] -= 2;

                break;

            }

        }

    }

    // Find the middle digit (if any)

    for (int i = 9; i >= 0; i--) {

        if (cnt[i]) {

            lp[lpIndex++] = '0' + i;

            break;

        }

    }

    // Reverse the right partition

    for (int i = rpIndex - 1; i >= 0; i--) {

        lp[lpIndex++] = rp[i];

    }

    // Null-terminate the result

    lp[lpIndex] = '\0';

    free(rp);

    return lp;

}

int main() {

    char num[] = "444947137";

    char\* result = largestPalindromic(num);

    printf("Largest Palindromic Integer: %s\n", result);

    free(result); // Don't forget to free allocated memory

    return 0;

}

Output: Use the digits "4449477" from "**444947**13**7**" to form the palindromic integer "7449447".

It can be shown that "7449447" is the largest palindromic integer that can be formed.

Time/Space Complexity:O(n)/O(n)

## Basic Calculator I

#include <stdio.h>

#include <ctype.h>

int evaluate(char operator, int x, int y) {

    if (operator == '+') {

        return x + y;

    } else if (operator == '-') {

        return x - y;

    } else if (operator == '\*') {

        return x \* y;

    }

    return x / y;

}

int calculate(const char\* s) {

    int result = 0;

    int num = 0;

    char op = '+';

    const char\* ptr = s;

    while (\*ptr != '\0') {

        if (isdigit(\*ptr)) {

            num = 0;

            while (isdigit(\*ptr)) {

                num = num \* 10 + (\*ptr - '0');

                ptr++;

            }

            result = evaluate(op, result, num);

        } else if (\*ptr == '+' || \*ptr == '-' || \*ptr == '\*' || \*ptr == '/') {

            op = \*ptr;

            ptr++;

        } else {

            // Ignore whitespace

            ptr++;

        }

    }

    return result;

}

int main() {

    const char\* expression = "3 + 4 \* 2 - 10 / 5"; // Example expression

    int result = calculate(expression);

    printf("Result: %d\n", result); // Output the result

    return 0;

}

**Example 2:**

**Input:** s = " 2-1 + 2 "

**Output:** 3

**Example 3:**

**Input:** s = "(1+(4+5+2)-3)+(6+8)"

**Output:** 23

Time/Space Complexity : O(n)/O(1)

## Basic Calculator II

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

int parseExpr(char\* s, int\* i);

int parseNum(char\* s, int\* i);

int calculate(char\* s);

int main() {

    char expression[] = "3 + 4 \* 2 - 10 / 5"; // Example expression

    int result = calculate(expression);

    printf("Result: %d\n", result); // Output the result

    return 0;

}

int calculate(char\* s) {

    int i = 0;

    return parseExpr(s, &i);

}

int parseExpr(char\* s, int\* i) {

    int nums[1000]; // Assuming a maximum of 1000 numbers

    int numIndex = 0;

    char op = '+';

    for (; \*i < strlen(s); (\*i)++) {

        if (s[\*i] == ' ') continue;

        int n = parseNum(s, i);

        switch(op) {

            case '+' : nums[numIndex++] = n; break;

            case '-' : nums[numIndex++] = -n; break;

            case '\*' : nums[numIndex - 1] \*= n; break;

            case '/' : nums[numIndex - 1] /= n; break;

        }

        op = s[\*i];

    }

    int result = 0;

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

        result += nums[j];

    }

    return result;

}

int parseNum(char\* s, int\* i) {

    int n = 0;

    while(\*i < strlen(s) && isdigit(s[\*i]))

        n = s[(\*i)++] - '0' + 10 \* n;

    return n;

}

**Input:** s = "3+2\*2"

**Output:** 7

Time/Space Complexity: O(n)/O(1)

## Basic Calculator III

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

int evaluate(char operator, int x, int y) {

    if (operator == '+') {

        return x;

    } else if (operator == '-') {

        return -x;

    } else if (operator == '\*') {

        return x \* y;

    }

    return x / y;

}

int solve(const char\* s, int\* i) {

    int stack[1000];

    int top = -1;

    int curr = 0;

    char previousOperator = '+';

    while (s[\*i] != '\0') {

        char c = s[(\*i)++];

        if (c == '(') {

            curr = solve(s, i);

        } else if (isdigit(c)) {

            curr = curr \* 10 + (c - '0');

        } else {

            if (previousOperator == '\*' || previousOperator == '/') {

                stack[++top] = evaluate(previousOperator, stack[top--], curr);

            } else {

                stack[++top] = evaluate(previousOperator, curr, 0);

            }

            if (c == ')') {

                break;

            }

            curr = 0;

            previousOperator = c;

        }

    }

    int ans = 0;

    for (int j = 0; j <= top; j++) {

        ans += stack[j];

    }

    return ans;

}

int calculate(const char\* s) {

    char str[strlen(s) + 2];

    strcpy(str, s);

    strcat(str, "@");

    int i = 0;

    return solve(str, &i);

}

int main() {

    const char\* expression = "2\*(5+5\*2)/3+(6/2+8)";

    int result = calculate(expression);

    printf("Result: %d\n", result);

    return 0;

}

**Input:** s = "2\*(5+5\*2)/3+(6/2+8)"

**Output:** 21

s

Time/Space Complexity : O(n)/O(n)

## Inorder Successor of Binary Search Tree

Given the root of a binary search tree and a node p in it, return *the in-order successor of that node in the BST*. If the given node has no in-order successor in the tree, return null.

The successor of a node p is the node with the smallest key greater than p.val.

**Example 1:**

A diagram of a diagram

Description automatically generated

**Input:** root = [2,1,3], p = 1

**Output:** 2

**Explanation:** 1's in-order successor node is 2. Note that both p and the return value is of TreeNode type.

**Example 2:**

A diagram of a network

Description automatically generated

**Input:** root = [5,3,6,2,4,null,null,1], p = 6

**Output:** null

**Explanation:** There is no in-order successor of the current node, so the answer is null.

**C :**

#include <stdio.h>

#include <stdlib.h>

// Definition for a binary tree node.

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Function to find the inorder successor of a given node in a binary search tree

struct TreeNode\* inorderSuccessor(struct TreeNode\* root, struct TreeNode\* p) {

    struct TreeNode \*successor = NULL;

    while (root != NULL) {

        if (p->val < root->val) {

            successor = root;

            root = root->left;

        } else {

            root = root->right;

        }

    }

    return successor;

}

// Function to create a new node

struct TreeNode\* newNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

// Main function for testing the inorderSuccessor function

int main() {

    struct TreeNode\* root = newNode(2);

    root->left = newNode(1);

    root->right = newNode(3);

    struct TreeNode\* target = root->left; // Node with value 6

    struct TreeNode\* successor = inorderSuccessor(root, target);

    if (successor != NULL) {

        printf("Inorder successor of %d is %d.\n", target->val, successor->val);

    } else {

        printf("Inorder successor of %d does not exist.\n", target->val);

    }

    // Free allocated memory

    free(root->left);

    free(root->right);

    free(root);

    return 0;

}

Output: 1's in-order successor node is 2. Note that both p and the return value is of TreeNode type.

Time/Space Complexity:O(height)/ O(1)

## Regular Expression Matching

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

bool isMatch(char\* s, char\* p) {

    int m = strlen(s);

    int n = strlen(p);

    // Create a 2D array to store intermediate results

    bool dp[m + 1][n + 1];

    memset(dp, false, sizeof(dp));

    // Empty pattern matches empty string

    dp[0][0] = true;

    // Handle patterns with '\*'

    for (int j = 1; j <= n; j++) {

        if (p[j - 1] == '\*') {

            dp[0][j] = dp[0][j - 2];

        }

    }

    // Dynamic programming to fill the rest of the array

    for (int i = 1; i <= m; i++) {

        for (int j = 1; j <= n; j++) {

            if (p[j - 1] == s[i - 1] || p[j - 1] == '.') {

                dp[i][j] = dp[i - 1][j - 1];

            } else if (p[j - 1] == '\*') {

                dp[i][j] = dp[i][j - 2];

                if ((s[i - 1] == p[j - 2]) || (p[j - 2] == '.')) {

                    dp[i][j] = dp[i][j] | dp[i-1][j];

                }

            }

        }

    }

    return dp[m][n];

}

// Main function for testing the isMatch function

int main() {

    char s[] = "aa";

    char p[] = "a\*";

    bool result = isMatch(s, p);

    if (result) {

        printf("String \"%s\" matches pattern \"%s\".\n", s, p);

    } else {

        printf("String \"%s\" does not match pattern \"%s\".\n", s, p);

    }

    return 0;

}

**Example 1:**

**Input:** s = "aa", p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:** s = "aa", p = "a\*"

**Output:** true

**Explanation:** '\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

**Example 3:**

**Input:** s = "ab", p = ".\*"

**Output:** true

**Explanation:** ".\*" means "zero or more (\*) of any character (.)".

Output:'\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

Time/Space Complexity:O(m\*n)/O(m\*n)

## Remove All Adjacent Duplicates In String

#include<stdio.h>

#include<string.h>

char\* removeDuplicates(char\* s) {

    int len = strlen(s);

    int top = -1; // Initialize top of the "stack"

    for (int i = 0; i < len; ++i) {

        if (top >= 0 && s[top] == s[i]) {

            // If the current character matches the top of the "stack", pop the "stack"

            --top;

        } else {

            // Otherwise, push the current character onto the "stack"

            s[++top] = s[i];

        }

    }

    // Null-terminate the string

    s[top + 1] = '\0';

    return s;

}

int main() {

    char s[] = "abbaca";

    printf("Original string: %s\n", s);

    char\* result = removeDuplicates(s);

    printf("String after removing duplicates: %s\n", result);

    return 0;

}

**Input:** s = "abbaca"

**Output:** "ca"

Time/Space Complexity:O(n)/O(1)

## Fibonacci Series

int fib(int N) {

    if (N <= 1) {

        return N;

    }

    int current = 0;

    int prev1 = 1;

    int prev2 = 0;

    for (int i = 2; i <= N; i++) {

        current = prev1 + prev2;

        prev2 = prev1;

        prev1 = current;

    }

    return current;

}

int main() {

    int n = 6;

    int result = fib(n);

    printf("fib(%d) = %d\n", n, result);

    return 0;

}

O(N)/O(1)

#include <stdio.h>

int fib(int n){

    if (n == 0) return 0;

    if (n == 1) return 1;

    return fib(n-1) + fib(n-2);

}

int main() {

    int n = 6;

    int result = fib(n);

    printf("fib(%d) = %d\n", n, result);

    return 0;

}

#include <stdio.h>

int memo[31] = {0}; // Memoization array for Fibonacci numbers

// Recursive function to calculate Fibonacci number

int fib(int n) {

    if (n <= 1)

        return n;

    if (memo[n]) // If the value is already memoized, return it

        return memo[n];

    return memo[n] = fib(n - 1) + fib(n - 2); // Memoize and return the Fibonacci number

    // Alternatively, you can use a 1-liner:

    // return memo[n] = memo[n] ? memo[n] : n <= 1 ? n : fib(n - 1) + fib(n - 2);

}

int main() {

    // Test the fib function

    int n = 10; // Example input

    printf("Fibonacci number at position %d is: %d\n", n, fib(n));

    return 0;

}

#include <stdio.h>

int dp[31] = {0}; // Dynamic programming array for Fibonacci numbers

// Function to calculate Fibonacci number iteratively using dynamic programming

int fib(int n) {

    if (n <= 1)

        return n;

    dp[0] = 0, dp[1] = 1; // Initialize dp array for base cases

    for (int i = 2; i <= n; i++)

        dp[i] = dp[i - 1] + dp[i - 2]; // Compute Fibonacci numbers iteratively

    return dp[n];

}

int main() {

    // Test the fib function

    int n = 10; // Example input

    printf("Fibonacci number at position %d is: %d\n", n, fib(n));

    return 0;

}

Time/Space Complexity: O(n)/O(n)

## Path With Minimum Effort (Hiking heights)

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <math.h>

// Structure to represent a priority queue node

typedef struct {

    int dist;

    int row;

    int col;

} PQNode;

// Structure to represent a priority queue

typedef struct {

    PQNode data[10000];

    int size;

} PriorityQueue;

// Function to initialize the priority queue

void initPriorityQueue(PriorityQueue \*pq) {

    pq->size = 0;

}

void push(PriorityQueue \*pq, PQNode node) {

    int idx = pq->size++;

    pq->data[idx] = node;

    // Maintain min heap property

    while (idx > 0 && pq->data[(idx - 1) / 2].dist > pq->data[idx].dist) {

        // Swap the current node with its parent

        PQNode temp = pq->data[idx];

        pq->data[idx] = pq->data[(idx - 1) / 2];

        pq->data[(idx - 1) / 2] = temp;

        // Move up in the heap

        idx = (idx - 1) / 2;

    }

}

// Function to pop the top node from the priority queue

PQNode pop(PriorityQueue \*pq) {

    PQNode top = pq->data[0];

    for (int i = 1; i < pq->size; ++i) {

        pq->data[i - 1] = pq->data[i];

    }

    pq->size--;

    return top;

}

// Function to find the minimum of two integers

int min(int a, int b) {

    return a < b ? a : b;

}

// Function to get the absolute difference of two integers

int absDiff(int a, int b) {

    return abs(a - b);

}

// Function to find the minimum effort path

int minimumEffortPath(int\*\* heights, int heightsSize, int\* heightsColSize) {

    int m = heightsSize;

    int n = \*heightsColSize;

    // Initialize distance array with a large value

    int dist[m][n];

    for (int i = 0; i < m; ++i) {

        for (int j = 0; j < n; ++j) {

            dist[i][j] = INT\_MAX;

        }

    }

    // Initialize priority queue

    PriorityQueue pq;

    initPriorityQueue(&pq);

    // Push source cell into the priority queue

    PQNode source = {0, 0, 0};

    push(&pq, source);

    dist[0][0] = 0;

    int ans = INT\_MAX;

    // Directions array

    int DIR[] = {0, 1, 0, -1, 0};

    // Dijkstra's algorithm

    while (pq.size > 0) {

        PQNode node = pop(&pq);

        int d = node.dist;

        int r = node.row;

        int c = node.col;

        if (r == m - 1 && c == n - 1) {

            ans = fmin(ans,d);

        }

        // Explore neighbors

        for (int i = 0; i < 4; ++i) {

            int nr = r + DIR[i];

            int nc = c + DIR[i + 1];

            if (nr >= 0 && nr < m && nc >= 0 && nc < n) {

                int newDist = fmax(d, absDiff(heights[nr][nc], heights[r][c]));

                if (dist[nr][nc] > newDist) {

                    dist[nr][nc] = newDist;

                    PQNode newNode = {newDist, nr, nc};

                    push(&pq, newNode);

                }

            }

        }

    }

    return (ans == INT\_MAX)?0:ans;

}

int main() {

    int heights1[3][3] = {{1, 2, 2}, {3, 8, 2}, {5, 3, 5}};

    int heightsSize1 = 3;

    int heightsColSize1 = 3;

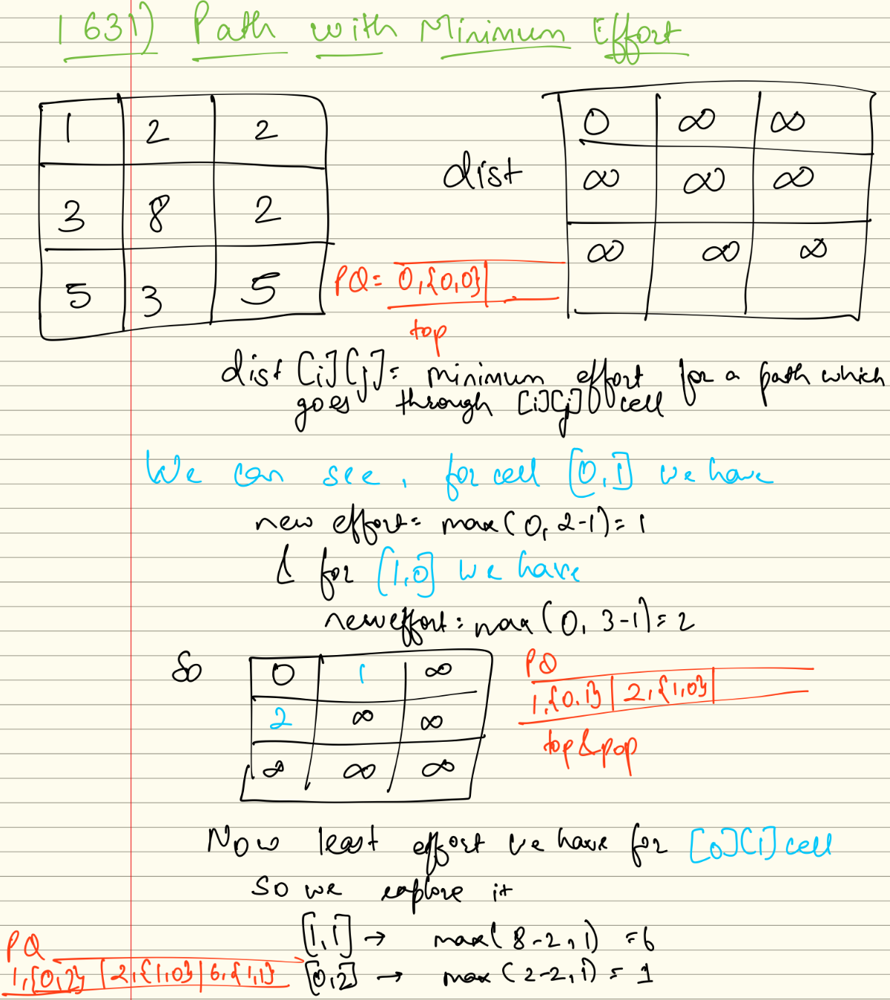
    printf("Minimum effort path for example 1: %d\n", minimumEffortPath((int\*\*)heights1, heightsSize1, &heightsColSize1));

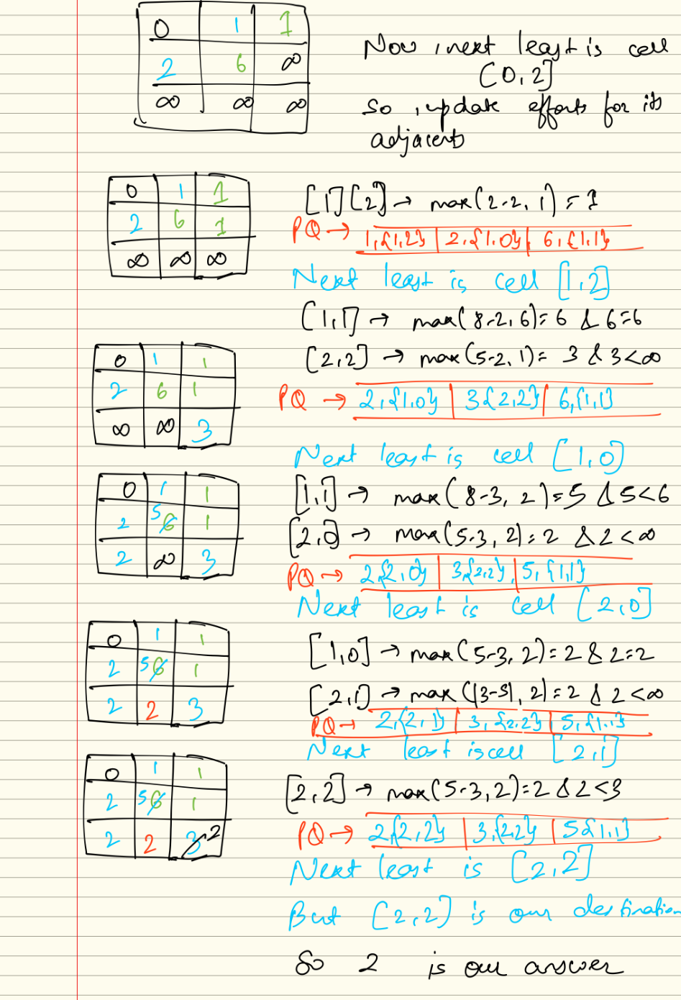
    return 0;

}

Time/Space Complexity:O(log n)/O(n)

I've explained the first test case below for better understanding.





## Longest Cycle In Graph

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

#define MAX\_NODES 1000 // Assuming a maximum of 1000 nodes

int longestCycle(int\* edges, int edgesSize) {

    int n = edgesSize;

    int visit[MAX\_NODES] = {0};

    int indegree[MAX\_NODES] = {0};

    // Count indegree of each node.

    for (int i = 0; i < n; i++) {

        int edge = edges[i];

        if (edge != -1) {

            indegree[edge]++;

        }

    }

    // Kahn's algorithm starts.

    int q[MAX\_NODES], front = 0, rear = -1;

    for (int i = 0; i < n; i++) {

        if (indegree[i] == 0) {

            q[++rear] = i;

        }

    }

    while (front <= rear) {

        int node = q[front++];

        visit[node] = 1;

        int neighbor = edges[node];

        if (neighbor != -1) {

            indegree[neighbor]--;

            if (indegree[neighbor] == 0) {

                q[++rear] = neighbor;

            }

        }

    }

    // Kahn's algorithm ends.

    int answer = -1;

    for (int i = 0; i < n; i++) {

        if (!visit[i]) {

            int neighbor = edges[i];

            int count = 1;

            visit[i] = 1;

            // Iterate in the cycle.

            while (neighbor != i) {

                visit[neighbor] = 1;

                count++;

                neighbor = edges[neighbor];

            }

            answer = (answer < count) ? count : answer;

        }

    }

    return answer;

}

int main() {

    int edges[] = {3,3,4,2,3}; // Example edges array

    int edgesSize = sizeof(edges) / sizeof(edges[0]);

    int result = longestCycle(edges, edgesSize);

    printf("Length of the longest cycle: %d\n", result);

    return 0;

}

A diagram of a diagram

Description automatically generated

**Input:** edges = [3,3,4,2,3]

**Output:** 3

**Explanation:** The longest cycle in the graph is the cycle: 2 -> 4 -> 3 -> 2.

The length of this cycle is 3, so 3 is returned.

Time/Space Complexity: O(n)/O(n)

## Maximum Value Of K Coins From Piles

#include <stdio.h>

#include <stdlib.h>

#define max(a, b) ((a) > (b) ? (a) : (b))

#define min(a, b) ((a) < (b) ? (a) : (b))

#define MAX\_N 100

#define MAX\_K 10000

int\*\* dp;

int knapsack(int\*\* piles, int\* pilesColSize, int n, int k) {

    if (n < 0 || k == 0) return 0;

    if (dp[n][k] != -1) return dp[n][k];

    int pileSize = pilesColSize[n];

    int maxSize = min(pileSize, k);

    int exclude = knapsack(piles, pilesColSize, n - 1, k);

    int include = 0;

    for (int j = 0, sum = 0; j < maxSize; j++) {

        sum += piles[n][j];

        include = max(sum + knapsack(piles, pilesColSize, n - 1, k - j - 1), include);

    }

    int res = max(include, exclude);

    dp[n][k] = res;

    return res;

}

int maxValueOfCoins(int\*\* piles, int pilesSize, int\* pilesColSize, int k) {

    dp = (int\*\*)malloc((pilesSize + 1) \* sizeof(int\*));

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

        dp[i] = (int\*)malloc((k + 1) \* sizeof(int));

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

            dp[i][j] = -1;

        }

    }

    int result = knapsack(piles, pilesColSize, pilesSize - 1, k);

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

        free(dp[i]);

    }

    free(dp);

    return result;

}

int main() {

    // Example 1

    int piles1[][3] = {{1,100,3}, {7,8,9}};

    int pilesColSize1[] = {3, 3};

    int k1 = 2;

    printf("Example 1 Output: %d\n", maxValueOfCoins((int\*\*)piles1, 2, pilesColSize1, k1)); // Output: 101

    // Example 2

    int piles2[][7] = {{100}, {100}, {100}, {100}, {100}, {100}, {1,1,1,1,1,1,700}};

    int pilesColSize2[] = {1, 1, 1, 1, 1, 1, 7};

    int k2 = 7;

    printf("Example 2 Output: %d\n", maxValueOfCoins((int\*\*)piles2, 7, pilesColSize2, k2)); // Output: 706

    return 0;

}

**Example 1:**

A screenshot of a graph

Description automatically generated

**Input:** piles = [[1,100,3],[7,8,9]], k = 2

**Output:** 101

**Explanation:**

The above diagram shows the different ways we can choose k coins.

The maximum total we can obtain is 101.

Time/Space Complexity: O(n\*k\*maxsize)/O(n\*k)

## Flip Equivalent Binary Trees

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

// Definition for a binary tree node.

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Function prototypes

void dfs(struct TreeNode\* node, int\*\* vals, int\* valsSize);

bool flipEquiv(struct TreeNode\* root1, struct TreeNode\* root2);

bool flipEquiv(struct TreeNode\* root1, struct TreeNode\* root2) {

    // If both roots are NULL, they are equivalent

    if (root1 == NULL && root2 == NULL) {

        return true;

    }

    // If one root is NULL and the other is not, they are not equivalent

    if (root1 == NULL || root2 == NULL) {

        return false;

    }

    // If values are not equal, they are not equivalent

    if (root1->val != root2->val) {

        return false;

    }

    // Check if either both nodes have no children or both nodes have children

    bool childrenEquivalent = (flipEquiv(root1->left, root2->left) && flipEquiv(root1->right, root2->right)) ||

                               (flipEquiv(root1->left, root2->right) && flipEquiv(root1->right, root2->left));

    return childrenEquivalent;

}

// Function to create a new tree node

struct TreeNode\* newNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

int main() {

    // Example trees

    struct TreeNode\* root1 = newNode(1);

    root1->left = newNode(2);

    root1->right = newNode(3);

    root1->left->left = newNode(4);

    root1->left->right = newNode(5);

    root1->right->left = newNode(6);

    root1->right->right = newNode(-1);

    root1->left->left->left = newNode(-1);

    root1->left->left->right = newNode(-1);

    root1->left->right->left = newNode(7);

    root1->left->right->right = newNode(8);

    struct TreeNode\* root2 = newNode(1);

    root2->left = newNode(3);

    root2->right = newNode(2);

    root2->left->left = newNode(-1);

    root2->left->right = newNode(4);

    root2->left->right->left = newNode(-1);

    root2->left->right->right = newNode(-1);

    root2->right->left = newNode(6);

    root2->right->left->left = newNode(-1);

    root2->right->left->right = newNode(-1);

    root2->right->right = newNode(5);

    root2->right->right->left = newNode(8);

    root2->right->right->right = newNode(7);

    // Check if the trees are flip equivalent

    printf("Trees are%s flip equivalent.\n", flipEquiv(root1, root2) ? "" : " not");

    // Free memory

    free(root1->left->right->left);

    free(root1->left->right->right);

    free(root1->left->right);

    free(root1->left);

    free(root1->right->left);

    free(root1->right->right);

    free(root1->right);

    free(root1);

    free(root2->left->right);

    free(root2->left);

    free(root2->right->left);

    free(root2->right->right->left);

    free(root2->right->right->right);

    free(root2->right->right);

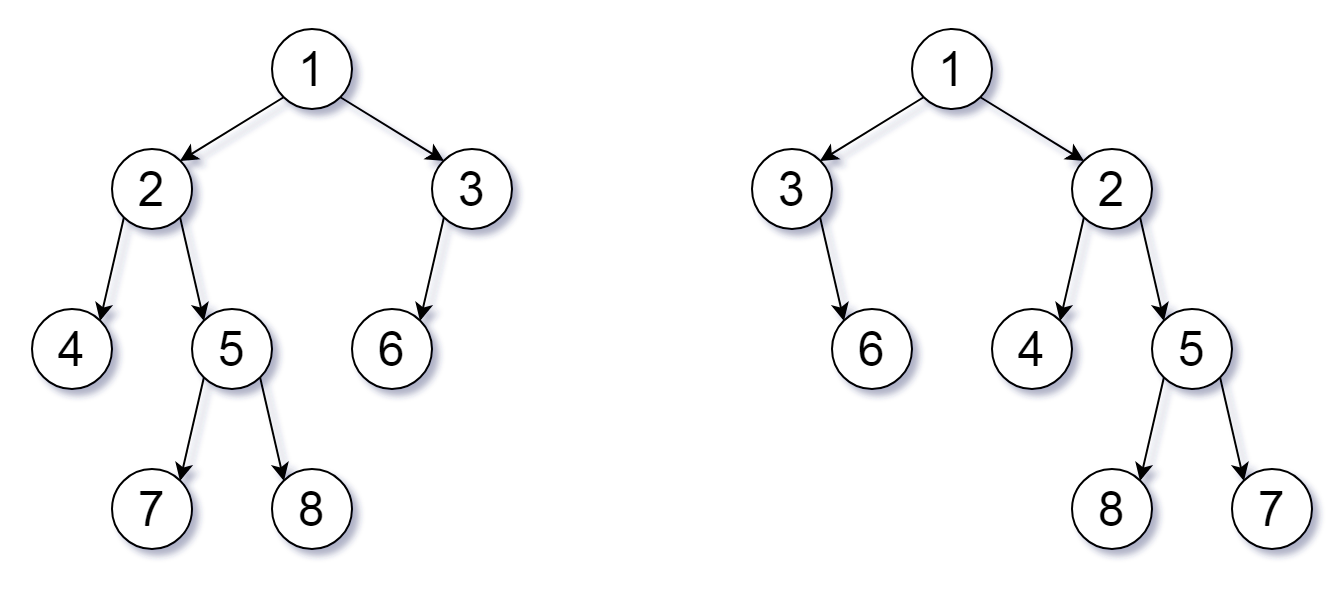
    free(root2->right);

    free(root2);

    return 0;

}

**Example 1:**



**Input:** root1 = [1,2,3,4,5,6,null,null,null,7,8], root2 = [1,3,2,null,6,4,5,null,null,null,null,8,7]

**Output:** true

**Explanation:** We flipped at nodes with values 1, 3, and 5.

Time/Space Complexity : O(n)/O(h)

// Definition for a binary tree node.

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Function prototypes

void dfs(struct TreeNode\* node, int\*\* vals, int\* valsSize);

bool flipEquiv(struct TreeNode\* root1, struct TreeNode\* root2);

// Function to perform depth-first traversal and store node values

void dfs(struct TreeNode\* node, int\*\* vals, int\* valsSize) {

    if (node != NULL) {

        // Add node value to the list of values

        (\*vals)[(\*valsSize)++] = node->val;

        // Store values of left and right children, or -1 if child is NULL

        int L = node->left != NULL ? node->left->val : -1;

        int R = node->right != NULL ? node->right->val : -1;

        // Choose the order of traversal based on values of left and right children

        if (L < R) {

            dfs(node->left, vals, valsSize);

            dfs(node->right, vals, valsSize);

        } else {

            dfs(node->right, vals, valsSize);

            dfs(node->left, vals, valsSize);

        }

        // Add a marker (null) to indicate end of subtree

        (\*vals)[(\*valsSize)++] = -1;

    }

}

// Function to check if two trees are flip equivalent

bool flipEquiv(struct TreeNode\* root1, struct TreeNode\* root2) {

    // If both roots are NULL, they are equivalent

    if (root1 == NULL && root2 == NULL) {

        return true;

    }

    // If one root is NULL and the other is not, they are not equivalent

    if (root1 == NULL || root2 == NULL) {

        return false;

    }

    // Allocate memory to store values of tree nodes during traversal

    int\* vals1 = (int\*)malloc(sizeof(int) \* 1000);

    int\* vals2 = (int\*)malloc(sizeof(int) \* 1000);

    int valsSize1 = 0, valsSize2 = 0;

    // Perform depth-first traversal for both trees

    dfs(root1, &vals1, &valsSize1);

    dfs(root2, &vals2, &valsSize2);

    // Check if the values obtained from both trees are equal

    bool areEquivalent = true;

    if (valsSize1 == valsSize2) {

        for (int i = 0; i < valsSize1; i++) {

            if (vals1[i] != vals2[i]) {

                areEquivalent = false;

                break;

            }

        }

    } else {

        areEquivalent = false;

    }

    // Free dynamically allocated memory

    free(vals1);

    free(vals2);

    return areEquivalent;

}

## Maximum Length Of a Concatenated String With Unique Characters

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function prototype

int solve(char\*\* arr, int arrSize, char\* str, int index);

int maxLength(char\*\* arr, int arrSize);

// Function to calculate the maximum length of a string without repeating characters

int maxLength(char\*\* arr, int arrSize) {

    return solve(arr, arrSize, "", 0);

}

// Function to recursively calculate the maximum length of a string without repeating characters

int solve(char\*\* arr, int arrSize, char\* str, int index) {

    // Create a set to store unique characters

    int charSet[26] = {0};

    // Calculate the length of the current string

    int len = strlen(str);

    // Check if the current string contains duplicate characters

    for (int i = 0; i < len; i++) {

        if (charSet[str[i] - 'a'] > 0) {

            return 0;

        }

        charSet[str[i] - 'a']++;

    }

    // Initialize answer with the length of the current string

    int ans = len;

    // Iterate over the array of strings starting from the given index

    for (int i = index; i < arrSize; i++) {

        // Concatenate the current string with the next string in the array

        char\* newStr = (char\*)malloc((len + strlen(arr[i]) + 1) \* sizeof(char));

        strcpy(newStr, str);

        strcat(newStr, arr[i]);

        // Recursively call solve function with the updated string and index

        int newLen = solve(arr, arrSize, newStr, i + 1);

        // Update the answer with the maximum length obtained so far

        ans = (ans > newLen) ? ans : newLen;

        // Free the dynamically allocated memory for newStr

        free(newStr);

    }

    // Return the maximum length of the string without repeating characters

    return ans;

}

int main() {

    // Example array of strings

    char\* arr[] = {"un", "iq", "ue"};

    // Calculate the maximum length of the string without repeating characters

    printf("Maximum length of string without repeating characters: %d\n", maxLength(arr, 3));

    return 0;

}

**Input:** arr = ["un","iq","ue"]

**Output:** 4

**Explanation:** All the valid concatenations are:

- ""

- "un"

- "iq"

- "ue"

- "uniq" ("un" + "iq")

- "ique" ("iq" + "ue")

Maximum length is 4.

Time/Space Complexity : O(n\*m\*k)/O(n)

## Buddy Strings

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

// Function prototype

bool buddyStrings(char\* s, char\* goal);

// Function to check if two strings are buddy strings

bool buddyStrings(char\* s, char\* goal) {

    // Calculate the lengths of the strings

    int len\_s = strlen(s);

    int len\_goal = strlen(goal);

    // If lengths are not equal, strings cannot be buddy strings

    if (len\_s != len\_goal) {

        return false;

    }

    // If strings are equal, check if there are duplicate characters

    if (strcmp(s, goal) == 0) {

        // Create an array to store the frequency of characters

        int frequency[26] = {0};

        // Count the frequency of each character in the string

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

            frequency[s[i] - 'a']++;

            if (frequency[s[i] - 'a'] == 2) {

                return true;

            }

        }

        // If there are no duplicate characters, strings cannot be buddy strings

        return false;

    }

    // Initialize indices to store positions of differing characters

    int firstIndex = -1;

    int secondIndex = -1;

    // Iterate through the strings to find differing characters

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

        if (s[i] != goal[i]) {

            if (firstIndex == -1) {

                firstIndex = i;

            } else if (secondIndex == -1) {

                secondIndex = i;

            } else {

                // If there are more than two differing characters, strings cannot be buddy strings

                return false;

            }

        }

    }

    // If there is only one differing character, strings cannot be buddy strings

    if (secondIndex == -1) {

        return false;

    }

    // Check if swapping characters at differing positions makes strings equal

    return s[firstIndex] == goal[secondIndex] && s[secondIndex] == goal[firstIndex];

}

int main() {

    // Example strings

    char s[] = "ab";

    char goal[] = "ba";

    // Check if the strings are buddy strings

    printf("Are the strings buddy strings? %s\n", buddyStrings(s, goal) ? "Yes" : "No");

    return 0;

}

**Input:** s = "ab", goal = "ba"

**Output:** true

**Explanation:** You can swap s[0] = 'a' and s[1] = 'b' to get "ba", which is equal to goal.

Time/Space Complexity: O(n)/O(1)

## Robot Room Cleaner

/\*\*

 \* // This is the robot's control interface.

 \* // You should not implement it, or speculate about its implementation

 \* interface Robot {

 \*     // Returns true if the cell in front is open and robot moves into the cell.

 \*     // Returns false if the cell in front is blocked and robot stays in the current cell.

 \*     public boolean move();

 \*

 \*     // Robot will stay in the same cell after calling turnLeft/turnRight.

 \*     // Each turn will be 90 degrees.

 \*     public void turnLeft();

 \*     public void turnRight();

 \*

 \*     // Clean the current cell.

 \*     public void clean();

 \* }

 \*/

class Solution {

    private final int[][] directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

    private Set<String> visited = new HashSet<>();

    public void cleanRoom(Robot robot) {

        dfs(robot, 0, 0, 0);

    }

    private void dfs(Robot robot, int x, int y, int dir) {

        visited.add(x + "-" + y);

        robot.clean();

        for (int i = 0; i < 4; i++) {

            int newDir = (dir + i) % 4;

            int newX = x + directions[newDir][0];

            int newY = y + directions[newDir][1];

            if (!visited.contains(newX + "-" + newY) && robot.move()) {

                dfs(robot, newX, newY, newDir);

                goBack(robot);

            }

            robot.turnRight();

        }

    }

    private void goBack(Robot robot) {

        robot.turnRight();

        robot.turnRight();

        robot.move();

        robot.turnRight();

        robot.turnRight();

    }

}

**Input:** room = [[1,1,1,1,1,0,1,1],[1,1,1,1,1,0,1,1],[1,0,1,1,1,1,1,1],[0,0,0,1,0,0,0,0],[1,1,1,1,1,1,1,1]], row = 1, col = 3

**Output:** Robot cleaned all rooms.

**Explanation:** All grids in the room are marked by either 0 or 1.

0 means the cell is blocked, while 1 means the cell is accessible.

The robot initially starts at the position of row=1, col=3.

From the top left corner, its position is one row below and three columns right.

Time/Space Complexity: O(n)/O(n)

## Count Unique Characters Of All Substrings Of A Given String

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#define ALPHABET\_SIZE 26

#define MOD ((int)pow(10, 9) + 7)

int uniqueLetterString(char\* s) {

    int mod = MOD;

    int index[ALPHABET\_SIZE][2];

    memset(index, -1, sizeof(index));

    int ans = 0;

    int n = strlen(s);

    for (int i = 0; i < n; i++) {

        int c = s[i] - 'A';

        ans = (ans + ((long long)(i - index[c][1]) \* (index[c][1] - index[c][0]) % mod) + mod) % mod;

        index[c][0] = index[c][1];

        index[c][1] = i;

    }

    for (int c = 0; c < ALPHABET\_SIZE; ++c)

        ans = (ans + ((long long)(n - index[c][1]) \* (index[c][1] - index[c][0]) % mod) + mod) % mod;

    return ans;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int uniqueLetterString(char\* s) {

    int sum = 0, curr = 0, n = strlen(s);

    int last[26];

    int prev[26];

    memset(last, -1, sizeof(last));

    memset(prev, -1, sizeof(prev));

    for (int i = 0; i < n; i++) {

        char c = s[i];

        curr += (i - last[c - 'A'] - 1) - (last[c - 'A'] - prev[c - 'A']) + 1;

        prev[c - 'A'] = last[c - 'A'];

        last[c - 'A'] = i;

        sum += curr;

    }

    return sum;

}

int main() {

    char s[] = "ABC";

    printf("Output: %d\n", uniqueLetterString(s));

    return 0;

}

**Input:** s = "ABC"

**Output:** 10

**Explanation:** All possible substrings are: "A","B","C","AB","BC" and "ABC".

Every substring is composed with only unique letters.

Sum of lengths of all substring is 1 + 1 + 1 + 2 + 2 + 3 = 10

Time/Space Complexity: O(n)/O(1)

## Minimum Fuel Cost To Report To The Capital

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the adjacency list

typedef struct Node {

    int data;

    struct Node\* next;

} Node;

// Structure to represent the adjacency list

typedef struct {

    Node\* head;

} List;

// Function to create a new node

Node\* createNode(int data) {

    Node\* newNode = (Node\*)malloc(sizeof(Node));

    if (newNode) {

        newNode->data = data;

        newNode->next = NULL;

    }

    return newNode;

}

// Function to initialize the adjacency list

List initializeList() {

    List list;

    list.head = NULL;

    return list;

}

// Function to add a new edge to the adjacency list

void addEdge(List\*\* adjList, int src, int dest) {

    Node\* newNode = createNode(dest);

    newNode->next = (\*adjList)[src].head;

    (\*adjList)[src].head = newNode;

    newNode = createNode(src);

    newNode->next = (\*adjList)[dest].head;

    (\*adjList)[dest].head = newNode;

}

// Function to perform depth-first search (DFS)

void dfs(List\* adjList, int node, int parent, int\* size) {

    size[node] = 1;

    Node\* current = adjList[node].head;

    while (current) {

        if (current->data != parent) {

            dfs(adjList, current->data, node, size);

            size[node] += size[current->data];

        }

        current = current->next;

    }

}

// Function to calculate the minimum fuel cost

long long minimumFuelCost(int\*\* roads, int roadsSize, int\* roadsColSize, int seats) {

    int n = roadsSize + 1;

    // Initialize size array to store subtree sizes

    int\* size = (int\*)calloc(n, sizeof(int));

    // Initialize adjacency list

    List\* adjList = (List\*)malloc(n \* sizeof(List));

    for (int i = 0; i < n; i++) {

        adjList[i] = initializeList();

    }

    // Construct the adjacency list from the input roads

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

        int a = roads[i][0];

        int b = roads[i][1];

        addEdge(&adjList, a, b);

    }

    // Perform depth-first search (DFS) to calculate subtree sizes

    dfs(adjList, 0, -1, size);

    // Calculate the minimum fuel cost

    long long ans = 0;

    for (int i = 1; i < n; ++i) {

        int cnt = size[i];

        ans += 1ll \* ((cnt + seats - 1) / seats);

    }

    // Free dynamically allocated memory

    for (int i = 0; i < n; i++) {

        Node\* current = adjList[i].head;

        while (current) {

            Node\* temp = current;

            current = current->next;

            free(temp);

        }

    }

    free(adjList);

    free(size);

    return ans;

}

int main() {

    int roadsSize = 6;

    int roadsColSize[] = {2, 2, 2, 2, 2, 2};

    int\*\* roads = (int\*\*)malloc(roadsSize \* sizeof(int\*));

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

        roads[i] = (int\*)malloc(2 \* sizeof(int));

    }

    roads[0][0] = 3; roads[0][1] = 1;

    roads[1][0] = 3; roads[1][1] = 2;

    roads[2][0] = 1; roads[2][1] = 0;

    roads[3][0] = 0; roads[3][1] = 4;

    roads[4][0] = 0; roads[4][1] = 5;

    roads[5][0] = 4; roads[5][1] = 6;

    int seats = 2;

    long long result = minimumFuelCost(roads, roadsSize, roadsColSize, seats);

    printf("Minimum fuel cost: %lld\n", result);

    // Free dynamically allocated memory for roads

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

        free(roads[i]);

    }

    free(roads);

    return 0;

}

**Input:** roads = [[3,1],[3,2],[1,0],[0,4],[0,5],[4,6]], seats = 2

**Output:** 7

**Explanation:**

- Representative2 goes directly to city 3 with 1 liter of fuel.

- Representative2 and representative3 go together to city 1 with 1 liter of fuel.

- Representative2 and representative3 go together to the capital with 1 liter of fuel.

- Representative1 goes directly to the capital with 1 liter of fuel.

- Representative5 goes directly to the capital with 1 liter of fuel.

- Representative6 goes directly to city 4 with 1 liter of fuel.

- Representative4 and representative6 go together to the capital with 1 liter of fuel.

It costs 7 liters of fuel at minimum.

It can be proven that 7 is the minimum number of liters of fuel needed.

Time/Space Complexity:O(V+E)/O(V+E)

## Remove Duplicate Letters(lexicographical small)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* removeDuplicateLetters(char\* s) {

    int n = strlen(s);

    // Array to store the last occurrence index of each character

    int lastIndex[26] = {0};

    for (int i = 0; i < n; i++) {

        lastIndex[s[i] - 'a'] = i;

    }

    // Array to keep track of seen characters

    int seen[26] = {0};

    // Stack to maintain selected characters

    int stack[26];

    int top = -1;

    // Iterate through the string

    for (int i = 0; i < n; i++) {

        int curr = s[i] - 'a';

        // If the character is already seen, continue to the next iteration

        if (seen[curr]) {

            continue;

        }

        // Pop characters from the stack if the current character is smaller,

        // there are remaining occurrences of the characters in the stack,

        // and the character at the top of the stack is greater than the current character

        while (top >= 0 && curr < stack[top] && i < lastIndex[stack[top]]) {

            seen[stack[top--]] = 0; // Pop out and mark unseen

        }

        // Push the current character onto the stack and mark it as seen

        stack[++top] = curr;

        seen[curr] = 1; // Mark seen

    }

    // Build the result string from the characters in the stack

    char\* result = (char\*)malloc((top + 2) \* sizeof(char));

    for (int i = 0; i <= top; i++) {

        result[i] = stack[i] + 'a';

    }

    result[top + 1] = '\0';

    return result;

}

// Main function for testing the removeDuplicateLetters function

int main() {

    char s[] = "bcabc";

    char\* result = removeDuplicateLetters(s);

    printf("Original string: %s\n", s);

    printf("String after removing duplicate letters: %s\n", result);

    free(result);

    return 0;

}

Output:abc

Time/Space complexity:O(n)/O(1)

## Score of Parantheses

#include <stdio.h>

int scoreOfParentheses(char\* S) {

    int ans = 0, bal = 0;

    for (int i = 0; S[i] != '\0'; ++i) {

        if (S[i] == '(') {

            bal++;

        } else {

            bal--;

            if (S[i - 1] == '(') {

                ans += 1 << bal;

            }

        }

    }

    return ans;

}

int main() {

    // Test cases

    char\* test1 = "(()())";

    char\* test2 = "(()(()))";

    char\* test3 = "()()()";

    char\* test4 = "()()";

    printf("Score of %s: %d\n", test1, scoreOfParentheses(test1));

    printf("Score of %s: %d\n", test2, scoreOfParentheses(test2));

    printf("Score of %s: %d\n", test3, scoreOfParentheses(test3));

    printf("Score of %s: %d\n", test4, scoreOfParentheses(test4));

    return 0;

}

**Input:** s = "()"

**Output:** 1

**Input:** s = "(())"

**Output:** 2

Time/Space Complexity:O(n)/O(1)

## Maximum Number Of Balloons

#include <stdio.h>

#include <string.h>

#define min(a,b) ((a) < (b) ? (a) : (b))

int maxNumberOfBalloons(char\* text) {

    int bCount = 0, aCount = 0, lCount = 0, oCount = 0, nCount = 0;

    // Count the frequency of all the five characters

    for (int i = 0; i < strlen(text); i++) {

        if (text[i] == 'b') {

            bCount++;

        } else if (text[i] == 'a') {

            aCount++;

        } else if (text[i] == 'l') {

            lCount++;

        } else if (text[i] == 'o') {

            oCount++;

        } else if (text[i] == 'n') {

            nCount++;

        }

    }

    // Find the potential of each character.

    // Except for 'l' and 'o', the potential is equal to the frequency.

    lCount = lCount / 2;

    oCount = oCount / 2;

    // Find the bottleneck.

    int ret = min(oCount, nCount);

    ret = min(ret, lCount);

    ret = min(ret, aCount);

    ret = min(ret, bCount);

    return ret;

}

int main() {

    // Test cases

    char\* test1 = "nlaebolko";

    char\* test2 = "loonbalxballpoon";

    char\* test3 = "nlaablonn";

    printf("Maximum number of balloons in \"%s\": %d\n", test1, maxNumberOfBalloons(test1));

    printf("Maximum number of balloons in \"%s\": %d\n", test2, maxNumberOfBalloons(test2));

    printf("Maximum number of balloons in \"%s\": %d\n", test3, maxNumberOfBalloons(test3));

    return 0;

}

Output: 1/ 2

Time/Space Complexity:O(n)/O(1)

## Buy Two Chocolates

#include <stdio.h>

int buyChoco(int\* prices, int pricesSize, int money) {

    // Assume minimum and second minimum

    int minimum = prices[0] < prices[1] ? prices[0] : prices[1];

    int secondMinimum = prices[0] > prices[1] ? prices[0] : prices[1];

    // Iterate over the remaining elements

    for (int i = 2; i < pricesSize; i++) {

        if (prices[i] < minimum) {

            secondMinimum = minimum;

            minimum = prices[i];

        } else if (prices[i] < secondMinimum) {

            secondMinimum = prices[i];

        }

    }

    // Minimum Cost

    int minCost = minimum + secondMinimum;

    // We can buy chocolates only if we have enough money

    if (minCost <= money) {

        // Return the Amount of Money Left

        return money - minCost;

    }

    // We cannot buy chocolates. Return the initial amount of money

    return money;

}

int main() {

    // Test cases

    int prices1[] = {1,2,2};

    int money1 = 3;

    int prices2[] = {3,2,3};

    int money2 = 3;

    printf("Money left after buying chocolates: %d\n", buyChoco(prices1, sizeof(prices1) / sizeof(prices1[0]), money1));

    printf("Money left after buying chocolates: %d\n", buyChoco(prices2, sizeof(prices2) / sizeof(prices2[0]), money2));

    return 0;

}

Output:0/3

Time/Space Complexity: O(n)/O(1)

## Permutations

#include <stdio.h>

#include <stdlib.h>

void swap(int\* a, int\* b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

void permute(int\* nums, int start, int end) {

    if (start == end) {

        // Print the current permutation

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

            printf("%d ", nums[i]);

        }

        printf("\n");

        return;

    }

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

        // Swap elements to create different permutations

        swap(&nums[start], &nums[i]);

        // Recursively generate permutations for the remaining elements

        permute(nums, start + 1, end);

        // Backtrack (undo the swap) to explore other possibilities

        swap(&nums[start], &nums[i]);

    }

}

int main() {

    int n;

    printf("Enter the number of elements: ");

    scanf("%d", &n);

    int\* nums = (int\*)malloc(n \* sizeof(int));

    printf("Enter the elements: ");

    for (int i = 0; i < n; i++) {

        scanf("%d", &nums[i]);

    }

    printf("Permutations:\n");

    permute(nums, 0, n - 1);

    free(nums);

    return 0;

}

Enter the number of elements: 4

Enter the elements: 1

2

3

4

Permutations:

1 2 3 4

1 2 4 3

1 3 2 4

1 3 4 2

1 4 3 2

1 4 2 3

2 1 3 4

2 1 4 3

2 3 1 4

2 3 4 1

2 4 3 1

2 4 1 3

3 2 1 4

3 2 4 1

3 1 2 4

3 1 4 2

3 4 1 2

3 4 2 1

4 2 3 1

4 2 1 3

4 3 2 1

4 3 1 2

4 1 3 2

4 1 2 3

## Combinations

#include <stdio.h>

#include <stdlib.h>

void combine(int\* nums, int n, int k, int start, int\* current, int currentSize) {

    if (currentSize == k) {

        // Print the current combination

        for (int i = 0; i < k; i++) {

            printf("%d ", current[i]);

        }

        printf("\n");

        return;

    }

    for (int i = start; i < n; i++) {

        current[currentSize] = nums[i];

        // Recursively generate combinations for the remaining elements

        combine(nums, n, k, i + 1, current, currentSize + 1);

    }

}

int main() {

    int n, k;

    printf("Enter the number of elements: ");

    scanf("%d", &n);

    int\* nums = (int\*)malloc(n \* sizeof(int));

    printf("Enter the elements: ");

    for (int i = 0; i < n; i++) {

        scanf("%d", &nums[i]);

    }

    printf("Enter the size of combinations (k): ");

    scanf("%d", &k);

    if (k > n) {

        printf("Invalid input! Size of combinations (k) cannot be greater than the number of elements.\n");

        free(nums);

        return 1;

    }

    int\* current = (int\*)malloc(k \* sizeof(int));

    printf("Combinations:\n");

    combine(nums, n, k, 0, current, 0);

    free(nums);

    free(current);

    return 0;

}

Enter the number of elements: 3

Enter the elements: 1

2

3

Enter the size of combinations (k): 2

Combinations:

1 2

1 3

2 3

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

void backtrack(int n, int k, int start, int\* current, int\*\* result, int\* returnSize, int\*\* returnColumnSizes, int currentSize) {

    if (currentSize == k) {

        // Add the current combination to the result

        result[\*returnSize] = (int\*)malloc(k \* sizeof(int));

        for (int i = 0; i < k; i++) {

            result[\*returnSize][i] = current[i];

        }

        (\*returnColumnSizes)[\*returnSize] = k;

        (\*returnSize)++;

        return;

    }

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

        current[currentSize] = i;

        // Recursively generate combinations

        backtrack(n, k, i + 1, current, result, returnSize, returnColumnSizes, currentSize + 1);

    }

}

int\*\* combine(int n, int k, int\* returnSize, int\*\* returnColumnSizes) {

    \*returnSize = 0;

    // Allocate memory for the result

    int\*\* result = (int\*\*)malloc(10000 \* sizeof(int\*)); // Assuming a maximum limit for the returnSize

    \*returnColumnSizes = (int\*)malloc(10000 \* sizeof(int));

    int\* current = (int\*)malloc(k \* sizeof(int));

    // Start backtracking from the first element

    backtrack(n, k, 1, current, result, returnSize, returnColumnSizes, 0);

    free(current);

    return result;

}

## Remove K Digits (return smallest possible integer)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* removeKdigits(char\* num, int k) {

    int n = strlen(num);

    // Check if k is equal to the length of the number

    if (k == n) {

        char\* result = (char\*)malloc(2 \* sizeof(char));

        result[0] = '0';

        result[1] = '\0';

        return result;

    }

    // Initialize a stack to keep track of the digits

    char\* stack = (char\*)malloc((n + 1) \* sizeof(char));

    int top = -1;

    // Iterate through the digits of the number

    for (int i = 0; i < n; i++) {

        // Pop elements from the stack if the current digit is smaller than the top of the stack

        while (k > 0 && top >= 0 && num[i] < stack[top]) {

            top--;

            k--;

        }

        // Push the current digit onto the stack

        stack[++top] = num[i];

    }

    // Pop remaining elements from the stack to remove excess digits

    while (k > 0 && top >= 0) {

        top--;

        k--;

    }

    // Find the first non-zero digit in the stack

    int nonZeroIndex = 0;

    while (nonZeroIndex <= top && stack[nonZeroIndex] == '0') {

        nonZeroIndex++;

    }

    // If the resulting number is empty, return "0"

    if (nonZeroIndex > top) {

        char\* result = (char\*)malloc(2 \* sizeof(char));

        result[0] = '0';

        result[1] = '\0';

        free(stack);

        return result;

    }

    // Construct the resulting number

    char\* result = (char\*)malloc((top - nonZeroIndex + 2) \* sizeof(char));

    int resultIndex = 0;

    for (int i = nonZeroIndex; i <= top; i++) {

        result[resultIndex++] = stack[i];

    }

    result[resultIndex] = '\0';

    free(stack);

    return result;

}

int main() {

    char num[] = "1432219";

    int k = 3;

    char\* result = removeKdigits(num, k);

    printf("Resulting number after removing %d digits: %s\n", k, result);

    free(result);

    return 0;

}

Output:1219

## Car Fleet (arrive at destination)

#include <stdio.h>

#include <stdlib.h>

int carFleet(int target, int\* position, int positionSize, int\* speed, int speedSize) {

    int res = 0;

    double\* timeArr = (double\*)malloc(target \* sizeof(double));

    // Calculate time to destination for each car

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

        timeArr[position[i]] = (double)(target - position[i]) / speed[i];

    }

    double prev = 0.0;

    for (int i = target - 1; i >= 0; i--) {

        double cur = timeArr[i];

        if (cur > prev) {

            prev = cur;

            res++;

        }

    }

    free(timeArr);

    return res;

}

int main() {

    int target = 12;

    int position[] = {10, 8, 0, 5, 3};

    int positionSize = sizeof(position) / sizeof(position[0]);

    int speed[] = {2, 4, 1, 1, 3};

    int speedSize = sizeof(speed) / sizeof(speed[0]);

    int fleet = carFleet(target, position, positionSize, speed, speedSize);

    printf("Number of car fleets: %d\n", fleet);

    return 0;

}

Output:3

**Input:** target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]

**Output:** 3

**Explanation:**

The cars starting at 10 (speed 2) and 8 (speed 4) become a fleet, meeting each other at 12.

The car starting at 0 does not catch up to any other car, so it is a fleet by itself.

The cars starting at 5 (speed 1) and 3 (speed 3) become a fleet, meeting each other at 6. The fleet moves at speed 1 until it reaches target.

Note that no other cars meet these fleets before the destination, so the answer is 3.

## Match Sticks To Square

#include <stdio.h>

#include <stdbool.h>

bool canMake(int index, int bucketsize, int\* matchsticks, int\* bucket, int matchstickSize) {

    if (index == matchstickSize)

        return bucket[0] == bucket[1] && bucket[1] == bucket[2] && bucket[2] == bucket[3];

    for (int i = 0; i < 4; i++) {

        if (bucket[i] + matchsticks[index] > bucketsize)

            continue;

        int j = i;

        while (--j >= 0)

            if (bucket[i] == bucket[j])

                break;

        if (j != -1)

            continue;

        bucket[i] += matchsticks[index];

        if (canMake(index + 1, bucketsize, matchsticks, bucket, matchstickSize))

            return true;

        bucket[i] -= matchsticks[index];

    }

    return false;

}

bool makesquare(int\* matchsticks, int matchstickSize) {

    int sum = 0;

    for (int i = 0; i < matchstickSize; i++)

        sum += matchsticks[i];

    if (sum == 0 || sum % 4)

        return false;

    for (int i = 0; i < matchstickSize - 1; i++) {

        for (int j = 0; j < matchstickSize - i - 1; j++) {

            if (matchsticks[j] < matchsticks[j + 1]) {

                // swap

                int temp = matchsticks[j];

                matchsticks[j] = matchsticks[j + 1];

                matchsticks[j + 1] = temp;

            }

        }

    }

    int bucket[4] = {0};

    return canMake(0, sum / 4, matchsticks, bucket, matchstickSize);

}

int main() {

    int matchsticks[] = {1, 1, 2, 2, 2};

    int matchstickSize = sizeof(matchsticks) / sizeof(matchsticks[0]);

    if (makesquare(matchsticks, matchstickSize)) {

        printf("It is possible to form a square with the given matchsticks.\n");

    } else {

        printf("It is not possible to form a square with the given matchsticks.\n");

    }

    return 0;

}

A square black frame with fire flames

Description automatically generated with medium confidence

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

**Output:** true

**Explanation:** You can form a square with length 2, one side of the square came two sticks with length 1.

## 132 Pattern

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h>

bool find132pattern(int\* nums, int numsSize) {

    if (numsSize < 3) {

        return false;

    }

    int\* min = malloc(numsSize \* sizeof(int));

    min[0] = nums[0];

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

        min[i] = fmin(min[i - 1], nums[i]);

    }

    int\* stack = malloc(numsSize \* sizeof(int));

    int top = -1;

    for (int j = numsSize - 1; j >= 0; j--) {

        if (nums[j] > min[j]) {

            while (top != -1 && stack[top] <= min[j]) {

                stack[top--];

            }

            if (top != -1 && stack[top] < nums[j]) {

                free(min);

                free(stack);

                return true;

            }

            stack[++top] = nums[j];

        }

    }

    free(min);

    free(stack);

    return false;

}

int main() {

    int nums[] = {3, 1, 4, 2};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    if (find132pattern(nums, numsSize)) {

        printf("The pattern 132 exists in the array.\n");

    } else {

        printf("The pattern 132 does not exist in the array.\n");

    }

    return 0;

}

Output: [1, 4, 2].

#include <limits.h>

#include <stdbool.h>

#include <stdlib.h>

#include <stdio.h>

typedef struct {

int \*data;

int top;

int capacity;

} Stack;

// Function to initialize the stack

Stack \*createStack(int capacity) {

Stack \*stack = (Stack \*)malloc(sizeof(Stack));

stack->data = (int \*)malloc(capacity \* sizeof(int));

stack->top = -1;

stack->capacity = capacity;

return stack;

}

// Function to check if the stack is empty

bool isEmpty(Stack \*stack) {

return stack->top == -1;

}

// Function to check if the stack is full

bool isFull(Stack \*stack) {

return stack->top == stack->capacity - 1;

}

// Function to push an element onto the stack

void push(Stack \*stack, int value) {

if (isFull(stack)) {

printf("Stack overflow\n");

return;

}

stack->data[++stack->top] = value;

}

// Function to pop an element from the stack

int pop(Stack \*stack) {

if (isEmpty(stack)) {

printf("Stack underflow\n");

return INT\_MIN;

}

return stack->data[stack->top--];

}

// Function to get the top element of the stack without popping

int peek(Stack \*stack) {

if (isEmpty(stack)) {

return INT\_MIN;

}

return stack->data[stack->top];

}

// Function to free the memory occupied by the stack

void freeStack(Stack \*stack) {

free(stack->data);

free(stack);

}

// Function to find the 132 pattern

bool find132pattern(int \*nums, int numsSize) {

int thirdValue = INT\_MIN;

Stack \*candidates = createStack(numsSize);

for (int i = numsSize - 1; i >= 0; --i) {

if (nums[i] < thirdValue) {

// Found a valid 132 pattern

freeStack(candidates);

return true;

}

while (!isEmpty(candidates) && peek(candidates) < nums[i]) {

thirdValue = pop(candidates);

}

push(candidates, nums[i]);

}

// No 132 pattern found

freeStack(candidates);

return false;

}

int main() {

// Example usage

int nums[] = {3, 1, 4, 2};

int numsSize = sizeof(nums) / sizeof(nums[0]);

bool result = find132pattern(nums, numsSize);

if (result) {

printf("Found 132 pattern\n");

} else {

printf("No 132 pattern found\n");

}

return 0;

}

## Word Search II

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct TrieNode {

    struct TrieNode\* children[26];

    char\* word;

} TrieNode;

// Trie Initialization

TrieNode\* createTrieNode() {

    TrieNode\* node = (TrieNode\*)malloc(sizeof(TrieNode));

    memset(node->children, 0, sizeof(node->children));

    node->word = NULL;

    return node;

}

// Trie Insertion

void insertWord(TrieNode\* root, char\* word) {

    TrieNode\* node = root;

    for (int i = 0; i < strlen(word); i++) {

        int index = word[i] - 'a';

        if (!node->children[index]) {

            node->children[index] = createTrieNode();

        }

        node = node->children[index];

    }

    node->word = word;

}

// DFS to find words on the board

void findWordsDFS(char\*\* board, int boardSize, int\* boardColSize, int row, int col, TrieNode\* node, char\*\*\* result, int\* returnSize) {

    char c = board[row][col];

    TrieNode\* child = node->children[c - 'a'];

    if (child == NULL) {

        return;

    }

    if (child->word != NULL) {

        (\*returnSize)++;

        \*result = (char\*\*)realloc(\*result, (\*returnSize) \* sizeof(char\*));

        (\*result)[(\*returnSize) - 1] = child->word;

        child->word = NULL; // Avoid duplicates

    }

    board[row][col] = '#'; // Mark visited

    int directions[][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    for (int i = 0; i < 4; i++) {

        int newRow = row + directions[i][0];

        int newCol = col + directions[i][1];

        if (newRow >= 0 && newRow < boardSize && newCol >= 0 && newCol < boardColSize[0] && board[newRow][newCol] != '#') {

            findWordsDFS(board, boardSize, boardColSize, newRow, newCol, child, result, returnSize);

        }

    }

    board[row][col] = c; // Restore the original character

}

// Find Words on the Board

char\*\* findWords(char\*\* board, int boardSize, int\* boardColSize, char\*\* words, int wordsSize, int\* returnSize) {

    \*returnSize = 0;

    char\*\* result = NULL;

    TrieNode\* root = createTrieNode();

    // Insert words into the Trie

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

        insertWord(root, words[i]);

    }

    // Traverse the board and search for words

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

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

            findWordsDFS(board, boardSize, boardColSize, i, j, root, &result, returnSize);

        }

    }

    // Free Trie and return result

    free(root);

    return result;

}

int main() {

    char\* board[] = {"oaan", "etae", "ihkr", "iflv"};

    int boardSize = sizeof(board) / sizeof(board[0]);

    int boardColSize[] = {4, 4, 4, 4};

    char\* words[] = {"oath", "pea", "eat", "rain"};

    int wordsSize = sizeof(words) / sizeof(words[0]);

    int returnSize = 0;

    char\*\* result = findWords(board, boardSize, boardColSize, words, wordsSize, &returnSize);

    printf("Found words:\n");

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

        printf("%s\n", result[i]);

    }

    // Free allocated memory

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

        free(result[i]);

    }

    free(result);

    return 0;

}

**Example 1:**

A close-up of a crossword

Description automatically generated

**Input:** board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]

**Output:** ["eat","oath"]

## Merge Intervals

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

**Example 1:**

**Input:** intervals = [[1,3],[2,6],[8,10],[15,18]]

**Output:** [[1,6],[8,10],[15,18]]

**Explanation:** Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

**Example 2:**

**Input:** intervals = [[1,4],[4,5]]

**Output:** [[1,5]]

**Explanation:** Intervals [1,4] and [4,5] are considered overlapping.

**C :**

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

// Definition for an interval.

#include <stdio.h>

#include <stdlib.h>

typedef struct {

    int start;

    int end;

} Interval;

int compareIntervals(const void\* a, const void\* b) {

    return ((Interval\*)a)->start - ((Interval\*)b)->start;

}

int\*\* merge(int\*\* intervals, int intervalsSize, int\* intervalsColSize, int\* returnSize, int\*\* returnColumnSizes) {

    if (intervalsSize <= 0) {

        \*returnSize = 0;

        \*returnColumnSizes = NULL;

        return NULL;

    }

    // Convert the input to Interval structures

    Interval\* intervalsStruct = (Interval\*)malloc(intervalsSize \* sizeof(Interval));

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

        intervalsStruct[i].start = intervals[i][0];

        intervalsStruct[i].end = intervals[i][1];

    }

    // Sort intervals based on the start time

    qsort(intervalsStruct, intervalsSize, sizeof(Interval), compareIntervals);

    // Initialize result array

    int\*\* result = (int\*\*)malloc(intervalsSize \* sizeof(int\*));

    \*returnSize = 0;

    \*returnColumnSizes = (int\*)malloc(intervalsSize \* sizeof(int));

    // Merge overlapping intervals

    result[\*returnSize] = (int\*)malloc(2 \* sizeof(int));

    result[\*returnSize][0] = intervalsStruct[0].start;

    result[\*returnSize][1] = intervalsStruct[0].end;

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

        if (result[\*returnSize][1] >= intervalsStruct[i].start) {

            // Merge overlapping intervals

            result[\*returnSize][1] = (result[\*returnSize][1] > intervalsStruct[i].end) ? result[\*returnSize][1] : intervalsStruct[i].end;

        } else {

            // Move to the next non-overlapping interval

            (\*returnSize)++;

            result[\*returnSize] = (int\*)malloc(2 \* sizeof(int));

            result[\*returnSize][0] = intervalsStruct[i].start;

            result[\*returnSize][1] = intervalsStruct[i].end;

        }

    }

    (\*returnSize)++;

    \*returnColumnSizes = (int\*)realloc(\*returnColumnSizes, \*returnSize \* sizeof(int));

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

        (\*returnColumnSizes)[i] = 2;

    }

    // Clean up

    free(intervalsStruct);

    return result;

}

int main() {

    int intervalsSize = 4;

    int intervalsColSize[] = {2, 2, 2, 2};

    int\*\* intervals = (int\*\*)malloc(intervalsSize \* sizeof(int\*));

    intervals[0] = (int\*)malloc(2 \* sizeof(int));

    intervals[0][0] = 1;

    intervals[0][1] = 3;

    intervals[1] = (int\*)malloc(2 \* sizeof(int));

    intervals[1][0] = 2;

    intervals[1][1] = 6;

    intervals[2] = (int\*)malloc(2 \* sizeof(int));

    intervals[2][0] = 8;

    intervals[2][1] = 10;

    intervals[3] = (int\*)malloc(2 \* sizeof(int));

    intervals[3][0] = 15;

    intervals[3][1] = 18;

    int returnSize;

    int\* returnColumnSizes;

    int\*\* mergedIntervals = merge(intervals, intervalsSize, intervalsColSize, &returnSize, &returnColumnSizes);

    printf("Merged Intervals:\n");

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

        printf("[%d, %d]\n", mergedIntervals[i][0], mergedIntervals[i][1]);

    }

    // Free allocated memory

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

        free(intervals[i]);

    }

    free(intervals);

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

        free(mergedIntervals[i]);

    }

    free(mergedIntervals);

    free(returnColumnSizes);

    return 0;

}

**Input:** intervals = [[1,3],[2,6],[8,10],[15,18]]

**Output:** [[1,6],[8,10],[15,18]]

**Explanation:** Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

## Merge Sorted Array

You are given two integer arrays nums1 and nums2, sorted in **non-decreasing order**, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

**Merge** nums1 and nums2 into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array*nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

**Example 1:**

**Input:** nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3

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

**Explanation:** The arrays we are merging are [1,2,3] and [2,5,6].

The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

**Example 2:**

**Input:** nums1 = [1], m = 1, nums2 = [], n = 0

**Output:** [1]

**Explanation:** The arrays we are merging are [1] and [].

The result of the merge is [1].

**Example 3:**

**Input:** nums1 = [0], m = 0, nums2 = [1], n = 1

**Output:** [1]

**Explanation:** The arrays we are merging are [] and [1].

The result of the merge is [1].

Note that because m = 0, there are no elements in nums1. The 0 is only there to ensure the merge result can fit in nums1.

**C :**

#include <stdio.h>

void merge(int\* nums1, int nums1Size, int m, int\* nums2, int nums2Size, int n){

    int i = m - 1;

    int j = n - 1;

    int k = m + n - 1;

    while (i >= 0 && j >= 0) {

        if (nums1[i] < nums2[j]) {

            nums1[k--] = nums2[j--];

        } else {

            nums1[k--] = nums1[i--];

        }

    }

    // If there are remaining elements in nums2, copy them to nums1

    while (i >= 0) {

        nums1[k--] = nums1[i--];

    }

    while (j >= 0) {

        nums1[k--] = nums2[j--];

    }

}

int main() {

    int nums1[] = {1, 2, 3, 0, 0, 0};

    int m = 3;

    int nums2[] = {2, 5, 6};

    int n = 3;

    int nums1Size = m + n;

    merge(nums1, nums1Size, m, nums2, n, n);

    printf("Merged Array: ");

    for (int i = 0; i < nums1Size; i++) {

        printf("%d ", nums1[i]);

    }

    printf("\n");

    return 0;

}

## Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

**Clarification:** The input/output format is the same as [how LeetCode serializes a binary tree](https://support.leetcode.com/hc/en-us/articles/360011883654-What-does-1-null-2-3-mean-in-binary-tree-representation-). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

**Example 1:**

A diagram of a network

Description automatically generated

**Input:** root = [1,2,3,null,null,4,5]

**Output:** [1,2,3,null,null,4,5]

**Example 2:**

**Input:** root = []

**Output:** []

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Definition for a binary tree node

struct TreeNode {

    int val;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

// Serialize helper function

void serializeHelper(struct TreeNode\* root, char\* result, int\* index) {

    if (root == NULL) {

        // Append "null," to the result string

        sprintf(result + \*index, "null,");

        (\*index) += strlen("null,");

    } else {

        // Convert the node value to a string and append it to the result

        sprintf(result + \*index, "%d,", root->val);

        (\*index) += strlen(result + \*index);

        // Recursively serialize the left and right subtrees

        serializeHelper(root->left, result, index);

        serializeHelper(root->right, result, index);

    }

}

// Deserialize helper function

struct TreeNode\* deserializeHelper(char\* data, int\* index) {

    if (data[\*index] == '\0' || strncmp(data + \*index, "null", 4) == 0) {

        // End of string or "null" node

        \*index += 5;  // Move index to the next position after "null,"

        return NULL;

    }

    // Extract the node value from the data string

    int val;

    sscanf(data + \*index, "%d,", &val);

    (\*index) += numDigits(val) + 1;  // Move index to the next position after the comma

    // Create a new tree node with the extracted value

    struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    newNode->val = val;

    // Recursively deserialize the left and right subtrees

    newNode->left = deserializeHelper(data, index);

    newNode->right = deserializeHelper(data, index);

    return newNode;

}

// Serialize function

char\* serialize(struct TreeNode\* root) {

    // Allocate memory for the result string

    char\* result = (char\*)malloc(1024 \* sizeof(char));

    int index = 0;

    // Call the serializeHelper to populate the result string

    serializeHelper(root, result, &index);

    // Null-terminate the result string

    result[index] = '\0';

    // Resize the result buffer to the actual size needed

    result = (char\*)realloc(result, (index + 1) \* sizeof(char));

    return result;

}

// Deserialize function

struct TreeNode\* deserialize(char\* data) {

    int index = 0;

    return deserializeHelper(data, &index);

}

// Utility function to calculate the number of digits in an integer

int numDigits(int num) {

    int count = 0;

    while (num != 0) {

        num /= 10;

        count++;

    }

    return count;

}

// Function to print the inorder traversal of a binary tree (for verification)

void inorderTraversal(struct TreeNode\* root) {

    if (root != NULL) {

        inorderTraversal(root->left);

        printf("%d ", root->val);

        inorderTraversal(root->right);

    }

}

int main() {

    // Construct a binary tree for testing

    struct TreeNode\* root = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->val = 1;

    root->left = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->left->val = 2;

    root->left->left = NULL;

    root->left->right = NULL;

    root->right = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->right->val = 3;

    root->right->left = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->right->left->val = 4;

    root->right->right = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->right->right->val = 5;

    // Serialize the binary tree

    char\* serializedTree = serialize(root);

    printf("Serialized tree: %s\n", serializedTree);

    // Deserialize the serialized string

    struct TreeNode\* deserializedRoot = deserialize(serializedTree);

    // Print the inorder traversal of the deserialized tree for verification

    printf("Inorder traversal of deserialized tree: ");

    inorderTraversal(deserializedRoot);

    printf("\n");

    // Clean up memory

    free(root->left);

    free(root->right);

    free(root);

    free(serializedTree);

    free(deserializedRoot);

    return 0;

}

A diagram of a network

Description automatically generated

**Input:** root = [1,2,3,null,null,4,5]

**Output:** [1,2,3,null,null,4,5]

Time/Space Complexity: O(n)/O(n)

## Bulb Switcher

#include <stdio.h>

#include <math.h>

int bulbSwitch(int n) {

    return sqrt(n);

}

int main() {

    int n;

    printf("Enter the number of bulbs: ");

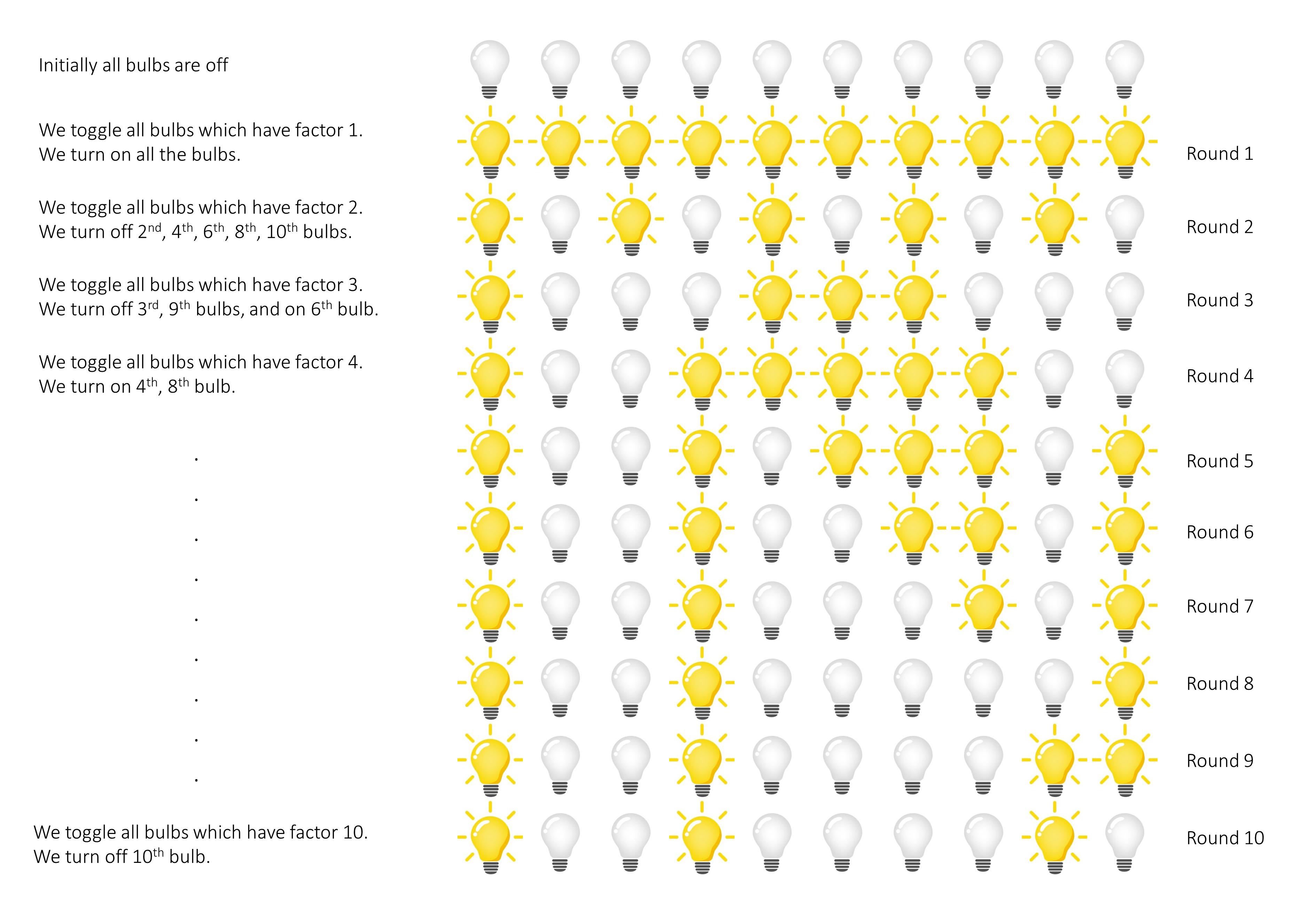
    scanf("%d", &n);

    int result = bulbSwitch(n);

    printf("Number of bulbs on after n rounds: %d\n", result);

    return 0;

}



## Sign Of The Product Of An Array

#include <stdio.h>

int arraySign(int\* nums, int numsSize) {

    int countNegatives = 0;

    int hasZero = 0;

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

        if (nums[i] == 0) {

            hasZero = 1;

            break;

        } else if (nums[i] < 0) {

            countNegatives++;

        }

    }

    if (hasZero) {

        return 0;

    }

    return countNegatives % 2 == 0 ? 1 : -1;

}

int main() {

    int nums[] = { -1, 2, -3, 4, 0 };

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int result = arraySign(nums, numsSize);

    printf("Sign of the product of all elements: %d\n", result);

    return 0;

}

## Excel Sheet Column Title

#include <stdio.h>

#include <stdlib.h>

char\* convertToTitle(int columnNumber) {

    // Calculate the length of the result string

    int len = 0;

    int temp = columnNumber;

    while (temp > 0) {

        len++;

        temp = (temp - 1) / 26;

    }

    // Allocate memory for the result string

    char\* result = (char\*)malloc((len + 1) \* sizeof(char));

    result[len] = '\0';

    // Fill in the result string from right to left

    for (int i = len - 1; i >= 0; i--) {

        result[i] = (columnNumber - 1) % 26 + 'A';

        columnNumber = (columnNumber - 1) / 26;

    }

    return result;

}

int main() {

    int columnNumber = 701; // Example input

    char\* title = convertToTitle(columnNumber);

    printf("Column title for column number %d: %s\n", columnNumber, title);

    free(title); // Free allocated memory

    return 0;

}

For example:

A -> 1

B -> 2

C -> 3

...

Z -> 26

AA -> 27

AB -> 28

...

## Remove All Adjacent Duplicates In String (K duplicate removal)

You are given a string s and an integer k, a k **duplicate removal** consists of choosing k adjacent and equal letters from s and removing them, causing the left and the right side of the deleted substring to concatenate together.

We repeatedly make k **duplicate removals** on s until we no longer can.

Return *the final string after all such duplicate removals have been made*. It is guaranteed that the answer is **unique**.

**Example 1:**

**Input:** s = "abcd", k = 2

**Output:** "abcd"

**Explanation:** There's nothing to delete.

**Example 2:**

**Input:** s = "deeedbbcccbdaa", k = 3

**Output:** "aa"

**Explanation:**

First delete "eee" and "ccc", get "ddbbbdaa"

Then delete "bbb", get "dddaa"

Finally delete "ddd", get "aa"

**Example 3:**

**Input:** s = "pbbcggttciiippooaais", k = 2

**Output:** "ps"

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct Stack {

    int \*arr;

    int top;

    int capacity;

};

// Function to initialize a stack

struct Stack\* createStack(int capacity) {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->capacity = capacity;

    stack->top = -1;

    stack->arr = (int\*)malloc(stack->capacity \* sizeof(int));

    return stack;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

    return stack->top == -1;

}

// Function to push an element onto the stack

void push(struct Stack\* stack, int item) {

    stack->arr[++stack->top] = item;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

    if (isEmpty(stack)) {

        return -1; // Stack underflow

    }

    return stack->arr[stack->top--];

}

// Function to remove duplicates in a string

char\* removeDuplicates(char\* s, int k) {

    int length = strlen(s);

    struct Stack\* counts = createStack(length);

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

        if (i == 0 || s[i] != s[i - 1]) {

            push(counts, 1);

        } else {

            int incremented = pop(counts) + 1;

            if (incremented == k) {

                memmove(s + i - k + 1, s + i + 1, length - i);

                i = i - k;

                length = length - k;

            } else {

                push(counts, incremented);

            }

        }

    }

    return s;

}

int main() {

    char s[] = "deeedbbcccbdaa"; // Example input string

    int k = 3; // Example value of k

    printf("Original string: %s\n", s);

    char\* result = removeDuplicates(s, k);

    printf("String after removing duplicates: %s\n", result);

    free(result); // Free dynamically allocated memory

    return 0;

}

## Number Of Islands

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return *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:** grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

**Output:** 1

**Example 2:**

**Input:** grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

]

**Output:** 3

**C :**

#include <stdio.h>

#include <stdlib.h>

void dfs(char\*\* grid, int gridSize, int gridColSize, int crow, int ccol) {

    if (crow < 0 || crow >= gridSize || ccol < 0 || ccol >= gridColSize || grid[crow][ccol] == '0') {

        return;

    }

    grid[crow][ccol] = '0';

    dfs(grid, gridSize, gridColSize, crow - 1, ccol);

    dfs(grid, gridSize, gridColSize, crow + 1, ccol);

    dfs(grid, gridSize, gridColSize, crow, ccol + 1);

    dfs(grid, gridSize, gridColSize, crow, ccol - 1);

}

int numIslands(char\*\* grid, int gridSize, int\* gridColSize) {

    if (grid == NULL || gridSize == 0 || gridColSize == NULL || \*gridColSize == 0) {

        return 0;

    }

    int num\_of\_islands = 0;

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

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

            if (grid[i][j] == '1') {

                num\_of\_islands++;

                dfs(grid, gridSize, \*gridColSize, i, j);

            }

        }

    }

    return num\_of\_islands;

}

int main() {

    // Example grid dimensions

    int gridSize = 4;

    int gridColSize[] = {5, 5, 5, 5}; // Column sizes for each row

    // Allocate memory for the grid

    char\*\* grid = (char\*\*)malloc(gridSize \* sizeof(char\*));

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

        grid[i] = (char\*)malloc((gridColSize[i] + 1) \* sizeof(char)); // +1 for the null terminator

    }

    // Populate the grid with the example data

    strcpy(grid[0], "11110");

    strcpy(grid[1], "11010");

    strcpy(grid[2], "11000");

    strcpy(grid[3], "00000");

    // Call the numIslands function

    int islands = numIslands(grid, gridSize, gridColSize);

    printf("Number of islands: %d\n", islands);

    // Free dynamically allocated memory

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

        free(grid[i]);

    }

    free(grid);

    return 0;

}

## Copy List With Random Pointer

A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) of the list. The deep copy should consist of exactly n **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random\_index] where:

* val: an integer representing Node.val
* random\_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.

Your code will **only** be given the head of the original linked list.

**Example 1:**

A diagram of a computer server

Description automatically generated

**Input:** head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Output:** [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Example 2:**

A diagram of a block

Description automatically generated

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

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

**Example 3:**

**A diagram of a computer server

Description automatically generated with medium confidence**

**Input:** head = [[3,null],[3,0],[3,null]]

**Output:** [[3,null],[3,0],[3,null]]

**C :**

/\*\*

 \* Definition for a Node.

 \* struct Node {

 \*     int val;

 \*     struct Node \*next;

 \*     struct Node \*random;

 \* };

 \*/

#include <stdio.h>

#include <stdlib.h>

// Definition for a Node

struct Node {

    int val;

    struct Node\* next;

    struct Node\* random;

};

// Function to copy a random linked list

struct Node\* copyRandomList(struct Node\* head) {

    if (head == NULL) {

        return NULL;

    }

    struct Node \*curr = head;

    while (curr != NULL) {

        struct Node\* new = (struct Node\*) malloc (sizeof(struct Node));

        new->val = curr->val;

        new->next = curr->next;

        new->random = NULL;

        curr->next = new;

        curr = new->next;

    }

    curr = head;

    while (curr != NULL) {

        if (curr->random != NULL) {

            curr->next->random = curr->random->next;

        }

        curr = curr->next->next;

    }

    curr= head;

    struct Node\* new\_head = head->next;

    struct Node \*new\_curr = new\_head;

    while (curr != NULL) {

        curr->next = new\_curr->next;

        curr = curr->next;

        if (curr != NULL) {

            new\_curr->next = curr->next;

            new\_curr = new\_curr->next;

        }

    }

    return new\_head;

}

int main() {

    // Create example nodes

    struct Node\* head = (struct Node\*)malloc(sizeof(struct Node));

    struct Node\* second = (struct Node\*)malloc(sizeof(struct Node));

    struct Node\* third = (struct Node\*)malloc(sizeof(struct Node));

    // Assign values

    head->val = 1;

    head->next = second;

    head->random = third;

    second->val = 2;

    second->next = third;

    second->random = head;

    third->val = 3;

    third->next = NULL;

    third->random = second;

    // Copy the linked list with random pointers

    struct Node\* newHead = copyRandomList(head);

    // Display original and copied list information

    printf("Original List:\n");

    struct Node\* curr = head;

    while (curr != NULL) {

        printf("Node val: %d, Random val: %d\n", curr->val, curr->random ? curr->random->val : -1);

        curr = curr->next;

    }

    printf("\nCopied List:\n");

    curr = newHead;

    while (curr != NULL) {

        printf("Node val: %d, Random val: %d\n", curr->val, curr->random ? curr->random->val : -1);

        curr = curr->next;

    }

    // Free allocated memory

    free(head);

    free(second);

    free(third);

    return 0;

}

## Two Sum

Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

**Input:** nums = [3,2,4], target = 6

**Output:** [1,2]

**Example 3:**

**Input:** nums = [3,3], target = 6

**Output:** [0,1]

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

typedef struct {

    int key;

    int value;

} hashEntry\_t;

typedef struct {

    hashEntry\_t \*table;

    int capacity;

} hashTable\_t;

void init(hashTable\_t \*t, int capacity) {

    t->table = (hashEntry\_t \*)malloc(sizeof(hashEntry\_t) \* capacity);

    t->capacity = capacity;

}

bool contains(hashTable\_t \*t, int complement, int \*index) {

    for (int i = 0; i < t->capacity; i++) {

        if (t->table[i].key == complement) {

            \*index = t->table[i].value;

            return true;

        }

    }

    return false;

}

int insert(hashTable\_t \*t, int number, int number\_index, int target) {

    int index = 0;

    int complement = target - number;

    if (contains(t, complement, &index)) {

        return index;

    } else {

        t->table[number\_index].key = number;

        t->table[number\_index].value = number\_index;

    }

    return -1;

}

int\* twoSum(int\* nums, int numsSize, int target, int\* returnSize) {

    hashTable\_t t;

    init(&t, numsSize);

    int \*result = (int \*)malloc(sizeof(int) \* 2);

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

        int j = insert(&t, nums[i], i, target);

        if (j != -1) {

            result[0] = i;

            result[1] = j;

            \*returnSize = 2;

            return result;

        }

    }

    free(t.table);

    return NULL;

}

int main() {

    int nums[] = {2, 7, 11, 15};

    int target = 9;

    int returnSize;

    int \*result = twoSum(nums, sizeof(nums) / sizeof(nums[0]), target, &returnSize);

    if (result != NULL) {

        printf("Indices: [%d, %d]\n", result[0], result[1]);

        free(result);

    } else {

        printf("No two elements found with the sum equal to the target.\n");

    }

    return 0;

}

## Flood Fill

An image is represented by an m x n integer grid image where image[i][j] represents the pixel value of the image.

You are also given three integers sr, sc, and color. You should perform a **flood fill** on the image starting from the pixel image[sr][sc].

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), and so on. Replace the color of all of the aforementioned pixels with color.

Return *the modified image after performing the flood fill*.

**Example 1:**

A diagram of a number

Description automatically generated

**Input:** image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

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

**Explanation:** From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) 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.

**Example 2:**

**Input:** image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0

**Output:** [[0,0,0],[0,0,0]]

**Explanation:** The starting pixel is already colored 0, so no changes are made to the image.

**C :**

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

#include <stdio.h>

#include <stdlib.h>

void dfs(int\*\* image, int imageSize, int\* imageColSize, int sr, int sc, int originalColor, int newColor) {

    // Base case: check if the current pixel is out of bounds or already colored with the new color

    if (sr < 0 || sr >= imageSize || sc < 0 || sc >= imageColSize[sr] || image[sr][sc] != originalColor) {

        return;

    }

    // Change the color of the current pixel

    image[sr][sc] = newColor;

    // Recursive calls for the neighboring pixels

    dfs(image, imageSize, imageColSize, sr - 1, sc, originalColor, newColor); // Up

    dfs(image, imageSize, imageColSize, sr + 1, sc, originalColor, newColor); // Down

    dfs(image, imageSize, imageColSize, sr, sc - 1, originalColor, newColor); // Left

    dfs(image, imageSize, imageColSize, sr, sc + 1, originalColor, newColor); // Right

}

int\*\* floodFill(int\*\* image, int imageSize, int\* imageColSize, int sr, int sc, int color, int\* returnSize, int\*\* returnColumnSizes) {

    if (image == NULL || imageSize == 0 || imageColSize == NULL) {

        \*returnSize = 0;

        return NULL;

    }

    int originalColor = image[sr][sc];

    // Check if the fill color is the same as the original color

    if (originalColor == color) {

        \*returnSize = imageSize;

        \*returnColumnSizes = imageColSize;

        return image;

    }

    // Allocate memory for the result image

    int\*\* result = (int\*\*)malloc(imageSize \* sizeof(int\*));

    \*returnColumnSizes = (int\*)malloc(imageSize \* sizeof(int));

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

        result[i] = (int\*)malloc(imageColSize[i] \* sizeof(int));

        (\*returnColumnSizes)[i] = imageColSize[i];

    }

    // Copy the original image to the result image

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

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

            result[i][j] = image[i][j];

        }

    }

    // Apply the flood fill algorithm

    dfs(result, imageSize, \*returnColumnSizes, sr, sc, originalColor, color);

    \*returnSize = imageSize;

    return result;

}

int main() {

    // Example image

    int imageSize = 3;

    int imageColSize[] = {3, 3, 3};

    int\*\* image = (int\*\*)malloc(imageSize \* sizeof(int\*));

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

        image[i] = (int\*)malloc(imageColSize[i] \* sizeof(int));

    }

    // Initialize the example image

    int exampleImage[3][3] = {

        {1, 1, 1},

        {1, 1, 0},

        {1, 0, 1}

    };

    // Copy the example image to the allocated memory

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

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

            image[i][j] = exampleImage[i][j];

        }

    }

    // Print the original image

    printf("Original Image:\n");

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

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

            printf("%d ", image[i][j]);

        }

        printf("\n");

    }

    printf("\n");

    // Apply flood fill to change color at position (1,1) to 2

    int returnSize;

    int\* returnColumnSizes;

    int\*\* filledImage = floodFill(image, imageSize, imageColSize, 1, 1, 2, &returnSize, &returnColumnSizes);

    // Print the filled image

    printf("Filled Image:\n");

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

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

            printf("%d ", filledImage[i][j]);

        }

        printf("\n");

    }

    // Free memory

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

        free(image[i]);

        free(filledImage[i]);

    }

    free(image);

    free(filledImage);

    free(returnColumnSizes);

    return 0;

}

**Example 1:**

A diagram of a number

Description automatically generated

**Input:** image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

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

**Explanation:** From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) 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.

## Maximum SubArray

Given an integer array nums, find the

subarray

 with the largest sum, and return *its sum*.

**Example 1:**

**Input:** nums = [-2,1,-3,4,-1,2,1,-5,4]

**Output:** 6

**Explanation:** The subarray [4,-1,2,1] has the largest sum 6.

**Example 2:**

**Input:** nums = [1]

**Output:** 1

**Explanation:** The subarray [1] has the largest sum 1.

**Example 3:**

**Input:** nums = [5,4,-1,7,8]

**Output:** 23

**Explanation:** The subarray [5,4,-1,7,8] has the largest sum 23.

**C :**

int maxSubArray(int\* nums, int numsSize) {

        if (numsSize == 0) {

        // Empty array case

        return 0;

    }

    int maxSum = nums[0];  // Initialize maxSum with the first element

    int currentSum = nums[0];  // Initialize currentSum with the first element

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

        // Update currentSum to either continue the current subarray or start a new one

        currentSum = (currentSum > 0) ? currentSum + nums[i] : nums[i];

        // Update maxSum if the current subarray sum is greater

        if (currentSum > maxSum) {

            maxSum = currentSum;

        }

    }

    return maxSum;

}

int main() {

    // Test cases

    int nums1[] = {-2, 1, -3, 4, -1, 2, 1, -5, 4};

    int size1 = sizeof(nums1) / sizeof(nums1[0]);

    printf("Max subarray sum of nums1: %d\n", maxSubArray(nums1, size1));

    int nums2[] = {1, 2, 3, 4, 5};

    int size2 = sizeof(nums2) / sizeof(nums2[0]);

    printf("Max subarray sum of nums2: %d\n", maxSubArray(nums2, size2));

    int nums3[] = {-1, -2, -3, -4, -5};

    int size3 = sizeof(nums3) / sizeof(nums3[0]);

    printf("Max subarray sum of nums3: %d\n", maxSubArray(nums3, size3));

    return 0;

}

## Sort Colors

Given an array nums 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.

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

You must solve this problem without using the library's sort function.

**Example 1:**

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

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

**Example 2:**

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

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

**C :**

#include <stdio.h>

void sortColors(int\* nums, int numsSize) {

    int low = 0;          // Index for 0

    int high = numsSize - 1;  // Index for 2

    for (int i = 0; i <= high;) {

        if (nums[i] == 0) {

            // Swap current element with the one at the low index

            int temp = nums[i];

            nums[i] = nums[low];

            nums[low] = temp;

            // Move low and i to the right

            low++;

            i++;

        } else if (nums[i] == 2) {

            // Swap current element with the one at the high index

            int temp = nums[i];

            nums[i] = nums[high];

            nums[high] = temp;

            // Move high to the left

            high--;

        } else {

            // Element is 1, move i to the right

            i++;

        }

    }

}

int main() {

    // Test case

    int nums[] = {2, 0, 2, 1, 1, 0};

    int size = sizeof(nums) / sizeof(nums[0]);

    printf("Original array: ");

    for (int i = 0; i < size; i++) {

        printf("%d ", nums[i]);

    }

    printf("\n");

    sortColors(nums, size);

    printf("Sorted array: ");

    for (int i = 0; i < size; i++) {

        printf("%d ", nums[i]);

    }

    printf("\n");

    return 0;

}

## Median Of Two Sorted Arrays

Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

**Example 1:**

**Input:** nums1 = [1,3], nums2 = [2]

**Output:** 2.00000

**Explanation:** merged array = [1,2,3] and median is 2.

**Example 2:**

**Input:** nums1 = [1,2], nums2 = [3,4]

**Output:** 2.50000

**Explanation:** merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

**C :**

#include <stdio.h>

double findMedianSortedArrays(int\* nums1, int nums1Size, int\* nums2, int nums2Size) {

    int totalSize = nums1Size + nums2Size;

    int merged[totalSize];

    // Merge the two sorted arrays

    int i = 0, j = 0, k = 0;

    while (i < nums1Size && j < nums2Size) {

        if (nums1[i] < nums2[j]) {

            merged[k++] = nums1[i++];

        } else {

            merged[k++] = nums2[j++];

        }

    }

    // Add remaining elements from both arrays

    while (i < nums1Size) {

        merged[k++] = nums1[i++];

    }

    while (j < nums2Size) {

        merged[k++] = nums2[j++];

    }

    // Calculate the median

    if (totalSize % 2 == 0) {

        return (double)(merged[totalSize / 2 - 1] + merged[totalSize / 2]) / 2.0;

    } else {

        return (double)merged[totalSize / 2];

    }

}

int main() {

    // Test case

    int nums1[] = {1, 3};

    int nums1Size = 2;

    int nums2[] = {2};

    int nums2Size = 1;

    double median = findMedianSortedArrays(nums1, nums1Size, nums2, nums2Size);

    printf("Median: %lf\n", median);

    return 0;

}

**nput:** nums1 = [1,3], nums2 = [2]

**Output:** 2.00000

**Explanation:** merged array = [1,2,3] and median is 2.

## Root Equals Sum Of Children

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

// Definition for a binary tree node.

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Function to check if a given node is a valid tree node

bool checkTree(struct TreeNode\* root) {

    return (root != NULL && (root->left != NULL) && (root->right != NULL) && (root->left->val + root->right->val == root->val)) ? true : false;

}

// Main function

int main() {

    // Example usage

    struct TreeNode\* root = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->val = 5;

    root->left = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->left->val = 2;

    root->right = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->right->val = 3;

    root->left->left = NULL;

    root->left->right = NULL;

    root->right->left = NULL;

    root->right->right = NULL;

    // Check if the root is a valid tree node

    if (checkTree(root)) {

        printf("Root node is a valid tree node.\n");

    } else {

        printf("Root node is not a valid tree node.\n");

    }

    // Free memory

    free(root->left);

    free(root->right);

    free(root);

    return 0;

}

A diagram of a number

Description automatically generated

**Input:** root = [10,4,6]

**Output:** true

**Explanation:** The values of the root, its left child, and its right child are 10, 4, and 6, respectively.

10 is equal to 4 + 6, so we return true.

Time/Space Complexity: O(1)/O(1)

## Generate Parantheses

Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

**Example 1:**

**Input:** n = 3

**Output:** ["((()))","(()())","(())()","()(())","()()()"]

**Example 2:**

**Input:** n = 1

**Output:** ["()"]

**C :**

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void backtracking(char\*\*\* result, int\* returnSize, char\* curString, int leftCount, int rightCount, int n, int index) {

    if (index == 2 \* n) {

        curString[index] = '\0';

        (\*result)[\*returnSize] = strdup(curString);

        (\*returnSize)++;

        return;

    }

    if (leftCount < n) {

        curString[index] = '(';

        backtracking(result, returnSize, curString, leftCount + 1, rightCount, n, index + 1);

    }

    if (leftCount > rightCount) {

        curString[index] = ')';

        backtracking(result, returnSize, curString, leftCount, rightCount + 1, n, index + 1);

    }

}

char\*\* generateParenthesis(int n, int\* returnSize) {

    char\*\* result = (char\*\*)malloc(1000 \* sizeof(char\*));

    \*returnSize = 0;

    char\* curString = (char\*)malloc((2 \* n + 1) \* sizeof(char));

    curString[0] = '\0';

    backtracking(&result, returnSize, curString, 0, 0, n, 0);

    free(curString);

    return result;

}

int main() {

    int n = 3;

    int returnSize;

    char\*\* result = generateParenthesis(n, &returnSize);

    printf("Generated Parentheses:\n");

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

        printf("%s\n", result[i]);

        free(result[i]);

    }

    free(result);

    return 0;

}

## Meeting Rooms II – Minimum Rooms Required

Given an array of meeting time intervals intervals where intervals[i] = [starti, endi], return *the minimum number of conference rooms required*.

**Example 1:**

**Input:** intervals = [[0,30],[5,10],[15,20]]

**Output:** 2

**Example 2:**

**Input:** intervals = [[7,10],[2,4]]

**Output:** 1

**C :**

#include <stdio.h>

#include <stdlib.h>

int compare(const void\* a, const void\* b) {

    return ((int\*\*)a)[0][0] - ((int\*\*)b)[0][0];

}

int minMeetingRooms(int\*\* intervals, int intervalsSize, int\* intervalsColSize) {

    if (intervals == NULL || intervalsSize == 0) {

        return 0;

    }

    // Sort the intervals based on start times

    qsort(intervals, intervalsSize, sizeof(int\*), compare);

    // Priority queue to store end times of meetings

    int\* endTimes = (int\*)malloc(sizeof(int) \* intervalsSize);

    int rooms = 0;

    // Iterate through the sorted intervals

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

        int start = intervals[i][0];

        int end = intervals[i][1];

        // Check if there is an available room

        int j;

        for (j = 0; j < rooms; j++) {

            if (endTimes[j] <= start) {

                endTimes[j] = end;

                break;

            }

        }

        // If no available room, allocate a new room

        if (j == rooms) {

            endTimes[rooms++] = end;

        }

    }

    free(endTimes);

    return rooms;

}

int main() {

    // Example intervals

    int intervalsSize = 3;

    int intervalsColSize[] = {2, 2, 2};

    int\*\* intervals = (int\*\*)malloc(intervalsSize \* sizeof(int\*));

    intervals[0] = (int\*)malloc(2 \* sizeof(int));

    intervals[0][0] = 0;

    intervals[0][1] = 30;

    intervals[1] = (int\*)malloc(2 \* sizeof(int));

    intervals[1][0] = 5;

    intervals[1][1] = 10;

    intervals[2] = (int\*)malloc(2 \* sizeof(int));

    intervals[2][0] = 15;

    intervals[2][1] = 20;

    int rooms = minMeetingRooms(intervals, intervalsSize, intervalsColSize);

    printf("Minimum meeting rooms required: %d\n", rooms);

    // Free allocated memory

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

        free(intervals[i]);

    }

    free(intervals);

    return 0;

}

## Reverse Linked List

Given the head of a singly linked list, reverse the list, and return *the reversed list*.

**Example 1:**

A diagram of a diagram

Description automatically generated

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

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

**Example 2:**

A diagram of a diagram

Description automatically generated

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

**Output:** [2,1]

**Example 3:**

**Input:** head = []

**Output:** []

**C :**

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     struct ListNode \*next;

 \* };

 \*/

#include <stdio.h>

#include <stdlib.h>

// Definition for singly-linked list.

struct ListNode {

    int val;

    struct ListNode \*next;

};

// Function to reverse a singly-linked list

struct ListNode\* reverseList(struct ListNode\* head) {

    struct ListNode \*prev = NULL;

    struct ListNode \*curr = head;

    struct ListNode \*next = NULL;

    while (curr != NULL) {

        next = curr->next;

        curr->next = prev;

        prev = curr;

        curr = next;

    }

    return prev;

}

// Function to create a new node

struct ListNode\* createNode(int val) {

    struct ListNode\* newNode = (struct ListNode\*)malloc(sizeof(struct ListNode));

    newNode->val = val;

    newNode->next = NULL;

    return newNode;

}

// Function to free memory allocated for the linked list

void freeList(struct ListNode\* head) {

    struct ListNode\* temp;

    while (head != NULL) {

        temp = head;

        head = head->next;

        free(temp);

    }

}

// Function to print the linked list

void printList(struct ListNode\* head) {

    while (head != NULL) {

        printf("%d -> ", head->val);

        head = head->next;

    }

    printf("NULL\n");

}

int main() {

    // Creating a sample linked list: 1 -> 2 -> 3 -> 4 -> 5 -> NULL

    struct ListNode\* head = createNode(1);

    head->next = createNode(2);

    head->next->next = createNode(3);

    head->next->next->next = createNode(4);

    head->next->next->next->next = createNode(5);

    printf("Original list: ");

    printList(head);

    // Reversing the linked list

    head = reverseList(head);

    printf("Reversed list: ");

    printList(head);

    // Freeing the memory allocated for the linked list

    freeList(head);

    return 0;

}

## Implement Queue Using Stack

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

Implement the MyQueue class:

* void push(int x) Pushes element x to the back of the queue.
* int pop() Removes the element from the front of the queue and returns it.
* int peek() Returns the element at the front of the queue.
* boolean empty() Returns true if the queue is empty, false otherwise.

**Notes:**

* You must use **only** standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.
* Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

**Example 1:**

**Input**

["MyQueue", "push", "push", "peek", "pop", "empty"]

[[], [1], [2], [], [], []]

**Output**

[null, null, null, 1, 1, false]

**Explanation**

MyQueue myQueue = new MyQueue();

myQueue.push(1); // queue is: [1]

myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)

myQueue.peek(); // return 1

myQueue.pop(); // return 1, queue is [2]

myQueue.empty(); // return false

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Definition for a MyQueue struct

typedef struct {

    int\* stack1;  // Stack for enqueue operation (push)

    int\* stack2;  // Stack for dequeue operation (pop, peek)

    int top1;     // Top index for stack1

    int top2;     // Top index for stack2

    int capacity; // Capacity of the stacks

} MyQueue;

// Function to create a new MyQueue

MyQueue\* myQueueCreate() {

    MyQueue\* queue = (MyQueue\*)malloc(sizeof(MyQueue));

    queue->capacity = 100; // You can adjust the capacity based on your needs

    queue->stack1 = (int\*)malloc(sizeof(int) \* queue->capacity);

    queue->stack2 = (int\*)malloc(sizeof(int) \* queue->capacity);

    queue->top1 = -1;

    queue->top2 = -1;

    return queue;

}

// Function to enqueue an element into the queue

void enqueue(MyQueue\* obj, int x) {

    if (obj->top1 == obj->capacity - 1) {

        // Resize stack1 if needed

        obj->capacity \*= 2;

        obj->stack1 = realloc(obj->stack1, sizeof(int) \* obj->capacity);

    }

    obj->stack1[++(obj->top1)] = x;

}

// Function to transfer elements from stack1 to stack2

void transferStacks(MyQueue\* obj) {

    while (obj->top1 != -1) {

        obj->stack2[++(obj->top2)] = obj->stack1[(obj->top1)--];

    }

}

// Function to push an element into the queue

void myQueuePush(MyQueue\* obj, int x) {

    enqueue(obj, x);

}

// Function to pop an element from the queue

int myQueuePop(MyQueue\* obj) {

    if (obj->top2 == -1) {

        // If stack2 is empty, transfer elements from stack1

        transferStacks(obj);

    }

    if (obj->top2 == -1) {

        return -1; // Queue is empty

    }

    return obj->stack2[(obj->top2)--];

}

// Function to peek the front element of the queue

int myQueuePeek(MyQueue\* obj) {

    if (obj->top2 == -1) {

        // If stack2 is empty, transfer elements from stack1

        transferStacks(obj);

    }

    if (obj->top2 == -1) {

        return -1; // Queue is empty

    }

    return obj->stack2[obj->top2];

}

// Function to check if the queue is empty

bool myQueueEmpty(MyQueue\* obj) {

    return (obj->top1 == -1) && (obj->top2 == -1);

}

// Function to free memory allocated for the queue

void myQueueFree(MyQueue\* obj) {

    free(obj->stack1);

    free(obj->stack2);

    free(obj);

}

// Main function to test the MyQueue implementation

int main() {

    // Create a new queue

    MyQueue\* obj = myQueueCreate();

    // Push elements into the queue

    myQueuePush(obj, 1);

    myQueuePush(obj, 2);

    myQueuePush(obj, 3);

    // Pop and print elements from the queue

    printf("Dequeued element: %d\n", myQueuePop(obj)); // Output: 1

    printf("Dequeued element: %d\n", myQueuePop(obj)); // Output: 2

    // Peek and print the front element of the queue

    printf("Front element: %d\n", myQueuePeek(obj)); // Output: 3

    // Check if the queue is empty

    printf("Is the queue empty? %s\n", myQueueEmpty(obj) ? "Yes" : "No"); // Output: No

    // Free memory allocated for the queue

    myQueueFree(obj);

    return 0;

}

## Implement Stack Using Queue

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

* void push(int x) Pushes element x to the top of the stack.
* int pop() Removes the element on the top of the stack and returns it.
* int top() Returns the element on the top of the stack.
* boolean empty() Returns true if the stack is empty, false otherwise.

**Notes:**

* You must use **only** standard operations of a queue, which means that only push to back, peek/pop from front, size and is empty operations are valid.
* Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

**Example 1:**

**Input**

["MyStack", "push", "push", "top", "pop", "empty"]

[[], [1], [2], [], [], []]

**Output**

[null, null, null, 2, 2, false]

**Explanation**

MyStack myStack = new MyStack();

myStack.push(1);

myStack.push(2);

myStack.top(); // return 2

myStack.pop(); // return 2

myStack.empty(); // return False

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Definition for MyStack struct

typedef struct {

    int\* queue;

    int front;

    int rear;

} MyStack;

// Function to create a new MyStack

MyStack\* myStackCreate() {

    MyStack\* obj = (MyStack\*)malloc(sizeof(MyStack));

    obj->queue = (int\*)malloc(1000 \* sizeof(int));  // Adjust the size according to your requirements

    obj->front = obj->rear = -1;

    return obj;

}

// Function to enqueue an element into the queue

void enqueue(int\* queue, int\* rear, int x) {

    queue[++(\*rear)] = x;

}

// Function to dequeue an element from the queue

int dequeue(int\* queue, int\* front) {

    return queue[(\*front)++];

}

// Function to push an element onto the stack

void myStackPush(MyStack\* obj, int x) {

    enqueue(obj->queue, &(obj->rear), x);

    // Rotate the queue to make the recently pushed element at the front

    int size = obj->rear - obj->front + 1;

    for (int i = 0; i < size - 1; ++i) {

        enqueue(obj->queue, &(obj->rear), dequeue(obj->queue, &(obj->front)));

    }

}

// Function to pop the top element from the stack

int myStackPop(MyStack\* obj) {

    if (obj == NULL || obj->front > obj->rear) {

        // Stack is empty

        return -1;

    }

    return dequeue(obj->queue, &(obj->front));

}

// Function to get the top element of the stack without popping it

int myStackTop(MyStack\* obj) {

    if (obj == NULL || obj->front > obj->rear) {

        // Stack is empty

        return -1;

    }

    return obj->queue[obj->front];

}

// Function to check if the stack is empty

bool myStackEmpty(MyStack\* obj) {

    return obj == NULL || obj->front > obj->rear;

}

// Function to free memory allocated for the stack

void myStackFree(MyStack\* obj) {

    free(obj->queue);

    free(obj);

}

// Main function to test the MyStack implementation

int main() {

    // Create a new stack

    MyStack\* obj = myStackCreate();

    // Push elements onto the stack

    myStackPush(obj, 1);

    myStackPush(obj, 2);

    myStackPush(obj, 3);

    // Pop and print elements from the stack

    printf("Popped element: %d\n", myStackPop(obj)); // Output: 3

    printf("Popped element: %d\n", myStackPop(obj)); // Output: 2

    // Get the top element of the stack without popping it

    printf("Top element: %d\n", myStackTop(obj)); // Output: 1

    // Check if the stack is empty

    printf("Is the stack empty? %s\n", myStackEmpty(obj) ? "Yes" : "No"); // Output: No

    // Free memory allocated for the stack

    myStackFree(obj);

    return 0;

}

/\*\*

 \* Your MyStack struct will be instantiated and called as such:

 \* MyStack\* obj = myStackCreate();

 \* myStackPush(obj, x);

 \* int param\_2 = myStackPop(obj);

 \* int param\_3 = myStackTop(obj);

 \* bool param\_4 = myStackEmpty(obj);

 \* myStackFree(obj);

\*/

## Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

**Example 1:**

A black and blue logo

Description automatically generated

**Input:** height = [0,1,0,2,1,0,1,3,2,1,2,1]

**Output:** 6

**Explanation:** The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

**Example 2:**

**Input:** height = [4,2,0,3,2,5]

**Output:** 9

**C :**

#include <stdio.h>

int trap(int\* height, int heightSize) {

    int left = 0, right = heightSize - 1;

    int ans = 0;

    int leftMax = 0, rightMax = 0;

    while (left < right) {

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

            if (height[left] >= leftMax) {

                leftMax = height[left];

            } else {

                ans += (leftMax - height[left]);

            }

            left++;

        } else {

            if (height[right] >= rightMax) {

                rightMax = height[right];

            } else {

                ans += (rightMax - height[right]);

            }

            right--;

        }

    }

    return ans;

}

int main() {

    // Example heights

    int height[] = {0,1,0,2,1,0,1,3,2,1,2,1};

    int heightSize = sizeof(height) / sizeof(height[0]);

    // Calculate the trapped water

    int trappedWater = trap(height, heightSize);

    // Print the result

    printf("Trapped water: %d\n", trappedWater);

    return 0;

}

## Longest SubString Without Repeating Characters

Given a string s, find the length of the **longest**

**substring**

 without repeating characters.

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

**Example 2:**

**Input:** s = "bbbbb"

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

**Example 3:**

**Input:** s = "pwwkew"

**Output:** 3

**Explanation:** The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int lengthOfLongestSubstring(char\* s) {

    int n = strlen(s);

    if (n == 0) {

        return 0;

    }

    int\* charIndex = (int\*)malloc(sizeof(int) \* 128);

    for (int i = 0; i < 128; i++) {

        charIndex[i] = -1;

    }

    int maxLength = 0;

    int start = 0;

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

        if (charIndex[s[end]] != -1) {

            start = fmax(charIndex[s[end]] + 1, start);

        }

        maxLength = fmax(maxLength, end - start + 1);

        charIndex[s[end]] = end;

    }

    free(charIndex);

    return maxLength;

}

int main() {

    // Test string

    char\* s = "abcabcbb";

    // Calculate the length of the longest substring without repeating characters

    int length = lengthOfLongestSubstring(s);

    // Print the result

    printf("Length of the longest substring without repeating characters: %d\n", length);

    return 0;

}

class Solution {

    public int lengthOfLongestSubstring(String s) {

        if (s.length() == 0) return 0;

        else if (s.length() == 1) return 1;

        else {

            HashSet <Character> hs = new HashSet<Character>();

            int max\_length = 0;

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

                hs.add(s.charAt(k));

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

                    if (!hs.contains(s.charAt(i))) {

                        hs.add(s.charAt(i));

                        if (i == s.length()-1) {

                            max\_length = Math.max(max\_length,hs.size());

                            hs.clear();

                        }

                    }

                    else {

                        max\_length = Math.max(max\_length,hs.size());

                        hs.clear();

                        break;

                    }

                }

            }

            return max\_length;

        }

    }

}

## SubArray Sum Equals K

Given an array of integers nums and an integer k, return *the total number of subarrays whose sum equals to* k.

A subarray is a contiguous **non-empty** sequence of elements within an array.

**Example 1:**

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

**Output:** 2

**Example 2:**

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

**Output:** 2

**C :**

#include <stdio.h>

#include <stdlib.h>

int subarraySum(int\* nums, int numsSize, int k) {

    int count = 0;

    int sum = 0;

    // Create a hashmap to store cumulative sum frequencies

    int\* hashmap = (int\*)malloc(sizeof(int) \* (numsSize + 1));

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

        hashmap[i] = 0;

    }

    hashmap[0] = 1;

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

        sum += nums[i];

        int diff = sum - k;

        if (hashmap[diff] > 0) {

            count += hashmap[diff];

        }

        hashmap[sum]++;

    }

    free(hashmap);

    // Debugging output

    printf("Count: %d\n", count);

    return count;

}

int main() {

    // Test array

    int nums[] = {1, 1, 1};

    // Size of the array

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Target sum

    int k = 2;

    // Calculate the number of subarrays with the sum equal to k

    int count = subarraySum(nums, numsSize, k);

    // Print the result

    printf("Number of subarrays with sum equal to %d: %d\n", k, count);

    return 0;

}

## Search In A Rotated Sorted Array

There is an integer array nums sorted in ascending order (with **distinct** values).

Prior to being passed to your function, nums is **possibly rotated** at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums **after** the possible rotation and an integer target, return *the index of*target*if it is in*nums*, or*-1*if it is not in*nums.

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [4,5,6,7,0,1,2], target = 0

**Output:** 4

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2], target = 3

**Output:** -1

**Example 3:**

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

**Output:** -1

**C :**

#include <stdio.h>

int search(int\* nums, int numsSize, int target) {

    int n = numsSize;

    int left = 0, right = n - 1;

    while (left <= right) {

        int mid = left + (right - left) / 2;

        // Case 1: find target

        if (nums[mid] == target) {

            return mid;

        }

        // Case 2: subarray on mid's left is sorted

        else if (nums[mid] >= nums[left]) {

            if (target >= nums[left] && target < nums[mid]) {

                right = mid - 1;

            } else {

                left = mid + 1;

            }

        }

        // Case 3: subarray on mid's right is sorted

        else {

            if (target <= nums[right] && target > nums[mid]) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

    }

    return -1; // Target not found

}

int main() {

    // Test array

    int nums[] = {4, 5, 6, 7, 0, 1, 2};

    // Size of the array

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Target element to search

    int target = 0;

    // Search for the target element in the rotated sorted array

    int index = search(nums, numsSize, target);

    // Print the result

    if (index != -1) {

        printf("Target element %d found at index %d.\n", target, index);

    } else {

        printf("Target element %d not found in the array.\n", target);

    }

    return 0;

}

## Top K elements

#include <stdio.h>

#include <stdlib.h>

// Function prototypes

void swap(int\* a, int\* b);

int partition(int arr[], int low, int high);

void quickSort(int arr[], int low, int high);

void printTopK(int arr[], int n, int k);

// Function to swap two elements

void swap(int\* a, int\* b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

// Function to perform partitioning

int partition(int arr[], int low, int high) {

    int pivot = arr[high]; // Pivot element

    int i = low - 1; // Index of smaller element

    for (int j = low; j < high; j++) {

        // If current element is smaller than or equal to pivot

        if (arr[j] <= pivot) {

            i++; // Increment index of smaller element

            swap(&arr[i], &arr[j]); // Swap current element with arr[i]

        }

    }

    swap(&arr[i + 1], &arr[high]); // Swap arr[i+1] with arr[high] (or pivot)

    return i + 1; // Return the partitioning index

}

// Function to perform quick sort

void quickSort(int arr[], int low, int high) {

    if (low < high) {

        // Partitioning index

        int pi = partition(arr, low, high);

        // Recursively sort elements before partition and after partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

// Function to print top k elements

void printTopK(int arr[], int n, int k) {

    if (k > n) {

        printf("k is greater than the array size\n");

        return;

    }

    // Sort the array in non-increasing order

    quickSort(arr, 0, n - 1);

    // Print the top k elements

    printf("Top %d elements: ", k);

    for (int i = 0; i < k; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main() {

    int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}; // Sample array

    int n = sizeof(arr) / sizeof(arr[0]); // Size of the array

    int k = 3; // Number of top elements to find

    printTopK(arr, n, k);

    return 0;

}

Or

#include <stdio.h>

#include <stdlib.h>

// Function prototypes

void minHeapify(int arr[], int n, int i);

void buildMinHeap(int arr[], int n);

void heapSort(int arr[], int n);

void printTopK(int arr[], int n, int k);

// Function to heapify a subtree rooted with node i which is an index in arr[]

void minHeapify(int arr[], int n, int i) {

    int smallest = i; // Initialize smallest as root

    int left = 2 \* i + 1; // left = 2\*i + 1

    int right = 2 \* i + 2; // right = 2\*i + 2

    // If left child is smaller than root

    if (left < n && arr[left] < arr[smallest])

        smallest = left;

    // If right child is smaller than smallest so far

    if (right < n && arr[right] < arr[smallest])

        smallest = right;

    // If smallest is not root

    if (smallest != i) {

        int temp = arr[i];

        arr[i] = arr[smallest];

        arr[smallest] = temp;

        // Recursively heapify the affected sub-tree

        minHeapify(arr, n, smallest);

    }

}

// Function to build a min heap

void buildMinHeap(int arr[], int n) {

    // Index of last non-leaf node

    int startIdx = (n / 2) - 1;

    // Perform reverse level order traversal from last non-leaf node and heapify each node

    for (int i = startIdx; i >= 0; i--) {

        minHeapify(arr, n, i);

    }

}

// Function to perform heap sort

void heapSort(int arr[], int n) {

    // Build heap (rearrange array)

    buildMinHeap(arr, n);

    // Extract one by one from heap

    for (int i = n - 1; i > 0; i--) {

        // Move current root to end

        int temp = arr[0];

        arr[0] = arr[i];

        arr[i] = temp;

        // Call min heapify on the reduced heap

        minHeapify(arr, i, 0);

    }

}

// Function to print top k elements

void printTopK(int arr[], int n, int k) {

    if (k > n) {

        printf("k is greater than the array size\n");

        return;

    }

    // Sort the array using heap sort

    heapSort(arr, n);

    // Print top k elements

    printf("Top %d elements: ", k);

    for (int i = n - 1; i >= n - k; i--) {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main() {

    int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}; // Sample array

    int n = sizeof(arr) / sizeof(arr[0]); // Size of the array

    int k = 3; // Number of top elements to find

    printTopK(arr, n, k);

    return 0;

}

## Top K Frequent Elements

Given an integer array nums and an integer k, return *the* k *most frequent elements*. You may return the answer in **any order**.

**Example 1:**

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

**Output:** [1,2]

**Example 2:**

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

**Output:** [1]

**C :**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

    int value;

    int frequency;

} Element;

int compare(const void \*a, const void \*b) {

    return ((Element \*)b)->frequency - ((Element \*)a)->frequency;

}

int\* topKFrequent(int\* nums, int numsSize, int k, int\* returnSize) {

    // Initialize a hash table with calloc

    int\* hashTable = (int\*)calloc(1001, sizeof(int));

    // Count the frequency of each element and store it in the hash table

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

        hashTable[nums[i]]++;

    }

    // Populate the elements array with values and frequencies from the hash table

    Element\* elements = (Element\*)malloc(1001 \* sizeof(Element));

    int count = 0;

    for (int i = 0; i <= 1000; ++i) {

        if (hashTable[i] > 0) {

            elements[count++] = (Element){i, hashTable[i]};

        }

    }

    // Sort the elements array based on frequency in descending order

    qsort(elements, count, sizeof(Element), compare);

    // Allocate memory for the result array

    int\* result = (int\*)malloc(k \* sizeof(int));

    // Copy the top k frequent elements to the result array

    for (int i = 0; i < k; ++i) {

        result[i] = elements[i].value;

    }

    // Update the returnSize

    \*returnSize = k;

    // Free allocated memory

    free(elements);

    free(hashTable);

    return result;

}

int main() {

    // Test array

    int nums[] = {1, 1, 1, 2, 2, 3};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int k = 2; // Number of top frequent elements to find

    // Get the top k frequent elements

    int returnSize;

    int\* topKElements = topKFrequent(nums, numsSize, k, &returnSize);

    // Print the top k frequent elements

    printf("Top %d frequent elements: ", k);

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

        printf("%d ", topKElements[i]);

    }

    printf("\n");

    // Free allocated memory

    free(topKElements);

    return 0;

}

## Rotate Image

You are given an n x n 2D matrix representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm), which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

**Example 1:**

A black arrow pointing to a white rectangular object

Description automatically generated

**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [[7,4,1],[8,5,2],[9,6,3]]

**Example 2:**

A number with a arrow pointing to the right

Description automatically generated with medium confidence

**Input:** matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

**Output:** [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

**C :**

#include <stdio.h>

#include <stdlib.h>

void transpose(int\*\* matrix, int matrixSize, int matrixColSize) {

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

        for (int j = i + 1; j < matrixColSize; j++) {

            int temp = matrix[i][j];

            matrix[i][j] = matrix[j][i];

            matrix[j][i] = temp;

        }

    }

}

void reverse(int\*\* matrix, int matrixSize, int matrixColSize) {

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

        for (int j = 0; j < matrixColSize / 2; j++) {

            int temp = matrix[i][j];

            matrix[i][j] = matrix[i][matrixColSize - j - 1];

            matrix[i][matrixColSize - j - 1] = temp;

        }

    }

}

void rotate(int\*\* matrix, int matrixSize, int\* matrixColSize) {

    transpose(matrix, matrixSize, \*matrixColSize);

    reverse(matrix, matrixSize, \*matrixColSize);

}

int main() {

    // Example matrix

    int matrixSize = 3;

    int matrixColSize = 3;

    int\*\* matrix = (int\*\*)malloc(matrixSize \* sizeof(int\*));

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

        matrix[i] = (int\*)malloc(matrixColSize \* sizeof(int));

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

            matrix[i][j] = i \* matrixColSize + j + 1;

        }

    }

    // Print the original matrix

    printf("Original Matrix:\n");

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

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

            printf("%d ", matrix[i][j]);

        }

        printf("\n");

    }

    // Rotate the matrix

    rotate(matrix, matrixSize, &matrixColSize);

    // Print the rotated matrix

    printf("\nRotated Matrix:\n");

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

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

            printf("%d ", matrix[i][j]);

        }

        printf("\n");

    }

    // Free allocated memory

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

        free(matrix[i]);

    }

    free(matrix);

    return 0;

}

## Gas Station

There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next (i + 1)th station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* -1. If there exists a solution, it is **guaranteed** to be **unique**

**Example 1:**

**Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]

**Output:** 3

**Explanation:**

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.

Therefore, return 3 as the starting index.

**Example 2:**

**Input:** gas = [2,3,4], cost = [3,4,3]

**Output:** -1

**Explanation:**

You can't start at station 0 or 1, as there is not enough gas to travel to the next station.

Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 0. Your tank = 4 - 3 + 2 = 3

Travel to station 1. Your tank = 3 - 3 + 3 = 3

You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.

Therefore, you can't travel around the circuit once no matter where you start.

**C :**

#include <stdio.h>

int canCompleteCircuit(int\* gas, int gasSize, int\* cost, int costSize) {

    int totalGas = 0;

    int totalCost = 0;

    int currentGas = 0;

    int startStation = 0;

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

        totalGas += gas[i];

        totalCost += cost[i];

        currentGas += gas[i] - cost[i];

        // If the current sum becomes negative, reset the starting station to the next station

        if (currentGas < 0) {

            currentGas = 0;

            startStation = i + 1;

        }

    }

    // If the total gas is less than the total cost, there is no solution

    if (totalGas < totalCost) {

        return -1;

    }

    return startStation;

}

int main() {

    // Example gas and cost arrays

    int gas[] = {1, 2, 3, 4, 5};

    int gasSize = sizeof(gas) / sizeof(gas[0]);

    int cost[] = {3, 4, 5, 1, 2};

    int costSize = sizeof(cost) / sizeof(cost[0]);

    // Calculate the starting station

    int startStation = canCompleteCircuit(gas, gasSize, cost, costSize);

    // Print the result

    printf("Starting station: %d\n", startStation);

    return 0;

}

**Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]

**Output:** 3

**Explanation:**

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.

Therefore, return 3 as the starting index.

## Delete Node in BST

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return *the****root node reference****(possibly updated) of the BST*.

Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

**Example 1:**

A diagram of a diagram

Description automatically generated

**Input:** root = [5,3,6,2,4,null,7], key = 3

**Output:** [5,4,6,2,null,null,7]

**Explanation:** Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the above BST.

Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.

A diagram of a network

Description automatically generated

**Example 2:**

**Input:** root = [5,3,6,2,4,null,7], key = 0

**Output:** [5,3,6,2,4,null,7]

**Explanation:** The tree does not contain a node with value = 0.

**Example 3:**

**Input:** root = [], key = 0

**Output:** []

**Constraints:**

**C :**

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

#include <stdio.h>

#include <stdlib.h>

// Definition for a binary tree node.

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Function to find the minimum node in a BST

struct TreeNode\* findMin(struct TreeNode\* node) {

    while (node->left != NULL) {

        node = node->left;

    }

    return node;

}

// Function to delete a node with a given key from a BST

struct TreeNode\* deleteNode(struct TreeNode\* root, int key) {

    if (root == NULL) {

        return NULL;

    }

    if (key < root->val) {

        root->left = deleteNode(root->left, key);

    } else if (key > root->val) {

        root->right = deleteNode(root->right, key);

    } else {

        if (root->left == NULL) {

            struct TreeNode\* temp = root->right;

            free(root);

            return temp;

        } else if (root->right == NULL) {

            struct TreeNode\* temp = root->left;

            free(root);

            return temp;

        }

        struct TreeNode\* temp = findMin(root->right);

        root->val = temp->val;

        root->right = deleteNode(root->right, temp->val);

    }

    return root;

}

// Function to create a new TreeNode

struct TreeNode\* createNode(int val) {

    struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    newNode->val = val;

    newNode->left = newNode->right = NULL;

    return newNode;

}

// Function to print the inorder traversal of the tree

void inorderTraversal(struct TreeNode\* root) {

    if (root != NULL) {

        inorderTraversal(root->left);

        printf("%d ", root->val);

        inorderTraversal(root->right);

    }

}

// Main function to test the deleteNode function

int main() {

    // Creating a sample BST

    struct TreeNode\* root = createNode(5);

    root->left = createNode(3);

    root->right = createNode(6);

    root->left->left = createNode(2);

    root->left->right = createNode(4);

    root->right->right = createNode(7);

    printf("Original BST: ");

    inorderTraversal(root);

    printf("\n");

    // Deleting node with key 3

    root = deleteNode(root, 3);

    printf("BST after deleting node with key 3: ");

    inorderTraversal(root);

    printf("\n");

    // Freeing memory

    // You should implement a function to free the memory of the tree nodes

    // Here, we're just printing the tree after deletion to show the result

    return 0;

}

## Word Search

Given an m x n grid of characters board and a string word, return true *if* word *exists in the grid*.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

**Example 1:**

A white and orange squares with letters

Description automatically generated

**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"

**Output:** true

**Example 2:**

A white and orange squares with black letters

Description automatically generated

**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

**Output:** true

**Example 3:**

A grid of letters in black

Description automatically generated

**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"

**Output:** false

**C :**

bool backtrack(char\*\* board, int row, int col, int boardSize, int\* boardColSize, char\* word, int index) {

    if (index == strlen(word)) {

        // All characters of the word are found

        return true;

    }

    if (row < 0 || col < 0 || row >= boardSize || col >= boardColSize[row] || board[row][col] != word[index]) {

        // Out of bounds or the current cell does not match the current character of the word

        return false;

    }

    // Mark the current cell as visited

    char original = board[row][col];

    board[row][col] = '\0';

    // Explore neighbors in all four directions

    bool found = (backtrack(board, row - 1, col, boardSize, boardColSize, word, index + 1) ||

                  backtrack(board, row + 1, col, boardSize, boardColSize, word, index + 1) ||

                  backtrack(board, row, col - 1, boardSize, boardColSize, word, index + 1) ||

                  backtrack(board, row, col + 1, boardSize, boardColSize, word, index + 1));

    // Restore the original value of the current cell

    board[row][col] = original;

    return found;

}

bool exist(char\*\* board, int boardSize, int\* boardColSize, char\* word) {

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

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

            if (backtrack(board, i, j, boardSize, boardColSize, word, 0)) {

                // If the word is found starting from (i, j)

                return true;

            }

        }

    }

    return false;

}

int main() {

    char\* board[][4] = {{"A","B","C","E"},{"S","F","C","S"},{"A","D","E","E"}};

    int boardSize = sizeof(board) / sizeof(board[0]);

    int boardColSize[] = {4, 4, 4};

    char\* word = "ABCCED";

    // Check if the word exists on the board

    bool result = exist((char\*\*)board, boardSize, boardColSize, word);

    // Print the result

    if (result) {

        printf("The word '%s' exists on the board.\n", word);

    } else {

        printf("The word '%s' does not exist on the board.\n", word);

    }

    return 0;

}

## Integer To Roman

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given an integer, convert it to a roman numeral.

**Example 1:**

**Input:** num = 3

**Output:** "III"

**Explanation:** 3 is represented as 3 ones.

**Example 2:**

**Input:** num = 58

**Output:** "LVIII"

**Explanation:** L = 50, V = 5, III = 3.

**Example 3:**

**Input:** num = 1994

**Output:** "MCMXCIV"

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

**C :**

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

bool backtrack(char\*\* board, int row, int col, int boardSize, int\* boardColSize, char\* word, int index) {

    if (index == strlen(word)) {

        // All characters of the word are found

        return true;

    }

    if (row < 0 || col < 0 || row >= boardSize || col >= boardColSize[row] || board[row][col] != word[index]) {

        // Out of bounds or the current cell does not match the current character of the word

        return false;

    }

    // Mark the current cell as visited

    char original = board[row][col];

    board[row][col] = '\0';  // <-- This line causes the segmentation fault

    // Explore neighbors in all four directions

    bool found = (backtrack(board, row - 1, col, boardSize, boardColSize, word, index + 1) ||

                  backtrack(board, row + 1, col, boardSize, boardColSize, word, index + 1) ||

                  backtrack(board, row, col - 1, boardSize, boardColSize, word, index + 1) ||

                  backtrack(board, row, col + 1, boardSize, boardColSize, word, index + 1));

    // Restore the original value of the current cell

    board[row][col] = original;

    return found;

}

bool exist(char\*\* board, int boardSize, int\* boardColSize, char\* word) {

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

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

            if (backtrack(board, i, j, boardSize, boardColSize, word, 0)) {

                // If the word is found starting from (i, j)

                return true;

            }

        }

    }

    return false;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* intToRoman(int num) {

    const char\* thousands[] = {"", "M", "MM", "MMM"};

    const char\* hundreds[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};

    const char\* tens[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};

    const char\* ones[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};

    char \*result = (char \*)malloc(sizeof(char) \* 16);

    result[0] = '\0';

    strcat(result, thousands[(num / 1000)]);

    strcat(result, hundreds[(num % 1000) / 100]);

    strcat(result, tens[(num % 100) / 10]);

    strcat(result, ones[(num % 10)]);

    return result;

}

int main() {

    int num = 3549;

    char \*roman = intToRoman(num);

    printf("Roman numeral representation of %d is: %s\n", num, roman);

    free(roman); // Don't forget to free the allocated memory

    return 0;

}

## Merge String Alternately

You are given two strings word1 and word2. Merge the strings by adding letters in alternating order, starting with word1. If a string is longer than the other, append the additional letters onto the end of the merged string.

Return *the merged string.*

**Example 1:**

**Input:** word1 = "abc", word2 = "pqr"

**Output:** "apbqcr"

**Explanation:** The merged string will be merged as so:

word1: a b c

word2: p q r

merged: a p b q c r

**Example 2:**

**Input:** word1 = "ab", word2 = "pqrs"

**Output:** "apbqrs"

**Explanation:** Notice that as word2 is longer, "rs" is appended to the end.

word1: a b

word2: p q r s

merged: a p b q r s

**Example 3:**

**Input:** word1 = "abcd", word2 = "pq"

**Output:** "apbqcd"

**Explanation:** Notice that as word1 is longer, "cd" is appended to the end.

word1: a b c d

word2: p q

merged: a p b q c d

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* mergeAlternately(char\* word1, char\* word2) {

    int i = 0;

    int j = 0;

    int k = 0;

    int len1 = strlen(word1);

    int len2 = strlen(word2);

    int len = len1 + len2;

    char\* result = (char\*)malloc(sizeof(char) \* (len + 1));

    memset(result, 0, sizeof(char) \* (len + 1));

    while (i < len1 || j < len2) {

        if (i < len1) result[k++] = word1[i++];

        if (j < len2) result[k++] = word2[j++];

    }

    return result;

}

int main() {

    char\* word1 = "abc";

    char\* word2 = "defgh";

    char\* merged = mergeAlternately(word1, word2);

    printf("Merged string: %s\n", merged);

    free(merged); // Don't forget to free the allocated memory

    return 0;

}

## Reverse Linked List II (In between from one position to another)

Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return *the reversed list*.

**Example 1:**

A diagram of a diagram

Description automatically generated

**Input:** head = [1,2,3,4,5], left = 2, right = 4

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

**Example 2:**

**Input:** head = [5], left = 1, right = 1

**Output:** [5]

**C :**

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     struct ListNode \*next;

 \* };

 \*/

#include <stdio.h>

#include <stdlib.h>

struct ListNode {

    int val;

    struct ListNode \*next;

};

struct ListNode\* reverseBetween(struct ListNode\* head, int m, int n) {

    if (head == NULL) {

        return NULL;

    }

    struct ListNode\* dummy = (struct ListNode\*)malloc(sizeof(struct ListNode));

    dummy->next = head;

    struct ListNode\* pre = dummy;

    // Move pre to the node before the sublist to be reversed

    for (int i = 0; i < m - 1; i++) {

        pre = pre->next;

    }

    struct ListNode\* curr = pre->next; // Start of the sublist to be reversed

    struct ListNode\* next = curr->next; // Node that will be reversed

    // Reverse the sublist from m to n

    for (int i = 0; i < n - m; i++) {

        curr->next = next->next;

        next->next = pre->next;

        pre->next = next;

        next = curr->next;

    }

    // The sublist has been reversed, and we return the modified list

    return dummy->next;

}

// Function to create a new node with a given value

struct ListNode\* createNode(int val) {

    struct ListNode\* newNode = (struct ListNode\*)malloc(sizeof(struct ListNode));

    newNode->val = val;

    newNode->next = NULL;

    return newNode;

}

// Function to print the linked list

void printList(struct ListNode\* head) {

    while (head != NULL) {

        printf("%d ", head->val);

        head = head->next;

    }

    printf("\n");

}

int main() {

    // Create a sample linked list

    struct ListNode\* head = createNode(1);

    head->next = createNode(2);

    head->next->next = createNode(3);

    head->next->next->next = createNode(4);

    head->next->next->next->next = createNode(5);

    printf("Original list: ");

    printList(head);

    int m = 2, n = 4;

    struct ListNode\* reversed = reverseBetween(head, m, n);

    printf("Reversed list from %d to %d: ", m, n);

    printList(reversed);

    // Free the memory allocated for the linked lists

    while (head != NULL) {

        struct ListNode\* temp = head;

        head = head->next;

        free(temp);

    }

    return 0;

}

## Implement Trie – Prefix Tree

A [**trie**](https://en.wikipedia.org/wiki/Trie) (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

* Trie() Initializes the trie object.
* void insert(String word) Inserts the string word into the trie.
* boolean search(String word) Returns true if the string word is in the trie (i.e., was inserted before), and false otherwise.
* boolean startsWith(String prefix) Returns true if there is a previously inserted string word that has the prefix prefix, and false otherwise.

**Example 1:**

**Input**

["Trie", "insert", "search", "search", "startsWith", "insert", "search"]

[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]

**Output**

[null, null, true, false, true, null, true]

**Explanation**

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // return True

trie.search("app"); // return False

trie.startsWith("app"); // return True

trie.insert("app");

trie.search("app"); // return True

**C :**

// Define TrieNode structure

typedef struct TrieNode {

    struct TrieNode\* children[26];

    bool isEndOfWord;

} TrieNode;

// Define Trie structure

typedef struct {

    TrieNode\* root;

} Trie;

// Function to create a new TrieNode

TrieNode\* createNode() {

    TrieNode\* newNode = (TrieNode\*)malloc(sizeof(TrieNode));

    for (int i = 0; i < 26; i++) {

        newNode->children[i] = NULL;

    }

    newNode->isEndOfWord = false;

    return newNode;

}

// Initialize the Trie

Trie\* trieCreate() {

    Trie\* trie = (Trie\*)malloc(sizeof(Trie));

    trie->root = createNode();

    return trie;

}

// Insert a word into the Trie

void trieInsert(Trie\* obj, char\* word) {

    TrieNode\* node = obj->root;

    while (\*word) {

        int index = \*word - 'a';

        if (node->children[index] == NULL) {

            node->children[index] = createNode();

        }

        node = node->children[index];

        word++;

    }

    node->isEndOfWord = true;

}

// Search for a word in the Trie

bool trieSearch(Trie\* obj, char\* word) {

    TrieNode\* node = obj->root;

    while (\*word) {

        int index = \*word - 'a';

        if (node->children[index] == NULL) {

            return false;

        }

        node = node->children[index];

        word++;

    }

    return node->isEndOfWord;

}

// Check if a prefix exists in the Trie

bool trieStartsWith(Trie\* obj, char\* prefix) {

    TrieNode\* node = obj->root;

    while (\*prefix) {

        int index = \*prefix - 'a';

        if (node->children[index] == NULL) {

            return false;

        }

        node = node->children[index];

        prefix++;

    }

    return true;

}

// Free the Trie memory

void trieFree(Trie\* obj) {

    // Use recursive helper function to free Trie nodes

    void freeNode(TrieNode\* node) {

        if (node == NULL) {

            return;

        }

        for (int i = 0; i < 26; i++) {

            freeNode(node->children[i]);

        }

        free(node);

    }

    freeNode(obj->root);

    free(obj);

}

/\*\*

 \* Your Trie struct will be instantiated and called as such:

 \* Trie\* obj = trieCreate();

 \* trieInsert(obj, word);

 \* bool param\_2 = trieSearch(obj, word);

 \* bool param\_3 = trieStartsWith(obj, prefix);

 \* trieFree(obj);

\*/

## Odd Even Linked List

Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in O(1) extra space complexity and O(n) time complexity.

**Example 1:**

A diagram of a diagram

Description automatically generated

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

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

**Example 2:**

A diagram of a number

Description automatically generated

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

**Output:** [2,3,6,7,1,5,4]

**C :**

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     struct ListNode \*next;

 \* };

 \*/

#include <stdio.h>

#include <stdlib.h>

struct ListNode {

    int val;

    struct ListNode \*next;

};

struct ListNode\* oddEvenList(struct ListNode\* head) {

    if (head == NULL || head->next == NULL) {

        return head; // No or only one node, no reordering needed

    }

    struct ListNode\* oddHead = head;    // Head of the odd indices list

    struct ListNode\* evenHead = head->next; // Head of the even indices list

    struct ListNode\* odd = oddHead;     // Pointer for iterating over odd indices

    struct ListNode\* even = evenHead;   // Pointer for iterating over even indices

    while (even != NULL && even->next != NULL) {

        // Connect odd indices

        odd->next = even->next;

        odd = odd->next;

        // Connect even indices

        even->next = odd->next;

        even = even->next;

    }

    // Combine the odd and even lists

    odd->next = evenHead;

    return oddHead;

}

// Function to create a new node with a given value

struct ListNode\* createNode(int val) {

    struct ListNode\* newNode = (struct ListNode\*)malloc(sizeof(struct ListNode));

    newNode->val = val;

    newNode->next = NULL;

    return newNode;

}

// Function to print the linked list

void printList(struct ListNode\* head) {

    while (head != NULL) {

        printf("%d ", head->val);

        head = head->next;

    }

    printf("\n");

}

int main() {

    // Create a sample linked list

    struct ListNode\* head = createNode(1);

    head->next = createNode(2);

    head->next->next = createNode(3);

    head->next->next->next = createNode(4);

    head->next->next->next->next = createNode(5);

    printf("Original list: ");

    printList(head);

    // Reorder the list based on odd and even indices

    struct ListNode\* reordered = oddEvenList(head);

    printf("Reordered list: ");

    printList(reordered);

    // Free the memory allocated for the linked lists

    while (head != NULL) {

        struct ListNode\* temp = head;

        head = head->next;

        free(temp);

    }

    return 0;

}

## Best Time To Buy And Sell Stock (Single day to buy and different day to sell once)

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

**Example 1:**

**Input:** prices = [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.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

**Example 2:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, no transactions are done and the max profit = 0

**C :**

#include <stdio.h>

#include <limits.h>

int maxProfit(int\* prices, int pricesSize) {

    if (pricesSize <= 1) {

        return 0; // No transactions can be done with less than 2 prices

    }

    int minPrice = INT\_MAX; // Initialize minimum price to maximum possible value

    int maxProfit = 0;

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

        // Update the minimum price if the current price is smaller

        if (prices[i] < minPrice) {

            minPrice = prices[i];

        }

        // Update the maximum profit if selling at the current price yields a higher profit

        else if (prices[i] - minPrice > maxProfit) {

            maxProfit = prices[i] - minPrice;

        }

    }

    return maxProfit;

}

int main() {

    int prices[] = {7, 1, 5, 3, 6, 4};

    int pricesSize = sizeof(prices) / sizeof(prices[0]);

    int profit = maxProfit(prices, pricesSize);

    printf("Maximum profit: %d\n", profit);

    return 0;

}

## Longest Consecutive Sequence

Given an unsorted array of integers nums, return *the length of the longest consecutive elements sequence.*

You must write an algorithm that runs in O(n) time.

**Example 1:**

**Input:** nums = [100,4,200,1,3,2]

**Output:** 4

**Explanation:** The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.

**Example 2:**

**Input:** nums = [0,3,7,2,5,8,4,6,0,1]

**Output:** 9

**C :**

#include <stdio.h>

#include <stdlib.h>

// HashSet structure

typedef struct {

    int\* array;

    int size;

} HashSet;

// Function to initialize a HashSet

HashSet\* initHashSet(int size) {

    HashSet\* set = (HashSet\*)malloc(sizeof(HashSet));

    set->array = (int\*)calloc(size, sizeof(int));

    set->size = size;

    return set;

}

// Function to insert an element into the HashSet

void insertHashSet(HashSet\* set, int value) {

    set->array[value] = 1;

}

// Function to check if an element exists in the HashSet

int containsHashSet(HashSet\* set, int value) {

    return set->array[value];

}

// Function to calculate the length of the longest consecutive sequence

int longestConsecutive(int\* nums, int numsSize) {

    if (numsSize <= 1) {

        return numsSize; // No consecutive sequence with one or zero elements

    }

    // Find min and max values in the array

    int min = nums[0], max = nums[0];

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

        if (nums[i] < min) {

            min = nums[i];

        } else if (nums[i] > max) {

            max = nums[i];

        }

    }

    // Initialize the HashSet

    HashSet\* set = initHashSet(max - min + 1);

    // Insert all elements into the HashSet

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

        insertHashSet(set, nums[i] - min);

    }

    // Calculate the length of the longest consecutive sequence

    int longestStreak = 0, currentStreak = 0;

    for (int i = 0; i <= max - min; i++) {

        if (containsHashSet(set, i)) {

            currentStreak++;

        } else {

            longestStreak = currentStreak > longestStreak ? currentStreak : longestStreak;

            currentStreak = 0;

        }

    }

    // Free memory allocated for the HashSet

    free(set->array);

    free(set);

    return longestStreak > currentStreak ? longestStreak : currentStreak;

}

int main() {

    int nums[] = {100, 4, 200, 1, 3, 2};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int longest = longestConsecutive(nums, numsSize);

    printf("Length of the longest consecutive sequence: %d\n", longest);

    return 0;

}

## Spiral Matrix

Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

**Example 1:**

A square with numbers and arrows

Description automatically generated

**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [1,2,3,6,9,8,7,4,5]

**Example 2:**

A grid of numbers and arrows

Description automatically generated

**Input:** matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

**Output:** [1,2,3,4,8,12,11,10,9,5,6,7]

**C :**

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

#include <stdio.h>

#include <stdlib.h>

int\* spiralOrder(int\*\* matrix, int matrixSize, int\* matrixColSize, int\* returnSize) {

    if (matrixSize == 0 || matrixColSize[0] == 0) {

        \*returnSize = 0;

        return NULL;

    }

    int totalElements = matrixSize \* matrixColSize[0];

    int\* result = (int\*)malloc(sizeof(int) \* totalElements);

    \*returnSize = totalElements;

    int top = 0, bottom = matrixSize - 1;

    int left = 0, right = matrixColSize[0] - 1;

    int index = 0;

    while (top <= bottom && left <= right) {

        // Traverse top row

        for (int i = left; i <= right; i++) {

            result[index++] = matrix[top][i];

        }

        top++;

        // Traverse right column

        for (int i = top; i <= bottom; i++) {

            result[index++] = matrix[i][right];

        }

        right--;

        // Check if there's a bottom row to traverse

        if (top <= bottom) {

            // Traverse bottom row

            for (int i = right; i >= left; i--) {

                result[index++] = matrix[bottom][i];

            }

            bottom--;

        }

        // Check if there's a left column to traverse

        if (left <= right) {

            // Traverse left column

            for (int i = bottom; i >= top; i--) {

                result[index++] = matrix[i][left];

            }

            left++;

        }

    }

    return result;

}

int main() {

    int matrixSize = 3;

    int matrixColSize[] = {4, 4, 4};

    int\*\* matrix = (int\*\*)malloc(matrixSize \* sizeof(int\*));

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

        matrix[i] = (int\*)malloc(matrixColSize[i] \* sizeof(int));

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

            matrix[i][j] = i \* matrixColSize[i] + j + 1;

        }

    }

    int returnSize;

    int\* result = spiralOrder(matrix, matrixSize, matrixColSize, &returnSize);

    printf("Spiral order: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    // Free dynamically allocated memory

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

        free(matrix[i]);

    }

    free(matrix);

    free(result);

    return 0;

}

## Populating Next Right Pointer In Each Node

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.

**Example 1:**

A diagram of a diagram

Description automatically generated with medium confidence

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

**Output:** [1,#,2,3,#,4,5,6,7,#]

**Explanation:** Given the above perfect 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.

**Example 2:**

**Input:** root = []

**Output:** []

**C :**

/\*\*

 \* Definition for a Node.

 \* struct Node {

 \*     int val;

 \*     struct Node \*left;

 \*     struct Node \*right;

 \*     struct Node \*next;

 \* };

 \*/

#include <stdio.h>

#include <stdlib.h>

// Definition for a Node.

struct Node {

    int val;

    struct Node \*left;

    struct Node \*right;

    struct Node \*next;

};

struct Node\* connect(struct Node\* root) {

    if (root == NULL) {

        return NULL;

    }

    struct Node\* leftmost = root; // Start from the leftmost node in the current level

    while (leftmost->left != NULL) {

        struct Node\* current = leftmost;

        // Traverse the current level and connect nodes in the next level

        while (current != NULL) {

            current->left->next = current->right;

            if (current->next != NULL) {

                current->right->next = current->next->left;

            }

            current = current->next; // Move to the next node in the current level

        }

        leftmost = leftmost->left; // Move to the leftmost node in the next level

    }

    return root;

}

// Helper function to create a new node

struct Node\* createNode(int val) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->val = val;

    newNode->left = NULL;

    newNode->right = NULL;

    newNode->next = NULL;

    return newNode;

}

// Function to print the next pointers of each node in the tree

void printNextPointers(struct Node\* root) {

    while (root != NULL) {

        struct Node\* current = root;

        while (current != NULL) {

            printf("%d -> ", current->val);

            current = current->next;

        }

        printf("NULL\n");

        root = root->left;

    }

}

int main() {

    // Create a sample tree

    struct Node\* root = createNode(1);

    root->left = createNode(2);

    root->right = createNode(3);

    root->left->left = createNode(4);

    root->left->right = createNode(5);

    root->right->left = createNode(6);

    root->right->right = createNode(7);

    // Connect the nodes

    struct Node\* connectedRoot = connect(root);

    // Print the next pointers of each node in the tree

    printf("Next pointers of each node in the tree:\n");

    printNextPointers(connectedRoot);

    // Free dynamically allocated memory

    // Note: This part is omitted for brevity

    return 0;

}

## String Compression

Given an array of characters chars, compress it using the following algorithm:

Begin with an empty string s. For each group of **consecutive repeating characters** in chars:

* If the group's length is 1, append the character to s.
* Otherwise, append the character followed by the group's length.

The compressed string s **should not be returned separately**, but instead, be stored **in the input character array chars**. Note that group lengths that are 10 or longer will be split into multiple characters in chars.

After you are done **modifying the input array,** return *the new length of the array*.

You must write an algorithm that uses only constant extra space.

**Example 1:**

**Input:** chars = ["a","a","b","b","c","c","c"]

**Output:** Return 6, and the first 6 characters of the input array should be: ["a","2","b","2","c","3"]

**Explanation:** The groups are "aa", "bb", and "ccc". This compresses to "a2b2c3".

**Example 2:**

**Input:** chars = ["a"]

**Output:** Return 1, and the first character of the input array should be: ["a"]

**Explanation:** The only group is "a", which remains uncompressed since it's a single character.

**Example 3:**

**Input:** chars = ["a","b","b","b","b","b","b","b","b","b","b","b","b"]

**Output:** Return 4, and the first 4 characters of the input array should be: ["a","b","1","2"].

**Explanation:** The groups are "a" and "bbbbbbbbbbbb". This compresses to "ab12".

**C :**

#include <stdio.h>

#include <stdlib.h>

int compress(char\* chars, int charsSize) {

    if (charsSize <= 1) {

        return charsSize;

    }

    int writeIndex = 0, readIndex = 0;

    while (readIndex < charsSize) {

        char currentChar = chars[readIndex];

        int count = 0;

        while (readIndex < charsSize && chars[readIndex] == currentChar) {

            readIndex++;

            count++;

        }

        if (count == 1) {

            chars[writeIndex++] = currentChar;

        } else {

            chars[writeIndex++] = currentChar;

            // Process the count and update the array

            int tempCount = count;

            while (tempCount > 0) {

                chars[writeIndex++] = '0' + tempCount % 10;

                tempCount /= 10;

            }

            // Reverse the count characters in the array

            int start = writeIndex - count + 1;

            int end = writeIndex - 1;

            while (start < end) {

                char temp = chars[start];

                chars[start] = chars[end];

                chars[end] = temp;

                start++;

                end--;

            }

        }

    }

    return writeIndex;

}

int main() {

    char chars[] = {'a', 'a', 'b', 'b', 'c', 'c', 'c'};

    int charsSize = sizeof(chars) / sizeof(chars[0]);

    printf("Original array: ");

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

        printf("%c ", chars[i]);

    }

    printf("\n");

    int compressedSize = compress(chars, charsSize);

    printf("Compressed array: ");

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

        printf("%c ", chars[i]);

    }

    printf("\n");

    return 0;

}

## Construct Binary Tree From Inorder and post Order Traversal

Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return *the binary tree*.

**Example 1:**

A diagram of a triangle

Description automatically generated

**Input:** inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]

**Output:** [3,9,20,null,null,15,7]

**Example 2:**

**Input:** inorder = [-1], postorder = [-1]

**Output:** [-1]

**Constraints:**

* 1 <= inorder.length <= 3000
* postorder.length == inorder.length
* -3000 <= inorder[i], postorder[i] <= 3000
* inorder and postorder consist of **unique** values.
* Each value of postorder also appears in inorder.
* inorder is **guaranteed** to be the inorder traversal of the tree.
* postorder is **guaranteed** to be the postorder traversal of the tree.

**C :**

#include <stdio.h>

#include <stdlib.h>

// TreeNode structure

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Function to create a new TreeNode

struct TreeNode\* createNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

// Helper function to construct the binary tree from inorder and postorder traversals

struct TreeNode\* helper(int\* inorder, int in\_left, int in\_right, int\* post\_idx, int\* postorder) {

    // If there are no elements to construct subtrees

    if (in\_left > in\_right)

        return NULL;

    // Pick up post\_idx element as a root

    int root\_val = postorder[(\*post\_idx)];

    struct TreeNode\* root = createNode(root\_val);

    // Root splits inorder list into left and right subtrees

    int index = 0;

    for (int i = in\_left; i <= in\_right; i++) {

        if (inorder[i] == root\_val) {

            index = i;

            break;

        }

    }

    // Recursion

    (\*post\_idx)--;

    // Build the right subtree

    root->right = helper(inorder, index + 1, in\_right, post\_idx, postorder);

    // Build the left subtree

    root->left = helper(inorder, in\_left, index - 1, post\_idx, postorder);

    return root;

}

// Function to construct the binary tree from inorder and postorder traversals

struct TreeNode\* buildTree(int\* inorder, int inorderSize, int\* postorder, int postorderSize) {

    int post\_idx = postorderSize - 1;

    return helper(inorder, 0, inorderSize - 1, &post\_idx, postorder);

}

// Function to perform inorder traversal of the binary tree

void inorderTraversal(struct TreeNode\* root) {

    if (root != NULL) {

        inorderTraversal(root->left);

        printf("%d ", root->val);

        inorderTraversal(root->right);

    }

}

int main() {

    // Example inorder and postorder traversals

    int inorder[] = {9, 3, 15, 20, 7};

    int inorderSize = sizeof(inorder) / sizeof(inorder[0]);

    int postorder[] = {9, 15, 7, 20, 3};

    int postorderSize = sizeof(postorder) / sizeof(postorder[0]);

    // Construct the binary tree

    struct TreeNode\* root = buildTree(inorder, inorderSize, postorder, postorderSize);

    // Perform inorder traversal to verify the constructed tree

    printf("Inorder traversal of the constructed tree: ");

    inorderTraversal(root);

    printf("\n");

    // Free allocated memory for the tree nodes (not included here for brevity)

    return 0;

}

## Valid Sudoku

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits 1-9 without repetition.
2. Each column must contain the digits 1-9 without repetition.
3. Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

**Note:**

* A Sudoku board (partially filled) could be valid but is not necessarily solvable.
* Only the filled cells need to be validated according to the mentioned rules.

**Example 1:**

A square puzzle with numbers

Description automatically generated

**Input:** board =

[["5","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

**Output:** true

**Example 2:**

**Input:** board =

[["8","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

**Output:** false

**Explanation:** Same as Example 1, except with the **5** in the top left corner being modified to **8**. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

**C :**

#include <stdbool.h>

#include <stdio.h>

bool isValidSudoku(char\*\* board, int boardSize, int\* boardColSize) {

    // Arrays to check for duplicate digits in rows, columns, and sub-boxes

    bool row[9][9] = {false};

    bool col[9][9] = {false};

    bool box[9][9] = {false};

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

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

            if (board[i][j] != '.') {

                int digit = board[i][j] - '1';

                // Check for duplicate in the same row

                if (row[i][digit]) {

                    return false;

                }

                row[i][digit] = true;

                // Check for duplicate in the same column

                if (col[j][digit]) {

                    return false;

                }

                col[j][digit] = true;

                // Check for duplicate in the same 3x3 sub-box

                int boxIndex = (i / 3) \* 3 + j / 3;

                if (box[boxIndex][digit]) {

                    return false;

                }

                box[boxIndex][digit] = true;

            }

        }

    }

    return true;

}

int main() {

    char\* board[] = {

        "53..7....",

        "6..195...",

        ".98....6.",

        "8...6...3",

        "4..8.3..1",

        "7...2...6",

        ".6....28.",

        "...419..5",

        "....8..79"

    };

    int boardSize = sizeof(board) / sizeof(board[0]);

    int boardColSize[] = {9, 9, 9, 9, 9, 9, 9, 9, 9};

    bool valid = isValidSudoku(board, boardSize, boardColSize);

    if (valid) {

        printf("The Sudoku board is valid.\n");

    } else {

        printf("The Sudoku board is not valid.\n");

    }

    return 0;

}

Time/Space Complexity:O(n2)/O(n2)

#include <stdbool.h>

bool isValidSudoku(char\*\* board, int boardSize, int\* boardColSize) {

    int N = 9;

    // Use a binary number to record previous occurrence

    int rows[N];

    int cols[N];

    int boxes[N];

    for (int r = 0; r < N; r++) {

        for (int c = 0; c < N; c++) {

            // Check if the position is filled with a number

            if (board[r][c] == '.') {

                continue;

            }

            int val = board[r][c] - '0';

            int pos = 1 << (val - 1);

            // Check the row

            if ((rows[r] & pos) > 0) {

                return false;

            }

            rows[r] |= pos;

            // Check the column

            if ((cols[c] & pos) > 0) {

                return false;

            }

            cols[c] |= pos;

            // Check the box

            int idx = (r / 3) \* 3 + c / 3;

            if ((boxes[idx] & pos) > 0) {

                return false;

            }

            boxes[idx] |= pos;

        }

    }

    return true;

}

int main() {

    // Example usage

    char\* board[9] = {"53..7....",

                      "6..195...",

                      ".98....6.",

                      "8...6...3",

                      "4..8.3..1",

                      "7...2...6",

                      ".6....28.",

                      "...419..5",

                      "....8..79"};

    int boardSize = 9;

    int boardColSize[9] = {9, 9, 9, 9, 9, 9, 9, 9, 9};

    bool result = isValidSudoku(board, boardSize, boardColSize);

    if (result) {

        printf("Valid Sudoku board\n");

    } else {

        printf("Invalid Sudoku board\n");

    }

    return 0;

}

Time/Space Complexity : O(n2)/O(n)

## Path Sum

Given the root of a binary tree and an integer targetSum, return true if the tree has a **root-to-leaf** path such that adding up all the values along the path equals targetSum.

A **leaf** is a node with no children.

**Example 1:**

A diagram of a tree

Description automatically generated

**Input:** root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

**Output:** true

**Explanation:** The root-to-leaf path with the target sum is shown.

**Example 2:**

A diagram of a connected structure

Description automatically generated with medium confidence

**Input:** root = [1,2,3], targetSum = 5

**Output:** false

**Explanation:** There two root-to-leaf paths in the tree:

(1 --> 2): The sum is 3.

(1 --> 3): The sum is 4.

There is no root-to-leaf path with sum = 5.

**Example 3:**

**Input:** root = [], targetSum = 0

**Output:** false

**Explanation:** Since the tree is empty, there are no root-to-leaf paths.

**C :**

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// TreeNode structure

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Function to create a new TreeNode

struct TreeNode\* createNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

// Function to check if a binary tree has a path sum equal to the target sum

bool hasPathSum(struct TreeNode\* root, int targetSum) {

    // Base case: If the current node is NULL, return false.

    if (root == NULL) {

        return false;

    }

    // Subtract the value of the current node from the target sum.

    targetSum -= root->val;

    // If the current node is a leaf node and the remaining sum is 0, return true.

    if (root->left == NULL && root->right == NULL) {

        return targetSum == 0;

    }

    // Recursively check the left and right subtrees.

    bool leftResult = hasPathSum(root->left, targetSum);

    bool rightResult = hasPathSum(root->right, targetSum);

    // Return true if either the left or right subtree has a path sum equal to the target sum.

    return leftResult || rightResult;

}

int main() {

    // Construct a sample binary tree

    struct TreeNode\* root = createNode(5);

    root->left = createNode(4);

    root->right = createNode(8);

    root->left->left = createNode(11);

    root->left->left->left = createNode(7);

    root->left->left->right = createNode(2);

    root->right->left = createNode(13);

    root->right->right = createNode(4);

    root->right->right->right = createNode(1);

    // Target sum to search for

    int targetSum = 22;

    // Check if the binary tree has a path sum equal to the target sum

    bool result = hasPathSum(root, targetSum);

    // Print the result

    if (result) {

        printf("The binary tree has a path sum equal to %d.\n", targetSum);

    } else {

        printf("The binary tree does not have a path sum equal to %d.\n", targetSum);

    }

    // Free allocated memory for the tree nodes (not included here for brevity)

    return 0;

}

## Design Add and Search Words DataStructure

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the WordDictionary class:

* WordDictionary() Initializes the object.
* void addWord(word) Adds word to the data structure, it can be matched later.
* bool search(word) Returns true if there is any string in the data structure that matches word or false otherwise. word may contain dots '.' where dots can be matched with any letter.

**Example:**

**Input**

["WordDictionary","addWord","addWord","addWord","search","search","search","search"]

[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]

**Output**

[null,null,null,null,false,true,true,true]

**Explanation**

WordDictionary wordDictionary = new WordDictionary();

wordDictionary.addWord("bad");

wordDictionary.addWord("dad");

wordDictionary.addWord("mad");

wordDictionary.search("pad"); // return False

wordDictionary.search("bad"); // return True

wordDictionary.search(".ad"); // return True

wordDictionary.search("b.."); // return True

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Definition for a trie node

typedef struct TrieNode {

    bool isEnd;

    struct TrieNode\* children[26];

} TrieNode;

// Definition for WordDictionary

typedef struct {

    TrieNode\* root;

} WordDictionary;

// Helper function to create a new trie node

TrieNode\* createTrieNode() {

    TrieNode\* node = (TrieNode\*)malloc(sizeof(TrieNode));

    node->isEnd = false;

    memset(node->children, 0, sizeof(node->children));

    return node;

}

WordDictionary\* wordDictionaryCreate() {

    WordDictionary\* obj = (WordDictionary\*)malloc(sizeof(WordDictionary));

    obj->root = createTrieNode();

    return obj;

}

void addWordToTrie(TrieNode\* root, char\* word) {

    TrieNode\* node = root;

    for (int i = 0; i < strlen(word); ++i) {

        int index = word[i] - 'a';

        if (node->children[index] == NULL) {

            node->children[index] = createTrieNode();

        }

        node = node->children[index];

    }

    node->isEnd = true;

}

void wordDictionaryAddWord(WordDictionary\* obj, char\* word) {

    addWordToTrie(obj->root, word);

}

bool searchInTrie(TrieNode\* node, char\* word) {

    for (int i = 0; i < strlen(word); ++i) {

        if (word[i] == '.') {

            for (int j = 0; j < 26; ++j) {

                if (node->children[j] != NULL &&

                    searchInTrie(node->children[j], word + i + 1)) {

                    return true;

                }

            }

            return false;

        } else {

            int index = word[i] - 'a';

            if (node->children[index] == NULL) {

                return false;

            }

            node = node->children[index];

        }

    }

    return node->isEnd;

}

bool wordDictionarySearch(WordDictionary\* obj, char\* word) {

    return searchInTrie(obj->root, word);

}

void freeTrie(TrieNode\* node) {

    if (node == NULL) {

        return;

    }

    for (int i = 0; i < 26; ++i) {

        freeTrie(node->children[i]);

    }

    free(node);

}

void wordDictionaryFree(WordDictionary\* obj) {

    freeTrie(obj->root);

    free(obj);

}

int main() {

    WordDictionary\* obj = wordDictionaryCreate();

    // Example usage

    wordDictionaryAddWord(obj, "bad");

    wordDictionaryAddWord(obj, "dad");

    wordDictionaryAddWord(obj, "mad");

    printf("%s\n", wordDictionarySearch(obj, "pad") ? "true" : "false"); // Output: false

    printf("%s\n", wordDictionarySearch(obj, "bad") ? "true" : "false"); // Output: true

    printf("%s\n", wordDictionarySearch(obj, ".ad") ? "true" : "false"); // Output: true

    printf("%s\n", wordDictionarySearch(obj, "b..") ? "true" : "false"); // Output: true

    wordDictionaryFree(obj);

    return 0;

}

/\*\*

 \* Your WordDictionary struct will be instantiated and called as such:

 \* WordDictionary\* obj = wordDictionaryCreate();

 \* wordDictionaryAddWord(obj, word);

 \* bool param\_2 = wordDictionarySearch(obj, word);

 \* wordDictionaryFree(obj);

\*/

## Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all of the following rules**:

1. Each of the digits 1-9 must occur exactly once in each row.
2. Each of the digits 1-9 must occur exactly once in each column.
3. Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

The '.' character indicates empty cells.

**Example 1:**

A square puzzle with numbers

Description automatically generated

**Input:** board = [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]

**Output:** [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]

**Explanation:** The input board is shown above and the only valid solution is shown below:

A square grid with red and black numbers

Description automatically generated

**C :**

#define SIZE 9

bool isSafe(char\*\* board, int row, int col, char num) {

    // Check if 'num' is not already present in the current row and column

    for (int x = 0; x < SIZE; x++) {

        if (board[row][x] == num || board[x][col] == num) {

            return false;

        }

    }

    // Check if 'num' is not already present in the current 3x3 sub-grid

    int startRow = row - row % 3;

    int startCol = col - col % 3;

    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < 3; j++) {

            if (board[i + startRow][j + startCol] == num) {

                return false;

            }

        }

    }

    return true;

}

bool findUnassignedLocation(char\*\* board, int\* row, int\* col) {

    for (\*row = 0; \*row < SIZE; (\*row)++) {

        for (\*col = 0; \*col < SIZE; (\*col)++) {

            if (board[\*row][\*col] == '.') {

                return true;

            }

        }

    }

    return false;

}

bool solve(char\*\* board) {

    int row, col;

    if (!findUnassignedLocation(board, &row, &col)) {

        // No unassigned location, puzzle is solved

        return true;

    }

    for (char num = '1'; num <= '9'; num++) {

        if (isSafe(board, row, col, num)) {

            // Try placing 'num' in the current position

            board[row][col] = num;

            // Recursively solve the rest of the puzzle

            if (solve(board)) {

                return true;

            }

            // If placing 'num' didn't lead to a solution, backtrack

            board[row][col] = '.';

        }

    }

    // No number can be placed in the current position

    return false;

}

void solveSudoku(char\*\* board, int boardSize, int\* boardColSize) {

    solve(board);

}

int main() {

    char\* sudoku[SIZE] = {

        "53..7....",

        "6..195...",

        ".98....6.",

        "8...6...3",

        "4..8.3..1",

        "7...2...6",

        ".6....28.",

        "...419..5",

        "....8..79"

    };

    printf("Sudoku puzzle before solving:\n");

    for (int i = 0; i < SIZE; i++) {

        printf("%s\n", sudoku[i]);

    }

    solveSudoku(sudoku, SIZE, NULL);

    printf("\nSudoku puzzle after solving:\n");

    for (int i = 0; i < SIZE; i++) {

        printf("%s\n", sudoku[i]);

    }

    return 0;

}

## Max Area Of Island

You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected **4-directionally** (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The **area** of an island is the number of cells with a value 1 in the island.

Return *the maximum****area****of an island in*grid. If there is no island, return 0.

**Example 1:**

A colorful squares with numbers

Description automatically generated

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

**Output:** 6

**Explanation:** The answer is not 11, because the island must be connected 4-directionally.

**Example 2:**

**Input:** grid = [[0,0,0,0,0,0,0,0]]

**Output:** 0

**C :**

int maxAreaOfIsland(int\*\* grid, int gridSize, int\* gridColSize) {

    int maxArea = 0;

    if (grid == NULL || gridSize == 0) {

        return maxArea;

    }

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

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

            if (grid[i][j] == 1) {

                int area = evaluateGrid(grid, i, j, gridSize, gridColSize[i]);

                maxArea = area > maxArea ? area : maxArea;

            }

        }

    }

    return maxArea;

}

int evaluateGrid(int\*\* grid, int i, int j, int row, int col) {

    if (i < 0 || j < 0 || i >= row || j >= col || grid[i][j] != 1) {

        return 0;

    }

    grid[i][j] = 2;

    int area = 1; // Current cell contributes to the area

    // Explore neighbors

    area += evaluateGrid(grid, i - 1, j, row, col);

    area += evaluateGrid(grid, i + 1, j, row, col);

    area += evaluateGrid(grid, i, j - 1, row, col);

    area += evaluateGrid(grid, i, j + 1, row, col);

    return area;

}

int main() {

    // Example grid

    int row = 5;

    int col = 5;

    int \*\*grid = (int \*\*)malloc(row \* sizeof(int \*));

    for (int i = 0; i < row; i++) {

        grid[i] = (int \*)malloc(col \* sizeof(int));

    }

    grid[0][0] = 1;

    grid[0][1] = 1;

    grid[0][2] = 0;

    grid[0][3] = 0;

    grid[0][4] = 0;

    grid[1][0] = 1;

    grid[1][1] = 1;

    grid[1][2] = 0;

    grid[1][3] = 0;

    grid[1][4] = 0;

    grid[2][0] = 0;

    grid[2][1] = 0;

    grid[2][2] = 0;

    grid[2][3] = 1;

    grid[2][4] = 1;

    grid[3][0] = 0;

    grid[3][1] = 0;

    grid[3][2] = 0;

    grid[3][3] = 1;

    grid[3][4] = 1;

    grid[4][0] = 0;

    grid[4][1] = 0;

    grid[4][2] = 0;

    grid[4][3] = 1;

    grid[4][4] = 1;

    int maxSize = maxAreaOfIsland(grid, row, &col);

    printf("The maximum area of the island is: %d\n", maxSize);

    // Free memory

    for (int i = 0; i < row; i++) {

        free(grid[i]);

    }

    free(grid);

    return 0;

}

## Valid Parantheses

Given a string s 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.
3. Every close bracket has a corresponding open bracket of the same type.

**Example 1:**

**Input:** s = "()"

**Output:** true

**Example 2:**

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

**Output:** true

**Example 3:**

**Input:** s = "(]"

**Output:** false

**C :**

bool isValid(char\* s) {

    int length = 0;

    char\* stack = (char\*)malloc(strlen(s) \* sizeof(char));

    for (int i = 0; s[i] != '\0'; i++) {

        if (s[i] == '(' || s[i] == '[' || s[i] == '{') {

            stack[length++] = s[i];

        } else {

            if (length == 0) {

                free(stack);

                return false; // No corresponding open bracket

            }

            char openBracket = stack[--length];

            if ((s[i] == ')' && openBracket != '(') ||

                (s[i] == ']' && openBracket != '[') ||

                (s[i] == '}' && openBracket != '{')) {

                free(stack);

                return false; // Mismatched brackets

            }

        }

    }

    bool result = (length == 0); // Stack should be empty for a valid string

    free(stack);

    return result;

}

int main() {

    // Example string

    char\* inputString = "{[()]}";

    // Check if the input string is valid

    bool isValidString = isValid(inputString);

    // Print the result

    if (isValidString) {

        printf("The input string is valid.\n");

    } else {

        printf("The input string is not valid.\n");

    }

    return 0;

}

## Merge Two Sorted Lists

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

**Example 1:**

A diagram of a diagram

Description automatically generated

**Input:** list1 = [1,2,4], list2 = [1,3,4]

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

**Example 2:**

**Input:** list1 = [], list2 = []

**Output:** []

**Example 3:**

**Input:** list1 = [], list2 = [0]

**Output:** [0]

**C :**

struct ListNode\* mergeTwoLists(struct ListNode\* list1, struct ListNode\* list2){

    struct ListNode \*dummy = (struct ListNode \*)malloc(sizeof(struct ListNode));

    dummy->next = NULL;

    struct ListNode \*curr = dummy;

    while (list1 != NULL && list2 != NULL) {

        if (list1->val < list2->val) {

            dummy->next = list1;

            list1 = list1->next;

            dummy = dummy->next;

        } else {

            dummy->next = list2;

            list2 = list2->next;

            dummy = dummy->next;

        }

    }

    while (list1 != NULL) {

        dummy->next = list1;

        list1 = list1->next;

        dummy = dummy->next;

    }

    while (list2 != NULL) {

        dummy->next = list2;

        list2 = list2->next;

        dummy = dummy->next;

    }

    return curr->next;

}

int main() {

    // Define two sorted linked lists

    struct ListNode\* list1 = NULL;

    struct ListNode\* list2 = NULL;

    // Populate list1

    list1 = (struct ListNode\*)malloc(sizeof(struct ListNode));

    list1->val = 1;

    list1->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    list1->next->val = 2;

    list1->next->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    list1->next->next->val = 4;

    list1->next->next->next = NULL;

    // Populate list2

    list2 = (struct ListNode\*)malloc(sizeof(struct ListNode));

    list2->val = 1;

    list2->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    list2->next->val = 3;

    list2->next->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    list2->next->next->val = 4;

    list2->next->next->next = NULL;

    // Merge the two lists

    struct ListNode\* mergedList = mergeTwoLists(list1, list2);

    // Print the merged list

    printf("Merged List: ");

    while (mergedList != NULL) {

        printf("%d ", mergedList->val);

        mergedList = mergedList->next;

    }

    printf("\n");

    // Free the memory allocated for the lists

    struct ListNode\* temp;

    while (list1 != NULL) {

        temp = list1;

        list1 = list1->next;

        free(temp);

    }

    while (list2 != NULL) {

        temp = list2;

        list2 = list2->next;

        free(temp);

    }

    return 0;

}

## Container With Most Water

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

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

Return *the maximum amount of water a container can store*.

**Notice** that you may not slant the container.

**Example 1:**

A blue and black graph

Description automatically generated

**Input:** height = [1,8,6,2,5,4,8,3,7]

**Output:** 49

**Explanation:** The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

**Example 2:**

**Input:** height = [1,1]

**Output:** 1

**C :**

#define MAX(a,b) ((a) > (b)?(a):(b))

#define MIN(a,b) ((a) > (b)?(b):(a))

int maxArea(int\* height, int heightSize) {

    int max\_area = INT\_MIN;

    int left = 0;

    int right = heightSize-1;

    while (left < right) {

        max\_area = MAX(max\_area,(MIN(height[left],height[right])\*(abs(left-right))));

        if (height[left] < height[right]) left++;

        else right--;

    }

    return max\_area;

}

int main() {

    // Sample input array of heights

    int height[] = {1, 8, 6, 2, 5, 4, 8, 3, 7};

    int heightSize = sizeof(height) / sizeof(height[0]);

    // Calculate the maximum area

    int result = maxArea(height, heightSize);

    // Print the result

    printf("The maximum area of water that can be held is: %d\n", result);

    return 0;

}

## Pascals Triangle

Given an integer numRows, return the first numRows of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:

A white and blue hexagons with black numbers

Description automatically generated

**Example 1:**

**Input:** numRows = 5

**Output:** [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

**Example 2:**

**Input:** numRows = 1

**Output:** [[1]]

**C :**

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

int\*\* generate(int numRows, int\* returnSize, int\*\* returnColumnSizes) {

    if (numRows <= 0) {

        \*returnSize = 0;

        \*returnColumnSizes = NULL;

        return NULL;

    }

    // Allocate memory for the result

    int\*\* result = (int\*\*)malloc(sizeof(int\*) \* numRows);

    \*returnColumnSizes = (int\*)malloc(sizeof(int) \* numRows);

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

        result[i] = (int\*)malloc(sizeof(int) \* (i + 1));

        (\*returnColumnSizes)[i] = i + 1;

        // Set the first and last elements to 1

        result[i][0] = result[i][i] = 1;

        // Calculate the middle elements based on the previous row

        for (int j = 1; j < i; j++) {

            result[i][j] = result[i - 1][j - 1] + result[i - 1][j];

        }

    }

    \*returnSize = numRows;

    return result;

}

int main() {

    // Sample input for numRows

    int numRows = 5;

    // Variables to store the result and column sizes

    int\*\* result;

    int\* returnSize;

    int\* returnColumnSizes;

    // Call the generate function

    result = generate(numRows, returnSize, &returnColumnSizes);

    // Print the generated Pascal's triangle

    printf("Pascal's Triangle with %d rows:\n", numRows);

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

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

            printf("%d ", result[i][j]);

        }

        printf("\n");

    }

    // Free allocated memory

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

        free(result[i]);

    }

    free(result);

    free(returnColumnSizes);

    return 0;

}

## Valid Anagram

Given two strings s and t, return true *if* t *is an anagram of* s*, and* false *otherwise*.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

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

**Output:** true

**Example 2:**

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

**Output:** false

**C :**

bool isAnagram(char\* s, char\* t) {

    // Check if the lengths of both strings are equal

    int lenS = 0, lenT = 0;

    while (s[lenS] != '\0') {

        lenS++;

    }

    while (t[lenT] != '\0') {

        lenT++;

    }

    if (lenS != lenT) {

        return false;

    }

    // Initialize an array to store the frequency of characters

    int frequency[26] = {0};  // Assuming input only contains lowercase English letters

    // Increment frequency for characters in string s

    for (int i = 0; i < lenS; i++) {

        frequency[s[i] - 'a']++;

    }

    // Decrement frequency for characters in string t

    for (int i = 0; i < lenT; i++) {

        frequency[t[i] - 'a']--;

    }

    // Check if all frequencies are zero

    for (int i = 0; i < 26; i++) {

        if (frequency[i] != 0) {

            return false;

        }

    }

    // If all checks pass, the strings are anagrams

    return true;

}

int main() {

    // Sample input strings

    char s[] = "anagram";

    char t[] = "nagaram";

    // Call the isAnagram function

    bool result = isAnagram(s, t);

    // Output the result

    if (result) {

        printf("'%s' and '%s' are anagrams.\n", s, t);

    } else {

        printf("'%s' and '%s' are not anagrams.\n", s, t);

    }

    return 0;

}

## Unique Paths

There is a robot on an m x n grid. The robot is initially located at the **top-left corner** (i.e., grid[0][0]). The robot tries to move to the **bottom-right corner** (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

Given the two integers m and n, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to 2 \* 109.

**Example 1:**

A blue and white checkered pattern

Description automatically generated

**Input:** m = 3, n = 7

**Output:** 28

**Example 2:**

**Input:** m = 3, n = 2

**Output:** 3

**Explanation:** From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

1. Right -> Down -> Down

2. Down -> Down -> Right

3. Down -> Right -> Down

**C :**

int uniquePaths(int m, int n) {

   // Create a 2D array to store the number of unique paths for each position.

    int dp[m][n];

    // Initialize the leftmost column (only one way to reach any position in the leftmost column).

    for (int i = 0; i < m; i++) {

        dp[i][0] = 1;

    }

    // Initialize the top row (only one way to reach any position in the top row).

    for (int j = 0; j < n; j++) {

        dp[0][j] = 1;

    }

    // Fill in the rest of the array by summing the number of unique paths from the top and left.

    for (int i = 1; i < m; i++) {

        for (int j = 1; j < n; j++) {

            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];

        }

    }

    // The result is stored in the bottom-right corner of the array.

    return dp[m - 1][n - 1];

}

int main() {

    // Sample input values for m and n

    int m = 3;

    int n = 7;

    // Call the uniquePaths function

    int result = uniquePaths(m, n);

    // Output the result

    printf("Number of unique paths for a %d x %d grid: %d\n", m, n, result);

    return 0;

}

## Longest Increasing Subsequence

Given an integer array nums, return *the length of the longest****strictly increasing***

***subsequence***

.

**Example 1:**

**Input:** nums = [10,9,2,5,3,7,101,18]

**Output:** 4

**Explanation:** The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

**Example 2:**

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

**Output:** 4

**Example 3:**

**Input:** nums = [7,7,7,7,7,7,7]

**Output:** 1

**C :**

int lengthOfLIS(int\* nums, int numsSize) {

    if (numsSize == 0) {

        return 0;

    }

    // Initialize an array to store the elements of the increasing subsequence

    int sub[numsSize];

    sub[0] = nums[0];

    int subSize = 1;

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

        int num = nums[i];

        if (num > sub[subSize - 1]) {

            sub[subSize] = num;

            subSize++;

        } else {

            // Find the first element in sub that is greater than or equal to num

            int j = 0;

            while (num > sub[j]) {

                j++;

            }

            sub[j] = num;

        }

    }

    return subSize;

}

int main() {

    // Sample input array

    int nums[] = {10, 9, 2, 5, 3, 7, 101, 18};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Call the lengthOfLIS function

    int result = lengthOfLIS(nums, numsSize);

    // Output the result

    printf("Length of the longest increasing subsequence: %d\n", result);

    return 0;

}

## Group Anagrams

Given an array of strings strs, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

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

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

**Example 2:**

**Input:** strs = [""]

**Output:** [[""]]

**Example 3:**

**Input:** strs = ["a"]

**Output:** [["a"]]

**C :**

class Solution {

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

        HashMap<String, List<String>> hm = new HashMap<String, List<String>>();

        int[] count\_map = new int[26];

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

            Arrays.fill(count\_map, 0);

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

                count\_map[strs[i].charAt(j)-'a']++;

            }

            StringBuilder sb = new StringBuilder();

            for (int m = 0; m < 26; m++) {

                sb.append("\*");

                sb.append(count\_map[m]);

            }

            if (!hm.containsKey(sb.toString())) {

                hm.put(sb.toString(), new ArrayList());

            }

            hm.get(sb.toString()).add(strs[i]);

            System.out.println(sb.toString());

        }

        return new ArrayList(hm.values());

    }

}

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

// Structure to represent a hash table node

struct Node {

    char\* key;

    char\*\* values;

    int size;

    struct Node\* next;

};

// Function to calculate the hash value for a string

unsigned int calculateHash(const char\* str) {

    unsigned int hash = 5381;

    int c;

    while ((c = \*str++)) {

        hash = ((hash << 5) + hash) + c; // Hash \* 33 + character

    }

    return hash;

}

// Function to insert a key and value into the hash table

void insertIntoHash(struct Node\*\* hashTable, const char\* key, const char\* value) {

    unsigned int hashValue = calculateHash(key);

    int bucket = hashValue % 1009; // Use a prime number as the hash table size

    // Create a new node

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->key = strdup(key);

    newNode->values = (char\*\*)malloc(sizeof(char\*));

    newNode->values[0] = strdup(value);

    newNode->size = 1;

    newNode->next = NULL;

    // Insert into the hash table

    if (hashTable[bucket] == NULL) {

        hashTable[bucket] = newNode;

    } else {

        newNode->next = hashTable[bucket];

        hashTable[bucket] = newNode;

    }

}

// Function to group anagrams

char\*\*\* groupAnagrams(char\*\* strs, int strsSize, int\* returnSize, int\*\* returnColumnSizes) {

    // Initialize variables

    \*returnSize = 0;

    struct Node\* hashTable[1009] = {NULL}; // Hash table with 1009 buckets (prime number)

    char\*\*\* result = NULL;

    // Iterate through each string in the input

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

        // Count the frequency of each character in the string

        int count\_map[26] = {0};

        for (int j = 0; j < strlen(strs[i]); j++) {

            count\_map[strs[i][j] - 'a']++;

        }

        // Build a key using the frequency of characters

        char key[100];

        int keyIndex = 0;

        for (int k = 0; k < 26; k++) {

            keyIndex += sprintf(key + keyIndex, "%d\*", count\_map[k]);

        }

        // Check if the key is already in the hash table

        unsigned int hashValue = calculateHash(key);

        int bucket = hashValue % 1009;

        struct Node\* current = hashTable[bucket];

        while (current != NULL) {

            if (strcmp(current->key, key) == 0) {

                // Anagram found, add the original string to the values

                current->values = realloc(current->values, (current->size + 1) \* sizeof(char\*));

                current->values[current->size] = strdup(strs[i]);

                current->size++;

                break;

            }

            current = current->next;

        }

        // If the key is not in the hash table, create a new node

        if (current == NULL) {

            insertIntoHash(hashTable, key, strs[i]);

        }

    }

    // Count the number of unique groups

    int uniqueGroups = 0;

    for (int i = 0; i < 1009; i++) {

        struct Node\* current = hashTable[i];

        while (current != NULL) {

            uniqueGroups++;

            current = current->next;

        }

    }

    // Allocate memory for the result

    result = (char\*\*\*)malloc(uniqueGroups \* sizeof(char\*\*));

    \*returnColumnSizes = (int\*)malloc(uniqueGroups \* sizeof(int));

    // Populate the result array

    int resultIndex = 0;

    for (int i = 0; i < 1009; i++) {

        struct Node\* current = hashTable[i];

        while (current != NULL) {

            int groupSize = current->size;

            // Allocate memory for the group

            result[resultIndex] = (char\*\*)malloc(groupSize \* sizeof(char\*));

            (\*returnColumnSizes)[resultIndex] = groupSize;

            // Copy values to the result array

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

                result[resultIndex][j] = strdup(current->values[j]);

            }

            // Move to the next group

            resultIndex++;

            current = current->next;

        }

    }

    // Free memory used by the hash table

    for (int i = 0; i < 1009; i++) {

        struct Node\* current = hashTable[i];

        while (current != NULL) {

            free(current->key);

            for (int j = 0; j < current->size; j++) {

                free(current->values[j]);

            }

            free(current->values);

            struct Node\* temp = current;

            current = current->next;

            free(temp);

        }

    }

    \*returnSize = uniqueGroups;

    return result;

}

// Function to free memory allocated for the grouped anagrams result

void freeGroupedAnagrams(char\*\*\* result, int returnSize, int\* returnColumnSizes) {

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

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

            free(result[i][j]);

        }

        free(result[i]);

    }

    free(result);

    free(returnColumnSizes);

}

## Longest Palindromic SubString

Given a string s, return *the longest*

*palindromic*

*substring*

 in s.

**Example 1:**

**Input:** s = "babad"

**Output:** "bab"

**Explanation:** "aba" is also a valid answer.

**Example 2:**

**Input:** s = "cbbd"

**Output:** "bb"

**C :**

class Solution {

    public String longestPalindrome(String s) {

        int n = s.length();

        boolean[][] dp = new boolean[n][n];

        // All substrings of length 1 are palindromes

        for (int i = 0; i < n; i++) {

            dp[i][i] = true;

        }

        int start = 0; // Start index of the longest palindrome

        int maxLength = 1; // Length of the longest palindrome

        // Check substrings of length 2

        for (int i = 0; i < n - 1; i++) {

            if (s.charAt(i) == s.charAt(i + 1)) {

                dp[i][i + 1] = true;

                start = i;

                maxLength = 2;

            }

        }

        // Check substrings of length 3 and more

        for (int len = 3; len <= n; len++) {

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

                int j = i + len - 1;

                if (dp[i + 1][j - 1] && s.charAt(i) == s.charAt(j)) {

                    dp[i][j] = true;

                    start = i;

                    maxLength = len;

                }

            }

        }

        return s.substring(start, start + maxLength);

    }

}

char\* longestPalindrome(char\* s) {

    int n = strlen(s);

    // Create a new string to insert '#' between characters

    char\* modifiedString = (char\*)malloc((2 \* n + 2) \* sizeof(char));

    int j = 0;

    for (int i = 0; i < n; i++) {

        modifiedString[j++] = '#';

        modifiedString[j++] = s[i];

    }

    modifiedString[j++] = '#';

    modifiedString[j] = '\0';

    // Initialize the palindrome array

    int\* palindrome = (int\*)malloc((2 \* n + 2) \* sizeof(int));

    memset(palindrome, 0, (2 \* n + 2) \* sizeof(int));

    int center = 0, right = 0;

    for (int i = 0; i < 2 \* n + 1; i++) {

        int mirror = 2 \* center - i;

        // Check if the mirror of i is within the right boundary

        if (i < right) {

            palindrome[i] = (right - i < palindrome[mirror]) ? right - i : palindrome[mirror];

        }

        // Expand around the center

        int a = i + (1 + palindrome[i]);

        int b = i - (1 + palindrome[i]);

        while (a < 2 \* n + 1 && b >= 0 && modifiedString[a] == modifiedString[b]) {

            palindrome[i]++;

            a++;

            b--;

        }

        // If palindrome[i] reaches the boundary of the current rightmost palindrome,

        // update the center and right boundary accordingly

        if (i + palindrome[i] > right) {

            center = i;

            right = i + palindrome[i];

        }

    }

    // Find the maximum element in the palindrome array

    int maxLen = 0, centerIndex = 0;

    for (int i = 0; i < 2 \* n + 1; i++) {

        if (palindrome[i] > maxLen) {

            maxLen = palindrome[i];

            centerIndex = i;

        }

    }

    // Extract the longest palindrome from the modified string

    int start = (centerIndex - maxLen) / 2;

    char\* result = (char\*)malloc((maxLen + 1) \* sizeof(char));

    strncpy(result, s + start, maxLen);

    result[maxLen] = '\0';

    // Clean up

    free(modifiedString);

    free(palindrome);

    return result;

}

## Three Sum

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

**Example 1:**

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

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

**Explanation:**

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.

The distinct triplets are [-1,0,1] and [-1,-1,2].

Notice that the order of the output and the order of the triplets does not matter.

**Example 2:**

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

**Output:** []

**Explanation:** The only possible triplet does not sum up to 0.

**Example 3:**

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

**Output:** [[0,0,0]]

**Explanation:** The only possible triplet sums up to 0.

**C :**

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

int compare(const void \*a, const void \*b) {

    return (\*(int \*)a - \*(int \*)b);

}

int\*\* threeSum(int\* nums, int numsSize, int\* returnSize, int\*\* returnColumnSizes) {

     // Sort the array to simplify the solution

    qsort(nums, numsSize, sizeof(int), compare);

    // Allocate memory for the result

    int\*\* result = NULL;

    \*returnSize = 0;

    \*returnColumnSizes = NULL;

    // Iterate through the array

    for (int i = 0; i < numsSize - 2; ++i) {

        // Skip duplicates to avoid duplicate triplets

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

            continue;

        }

        int left = i + 1;

        int right = numsSize - 1;

        while (left < right) {

            int sum = nums[i] + nums[left] + nums[right];

            if (sum == 0) {

                // Found a triplet, allocate memory for it

                result = (int\*\*)realloc(result, (\*returnSize + 1) \* sizeof(int\*));

                result[\*returnSize] = (int\*)malloc(3 \* sizeof(int));

                // Assign values to the triplet

                result[\*returnSize][0] = nums[i];

                result[\*returnSize][1] = nums[left];

                result[\*returnSize][2] = nums[right];

                // Increment the result size

                (\*returnSize)++;

                // Skip duplicates for left pointer

                while (left < right && nums[left] == nums[left + 1]) {

                    left++;

                }

                // Skip duplicates for right pointer

                while (left < right && nums[right] == nums[right - 1]) {

                    right--;

                }

                // Move pointers

                left++;

                right--;

            } else if (sum < 0) {

                left++;

            } else {

                right--;

            }

        }

    }

    // Allocate memory for column sizes

    \*returnColumnSizes = (int\*)malloc(\*returnSize \* sizeof(int));

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

        (\*returnColumnSizes)[i] = 3;

    }

    return result;

}

int main() {

    int nums[] = {-1, 0, 1, 2, -1, -4};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int returnSize;

    int\* returnColumnSizes;

    int\*\* result = threeSum(nums, numsSize, &returnSize, &returnColumnSizes);

    printf("[");

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

        printf("[");

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

            printf("%d", result[i][j]);

            if (j < returnColumnSizes[i] - 1) {

                printf(", ");

            }

        }

        printf("]");

        if (i < returnSize - 1) {

            printf(", ");

        }

    }

    printf("]\n");

    // Free allocated memory

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

        free(result[i]);

    }

    free(result);

    free(returnColumnSizes);

    return 0;

}

class Solution {

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

        HashSet<Integer> hs = new HashSet<Integer>();

        HashMap<Integer,Integer> hm = new HashMap<Integer,Integer>();

        HashSet<List<Integer>> result = new HashSet<List<Integer>>();

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

            if (hs.add(nums[i])) {

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

                    int target = -nums[i]-nums[j];

                    if (hm.containsKey(target) && hm.get(target) == i) {

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

                        entry.add(nums[i]);

                        entry.add(nums[j]);

                        entry.add(target);

                        Collections.sort(entry);

                        result.add(entry);

                    } else {

                        hm.put(nums[j],i);

                    }

                }

            }

        }

        return new ArrayList(result);

    }

}

/\*

[-1,0,1,2,-1,-4]

duplicates: - 1 , hm : 0:0 and complement = 1

duplicates: -1 , hm: 0:0 and 2:0 complement = -1

\*/

## Single Element In A Sorted Array

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

Return *the single element that appears only once*.

Your solution must run in O(log n) time and O(1) space.

**Example 1:**

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

**Output:** 2

**Example 2:**

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

**Output:** 10

**C :**

int singleNonDuplicate(int\* nums, int numsSize) {

    int r = 0;

    for (int i = 0; i < numsSize; i++)

    r = r ^ nums[i];

    return r;

}

int main() {

    int nums[] = {1, 1, 2, 3, 3, 4, 4, 8, 8};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int result = singleNonDuplicate(nums, numsSize);

    printf("The single non-duplicate element is: %d\n", result);

    return 0;

}

## Letter Combinations Of A Phone Number

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

A close-up of a phone keypad

Description automatically generated

**Example 1:**

**Input:** digits = "23"

**Output:** ["ad","ae","af","bd","be","bf","cd","ce","cf"]

**Example 2:**

**Input:** digits = ""

**Output:** []

**Example 3:**

**Input:** digits = "2"

**Output:** ["a","b","c"]

**C :**

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

// Helper function to append a character to a string

void append(char\* str, char c) {

    int len = strlen(str);

    str[len] = c;

    str[len + 1] = '\0';

}

// Helper function to perform backtracking

void backtrack(char\*\* result, char\* digits, char\* current, char\*\* mapping, int index, int\* returnSize) {

    // If the current combination is complete, add it to the result

    if (digits[index] == '\0') {

        result[\*returnSize] = strdup(current);

        (\*returnSize)++;

        return;

    }

    // Get the letters corresponding to the current digit

    char\* letters = mapping[digits[index] - '0'];

    // Iterate through the letters and backtrack

    for (int i = 0; letters[i] != '\0'; i++) {

        append(current, letters[i]);

        backtrack(result, digits, current, mapping, index + 1, returnSize);

        // Remove the last character to backtrack

        current[strlen(current) - 1] = '\0';

    }

}

// Main function to generate letter combinations

char\*\* letterCombinations(char\* digits, int\* returnSize) {

    // Define the mapping of digits to letters

    char\* mapping[] = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

    // Allocate space for the result

    char\*\* result = (char\*\*)malloc(sizeof(char\*) \* 1000);

    \*returnSize = 0;

    // Check if the input is empty

    if (digits == NULL || \*digits == '\0') {

        return result;

    }

    // Start the backtracking process

    char current[1000] = "";

    backtrack(result, digits, current, mapping, 0, returnSize);

    return result;

}

int main() {

    char\* digits = "23"; // Example input

    int returnSize;

    char\*\* result = letterCombinations(digits, &returnSize);

    printf("Letter combinations for %s are:\n", digits);

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

        printf("%s\n", result[i]);

        free(result[i]); // Free memory allocated for each combination

    }

    free(result); // Free memory allocated for the array of combinations

    return 0;

}

class Solution {

    HashMap<Character, String> hm = new HashMap<Character, String>();

    public void generate\_words(String current\_word, String digits, int current\_index, List<String> result) {

        if (current\_index == digits.length()) {

            result.add(current\_word);

            return;

        }

        String seq = hm.get(digits.charAt(current\_index));

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

            generate\_words(current\_word+seq.charAt(i), digits, current\_index+1, result);

        }

    }

    public void helper(String digits, List<String> result) {

        generate\_words("", digits, 0, result);

    }

    public List<String> letterCombinations(String digits) {

        List<String> result = new ArrayList<String>();

        hm.put('2',"abc");

        hm.put('3',"def");

        hm.put('4',"ghi");

        hm.put('5',"jkl");

        hm.put('6',"mno");

        hm.put('7',"pqrs");

        hm.put('8',"tuv");

        hm.put('9',"wxyz");

        if (digits.length() == 0) return result;

        helper(digits,result);

        return result;

    }

}

## Find The Index Of The First Occurrence In The String (strstr – needle in haystack)

Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

**Example 1:**

**Input:** haystack = "sadbutsad", needle = "sad"

**Output:** 0

**Explanation:** "sad" occurs at index 0 and 6.

The first occurrence is at index 0, so we return 0.

**Example 2:**

**Input:** haystack = "leetcode", needle = "leeto"

**Output:** -1

**Explanation:** "leeto" did not occur in "leetcode", so we return -1.

**C :**

int strStr(char \* haystack, char \* needle){

    int i = 0;

    int j = 0;

    int start = -1;

    while (i < strlen(haystack)) {

        if (haystack[i] == needle[j]) {

            if (start == -1) {

                start = i;

            }

            if (j == strlen(needle)-1) {

                return start;

            }

            i++;

            j++;

        }

        else {

            if (i == strlen(haystack)-1) {

                break;

            }

            i=start+1;

            start = i;

            j = 0;

        }

    }

    return -1;

}

int main() {

    char haystack[] = "hello";

    char needle[] = "ll";

    int index = strStr(haystack, needle);

    if (index != -1) {

        printf("Needle found at index: %d\n", index);

    } else {

        printf("Needle not found.\n");

    }

    return 0;

}

## Best Time to Buy And Sell Stock II (Buy Sell Any Times)

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the****maximum****profit you can achieve*.

**Example 1:**

**Input:** prices = [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.

Total profit is 4 + 3 = 7.

**Example 2:**

**Input:** prices = [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.

Total profit is 4.

**Example 3:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

**C :**

int maxProfit(int\* prices, int pricesSize){

    int  i = 0;

    int result = 0;

    for (i = 1; i < pricesSize; i++) {

        if (prices[i] > prices[i-1]) result+=(prices[i]-prices[i-1]) ;

    }

    return result;

}

int main() {

    int prices[] = {7, 1, 5, 3, 6, 4};

    int pricesSize = sizeof(prices) / sizeof(prices[0]);

    int max\_profit = maxProfit(prices, pricesSize);

    printf("Max profit: %d\n", max\_profit);

    return 0;

}

## Intersection Of Two Arrays

Given two integer arrays nums1 and nums2, return an array of their intersection. Each element in the result must be **unique** and you may return the result in **any order**.

**Example 1:**

**Input:** nums1 = [1,2,2,1], nums2 = [2,2]

**Output:** [2]

**Example 2:**

**Input:** nums1 = [4,9,5], nums2 = [9,4,9,8,4]

**Output:** [9,4]

**Explanation:** [4,9] is also accepted.

**C :**

int compare(const void\* a, const void\* b) {

    return (\*(int\*)a - \*(int\*)b);

}

int\* intersection(int\* nums1, int nums1Size, int\* nums2, int nums2Size, int\* returnSize) {

    qsort(nums1, nums1Size, sizeof(int), compare);

    qsort(nums2, nums2Size, sizeof(int), compare);

    int\* result = (int\*)malloc(sizeof(int) \* (nums1Size + nums2Size));

    \*returnSize = 0;

    int i = 0, j = 0;

    while (i < nums1Size && j < nums2Size) {

        if (nums1[i] == nums2[j]) {

            // Add the common element to the result if it is not already present

            if (\*returnSize == 0 || result[\*returnSize - 1] != nums1[i]) {

                result[(\*returnSize)++] = nums1[i];

            }

            i++;

            j++;

        } else if (nums1[i] < nums2[j]) {

            i++;

        } else {

            j++;

        }

    }

    return result;

}

int main() {

    // Example input arrays

    int nums1[] = {1, 2, 2, 1};

    int nums1Size = sizeof(nums1) / sizeof(nums1[0]);

    int nums2[] = {2, 2};

    int nums2Size = sizeof(nums2) / sizeof(nums2[0]);

    // Variables to store the result

    int returnSize;

    int\* result = intersection(nums1, nums1Size, nums2, nums2Size, &returnSize);

    // Print the result

    printf("Intersection: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    // Free allocated memory

    free(result);

    return 0;

}

## Climbing Stairs

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

**Example 1:**

**Input:** n = 2

**Output:** 2

**Explanation:** There are two ways to climb to the top.

1. 1 step + 1 step

2. 2 steps

**Example 2:**

**Input:** n = 3

**Output:** 3

**Explanation:** There are three ways to climb to the top.

1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

**C :**

int climbStairs(int n) {

    if (n == 0) return 0;

    if (n == 1) return 1;

    if (n == 2) return 2;

    int \*result = (int \*)malloc(sizeof(int)\*(n+1));

    int i = 0;

    result[0] = 0;

    result[1] = 1;

    result[2] = 2;

    for (i = 3; i <= n; i++) {

        result[i] = result[i-1] + result[i-2];

    }

    return result[n];

}

int main() {

    int n = 5; // Example input: number of steps

    int ways = climbStairs(n);

    printf("Number of distinct ways to climb %d steps: %d\n", n, ways);

    return 0;

}

## Convert Sorted Array To Binary Search Tree

*binary search tree*.

**Example 1:**

A diagram of a network

Description automatically generated

**Input:** nums = [-10,-3,0,5,9]

**Output:** [0,-3,9,-10,null,5]

**Explanation:** [0,-10,5,null,-3,null,9] is also accepted:

A diagram of a network

Description automatically generated

**Example 2:**

A close up of a number

Description automatically generated

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

**Output:** [3,1]

**Explanation:** [1,null,3] and [3,1] are both height-balanced BSTs.

**C :**

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

struct TreeNode\* helper(int\* nums, int left, int right) {

    if (left > right)

        return NULL;

    int middle = (left + right) / 2;

    struct TreeNode\* result = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    result->val = nums[middle];

    result->left = helper(nums, left, middle - 1);

    result->right = helper(nums, middle + 1, right);

    return result;

}

struct TreeNode\* sortedArrayToBST(int\* nums, int numsSize) {

    return helper(nums, 0, numsSize - 1);

}

int main() {

    int nums[] = {-10, -3, 0, 5, 9}; // Example sorted array

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Convert sorted array to binary search tree

    struct TreeNode\* root = sortedArrayToBST(nums, numsSize);

    // Print the inorder traversal of the resulting binary search tree

    printf("Inorder traversal of the constructed BST: ");

    inorderTraversal(root);

    printf("\n");

    // Free dynamically allocated memory for the binary search tree

    // Note: This is optional in this example since it's a small program,

    // but it's good practice to free allocated memory in real applications.

    // You can implement a function to free the memory recursively.

    return 0;

}

## Kth Largest Element In Array

 Given an integer array nums and an integer k, return *the* kth *largest element in the array*.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Can you solve it without sorting?

**Example 1:**

**Input:** nums = [3,2,1,5,6,4], k = 2

**Output:** 5

**Example 2:**

**Input:** nums = [3,2,3,1,2,4,5,5,6], k = 4

**Output:** 4

**C :**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int findKthLargest(int\* nums, int numsSize, int k) {

    if (numsSize == 1)

        return nums[0];

    // Find the minimum and maximum values in the array

    int min = INT\_MAX, max = INT\_MIN;

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

        if (nums[i] > max)

            max = nums[i];

        if (nums[i] < min)

            min = nums[i];

    }

printf("\nmin = %d and max = %d",min,max);

    // Create an array to store counts

    int range = max - min + 1;

    int\* counts = (int\*)malloc(range \* sizeof(int));

    if (!counts) {

        printf("Memory allocation failed.\n");

        exit(1);

    }

    for (int i = 0; i < range; i++)

        counts[i] = 0;

    // Count occurrences of each number

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

        counts[nums[i] - min]++;

printf("\ncounts[%d] = %d",nums[i]-min,counts[nums[i] - min]);

}

    // Find the kth largest element

    int len = range - 1;

    while (k > 0) {

        if (counts[len] == 0) {

printf("\nHere counts[len] : %d = %d",counts[len],len);

            len--;

}

        else {

            counts[len]--;

printf("\nThere counts[len] : %d = %d, k = %d",counts[len],len,k);

            k--;

        }

    }

    int result = len + min;

    free(counts);

    return result;

}

int main() {

    int nums[] = {3, 2, 1, 5, 6, 4};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int k = 2;

    printf("Kth largest element is: %d\n", findKthLargest(nums, numsSize, k));

    return 0;

}

Time/Space Complexity:O(n)/O(n)

## Kth Smallest Element In Array

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int findKthSmallest(int\* nums, int numsSize, int k) {

    if (numsSize == 1)

        return nums[0];

    // Find the minimum and maximum values in the array

    int max = INT\_MIN;

    int min = INT\_MAX;

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

        if (nums[i] > max)

            max = nums[i];

        if (nums[i] < min)

            min = nums[i];

    }

printf("\nmin = %d and max = %d",min,max);

    // Create an array to store counts

    int range = max - min + 1;

    int\* counts = (int\*)malloc(range \* sizeof(int));

    if (!counts) {

        printf("Memory allocation failed.\n");

        exit(1);

    }

    for (int i = 0; i < range; i++) {

        counts[i] = 0;

    }

    // Count occurrences of each number

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

        counts[nums[i] - min]++;

printf("\ncounts[%d] = %d",nums[i]-min,counts[nums[i] - min]);

    }

    // Find the kth smallest element

    int len = 0;

    while (k > 0) {

        if (counts[len] == 0) {

printf("\nHere counts[len] : %d = %d",counts[len],len);

            len++;

        } else {

            counts[len]--;

printf("\nThere counts[len] : %d = %d, k = %d",counts[len],len,k);

            k--;

        }

    }

    int result = len + min;

    free(counts);

    return result;

}

int main() {

    int nums[] = {3, 2, 1, 5, 6, 4};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int k = 2;

    printf("Kth smallest element is: %d\n", findKthSmallest(nums, numsSize, k));

    return 0;

}

Time/Space Complexity:O(n)/O(n)

## Binary Tree ZigZag Level Order Traversal

 Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

**Example 1:**

A diagram of a triangle

Description automatically generated

**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[3],[20,9],[15,7]]

**Example 2:**

**Input:** root = [1]

**Output:** [[1]]

**Example 3:**

**Input:** root = []

**Output:** []

**C :**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

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

        if (root == null) return levels;

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

        queue.add(root);

        int level = 0;

        while ( !queue.isEmpty() ) {

            // start the current level

            levels.add(new ArrayList<Integer>());

            // number of elements in the current level

            int level\_length = queue.size();

            for(int i = 0; i < level\_length; ++i) {

                TreeNode node = queue.remove();

                // fulfill the current level

                levels.get(level).add(node.val);

                // add child nodes of the current level

                // in the queue for the next level

                if (node.left != null) queue.add(node.left);

                if (node.right != null) queue.add(node.right);

            }

            // go to next level

            level++;

        }

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

            if (i % 2 == 1) {

                Collections.reverse(levels.get(i));

            }

        }

        return levels;

    }

}

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

// Function to get the height of a binary tree

int getHeight(struct TreeNode\* root) {

    if (root == NULL) {

        return 0;

    }

    int leftHeight = getHeight(root->left);

    int rightHeight = getHeight(root->right);

    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);

}

// Function to perform zigzag level order traversal of a binary tree

void zigzagTraversalHelper(struct TreeNode\* root, int\*\* result, int level, int\* returnColumnSizes) {

    if (root == NULL) {

        return;

    }

    // Determine the index based on the level and whether it's a zigzag or normal level

    int index = (level % 2 == 0) ? level : (returnColumnSizes[level] - 1 - level);

    // Add the current node's value to the result array

    result[level][index] = root->val;

    // Recursively traverse the left and right subtrees

    zigzagTraversalHelper(root->left, result, level + 1, returnColumnSizes);

    zigzagTraversalHelper(root->right, result, level + 1, returnColumnSizes);

}

// Function to perform zigzag level order traversal and return the result as a 2D array

int\*\* zigzagLevelOrder(struct TreeNode\* root, int\* returnSize, int\*\* returnColumnSizes) {

    // Get the height of the binary tree

    int height = getHeight(root);

    // Allocate memory for the result array

    int\*\* result = (int\*\*)malloc(height \* sizeof(int\*));

    \*returnColumnSizes = (int\*)malloc(height \* sizeof(int));

    // Initialize the return size and column sizes

    \*returnSize = height;

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

        int levelSize = 1 << i; // Number of nodes at the current level

        result[i] = (int\*)malloc(levelSize \* sizeof(int));

        (\*returnColumnSizes)[i] = levelSize;

    }

    // Perform zigzag level order traversal

    zigzagTraversalHelper(root, result, 0, \*returnColumnSizes);

    return result;

}

// Function to create a new binary tree node with the given value

struct TreeNode\* newNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

## Find Minimum In Rotated Sorted Array

 Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

* [4,5,6,7,0,1,2] if it was rotated 4 times.
* [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in O(log n) time.

**Example 1:**

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

**Output:** 1

**Explanation:** The original array was [1,2,3,4,5] rotated 3 times.

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2]

**Output:** 0

**Explanation:** The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

**Example 3:**

**Input:** nums = [11,13,15,17]

**Output:** 11

**Explanation:** The original array was [11,13,15,17] and it was rotated 4 times.

**C:**

int findMin(int\* nums, int numsSize) {

    // If the list has just one element then return that element.

    if (numsSize == 1) {

        return nums[0];

    }

    // Initializing left and right pointers.

    int left = 0, right = numsSize - 1;

    // If the last element is greater than the first element, then there is no rotation.

    // Already sorted array. Hence the smallest element is the first element. A[0]

    if (nums[right] > nums[0]) {

        return nums[0];

    }

    // Binary search

    while (left <= right) {

        // Find the mid element

        int mid = left + (right - left) / 2;

        // If the mid element is greater than its next element, then mid+1 element is the smallest.

        // This point would be the point of change. From higher to lower value.

        if (nums[mid] > nums[mid + 1]) {

            return nums[mid + 1];

        }

        // If the mid element is lesser than its previous element, then mid element is the smallest.

        if (nums[mid - 1] > nums[mid]) {

            return nums[mid];

        }

        // If the mid element's value is greater than the 0th element, this means

        // the least value is still somewhere to the right, as we are still dealing with

        // elements greater than nums[0].

        if (nums[mid] > nums[0]) {

            left = mid + 1;

        } else {

            // If nums[0] is greater than the mid value, then this means the smallest value

            // is somewhere to the left.

            right = mid - 1;

        }

    }

    return INT\_MAX;

}

int main() {

    int nums[] = {4, 5, 6, 7, 0, 1, 2}; // Example rotated sorted array

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Find the minimum element in the rotated sorted array

    int minElement = findMin(nums, numsSize);

    // Print the result

    printf("The minimum element in the rotated sorted array is: %d\n", minElement);

    return 0;

}

s

## Binary Search

 Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

**Example 2:**

**Input:** nums = [-1,0,3,5,9,12], target = 2

**Output:** -1

**Explanation:** 2 does not exist in nums so return -1

**C:**

int search(int\* nums, int numsSize, int target) {

       int left = 0;

    int right = numsSize - 1;



    while (left <= right) {

        int mid = left + (right - left) / 2;

        if (nums[mid] == target) {

            return mid; // Target found

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

            left = mid + 1; // Target may be in the right half

        } else {

            right = mid - 1; // Target may be in the left half

        }

    }

return -1;

}

int main() {

    int nums[] = {1, 3, 5, 7, 9, 11, 13}; // Sorted array

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int target = 7;

    // Search for the target in the array

    int result = search(nums, numsSize, target);

    // Print the result

    if (result != -1) {

        printf("Target %d found at index %d\n", target, result);

    } else {

        printf("Target %d not found in the array\n", target);

    }

    return 0;

}

## Construct Binary Tree From PreOrder And Inorder Traversal

 Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return *the binary tree*.

**Example 1:**

A diagram of a triangle

Description automatically generated

**Input:** preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]

**Output:** [3,9,20,null,null,15,7]

**Example 2:**

**Input:** preorder = [-1], inorder = [-1]

**Output:** [-1]

**C:**

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

struct TreeNode\* buildTreeHelper(int\* preorder, int preStart, int preEnd, int\* inorder, int inStart, int inEnd) {

    if (preStart > preEnd || inStart > inEnd) {

        return NULL;

    }

    // The first element in preorder is the root of the current subtree

    struct TreeNode\* root = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->val = preorder[preStart];

    root->left = root->right = NULL;

    // Find the index of the root in inorder

    int rootIndex;

    for (rootIndex = inStart; rootIndex <= inEnd; rootIndex++) {

        if (inorder[rootIndex] == root->val) {

            break;

        }

    }

    // Calculate the number of elements in the left subtree

    int leftSubtreeSize = rootIndex - inStart;

    // Recursively build the left and right subtrees

    root->left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize, inorder, inStart, rootIndex - 1);

    root->right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd, inorder, rootIndex + 1, inEnd);

    return root;

}

struct TreeNode\* buildTree(int\* preorder, int preorderSize, int\* inorder, int inorderSize) {

    return buildTreeHelper(preorder, 0, preorderSize - 1, inorder, 0, inorderSize - 1);

}

int main() {

    int preorder[] = {3, 9, 20, 15, 7};

    int preorderSize = sizeof(preorder) / sizeof(preorder[0]);

    int inorder[] = {9, 3, 15, 20, 7};

    int inorderSize = sizeof(inorder) / sizeof(inorder[0]);

    // Build the binary tree

    struct TreeNode\* root = buildTree(preorder, preorderSize, inorder, inorderSize);

    // Print the inorder traversal of the binary tree

    printf("Inorder traversal: ");

    printInorder(root);

    printf("\n");

    return 0;

}

## Remove Nth Node From End Of List

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

**Example 1:**

A diagram of a diagram

Description automatically generated

**Input:** head = [1,2,3,4,5], n = 2

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

**Example 2:**

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

**Output:** []

**Example 3:**

**Input:** head = [1,2], n = 1

**Output:** [1]

**C:**

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     struct ListNode \*next;

 \* };

 \*/

struct ListNode\* removeNthFromEnd(struct ListNode\* head, int n){

    struct ListNode \*curr = head;

    struct ListNode \*d = head;

    for (int i = 0; i < n; i++) {

        curr = curr->next;

    }

    if (curr == NULL) return head->next;

    while (curr->next != NULL) {

        d = d->next;

        curr = curr->next;

    }

    struct ListNode \*del = d->next;

    d->next = d->next->next;

    free(del);

    return head;

    /\*

        struct ListNode \*fast = head, \*slow = head;

        for (int i = 0; i < n; i++) fast = fast->next;

        if (fast == NULL) return head->next;

        while (fast->next) fast = fast->next, slow = slow->next;

        slow->next = slow->next->next;

        return head;

    \*/

}

int main() {

    // Create a sample linked list

    struct ListNode\* head = (struct ListNode\*)malloc(sizeof(struct ListNode));

    head->val = 1;

    head->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    head->next->val = 2;

    head->next->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    head->next->next->val = 3;

    head->next->next->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    head->next->next->next->val = 4;

    head->next->next->next->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    head->next->next->next->next->val = 5;

    head->next->next->next->next->next = NULL;

    printf("Original linked list: ");

    printList(head);

    // Remove the 2nd node from the end

    int n = 2;

    head = removeNthFromEnd(head, n);

    printf("Linked list after removing %d node from the end: ", n);

    printList(head);

    // Free memory allocated for the linked list

    struct ListNode\* temp;

    while (head != NULL) {

        temp = head;

        head = head->next;

        free(temp);

    }

    return 0;

}

## Jump Game

You are given an integer array nums. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return true*if you can reach the last index, or*false*otherwise*.

**Example 1:**

**Input:** nums = [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:** nums = [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.

class Solution {

    public boolean canJump(int[] nums) {

        int target = nums.length-1;

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

            if (i+nums[i] >= target) target = i;

        }

        return (target == 0);

    }

}

bool canJump(int\* nums, int numsSize) {

    int maxReach = 0;

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

        // If the current index is beyond the furthest position we can reach

        if (i > maxReach) {

            return false;

        }

        // Update the furthest position we can reach

        maxReach = (i + nums[i]) > maxReach ? (i + nums[i]) : maxReach;

        // If we can reach the last index, return true

        if (maxReach >= numsSize - 1) {

            return true;

        }

    }

    return false;

}

int main() {

    int nums[] = {2, 3, 1, 1, 4};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Test Java solution

    printf("Java solution: %s\n", canJumpJava(nums, numsSize) ? "true" : "false");

    // Test C solution

    printf("C solution: %s\n", canJump(nums, numsSize) ? "true" : "false");

    return 0;

}

## Insert Delete GetRandom O(1)

Implement the RandomizedSet class:

* RandomizedSet() Initializes the RandomizedSet object.
* bool insert(int val) Inserts an item val into the set if not present. Returns true if the item was not present, false otherwise.
* bool remove(int val) Removes an item val from the set if present. Returns true if the item was present, false otherwise.
* int getRandom() Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.

You must implement the functions of the class such that each function works in **average** O(1) time complexity.

**Example 1:**

**Input**

["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]

[[], [1], [2], [2], [], [1], [2], []]

**Output**

[null, true, false, true, 2, true, false, 2]

**Explanation**

RandomizedSet randomizedSet = new RandomizedSet();

randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomizedSet.remove(2); // Returns false as 2 does not exist in the set.

randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now contains [1,2].

randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.

randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [2].

randomizedSet.insert(2); // 2 was already in the set, so return false.

randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() will always return 2.

// Define the structure for RandomizedSet

typedef struct {

    int\* dict;  // Map (dictionary) to store the index of each value

    int\* list;  // List to store the elements

    int size;   // Current size of the list

    int capacity;  // Capacity of the list

} RandomizedSet;

// Function to initialize RandomizedSet

RandomizedSet\* randomizedSetCreate() {

    RandomizedSet\* set = (RandomizedSet\*)malloc(sizeof(RandomizedSet));

    set->dict = (int\*)calloc(1000, sizeof(int));  // Assuming values are in the range [0, 999]

    set->list = (int\*)malloc(sizeof(int) \* 1000);  // Assuming initial capacity is 1000

    set->size = 0;

    set->capacity = 1000;

    return set;

}

// Function to insert a value into RandomizedSet

bool randomizedSetInsert(RandomizedSet\* obj, int val) {

    if (obj->dict[val] != 0) {

        return false;  // Value already exists

    }

    if (obj->size == obj->capacity) {

        // Resize the list if needed

        obj->capacity \*= 2;

        obj->list = realloc(obj->list, sizeof(int) \* obj->capacity);

    }

    obj->dict[val] = obj->size + 1;  // Adding 1 to differentiate from zero (0 indicates not present)

    obj->list[obj->size] = val;

    obj->size++;

    return true;

}

// Function to remove a value from RandomizedSet

bool randomizedSetRemove(RandomizedSet\* obj, int val) {

    if (obj->dict[val] == 0) {

        return false;  // Value not present

    }

    int lastElement = obj->list[obj->size - 1];

    int idx = obj->dict[val] - 1;  // Subtracting 1 to get the actual index

    obj->list[idx] = lastElement;

    obj->dict[lastElement] = idx + 1;  // Update the index of the swapped value

    obj->size--;

    obj->dict[val] = 0;  // Marking the value as not present

    return true;

}

// Function to get a random element from RandomizedSet

int randomizedSetGetRandom(RandomizedSet\* obj) {

    int randomIndex = rand() % obj->size;

    return obj->list[randomIndex];

}

// Function to free the memory used by RandomizedSet

void randomizedSetFree(RandomizedSet\* obj) {

    free(obj->dict);

    free(obj->list);

    free(obj);

}

int main() {

    // Test the RandomizedSet functionality

    RandomizedSet\* obj = randomizedSetCreate();

    // Inserting values

    printf("Inserting elements...\n");

    randomizedSetInsert(obj, 1);

    randomizedSetInsert(obj, 2);

    randomizedSetInsert(obj, 3);

    // Printing elements

    printf("Current elements in RandomizedSet: ");

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

        printf("%d ", obj->list[i]);

    }

    printf("\n");

    // Removing an element

    printf("Removing element 2...\n");

    randomizedSetRemove(obj, 2);

    // Printing elements after removal

    printf("Current elements in RandomizedSet after removal: ");

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

        printf("%d ", obj->list[i]);

    }

    printf("\n");

    // Getting a random element

    printf("Random element: %d\n", randomizedSetGetRandom(obj));

    // Freeing memory

    randomizedSetFree(obj);

    return 0;

}

/\*\*

 \* Your RandomizedSet struct will be instantiated and called as such:

 \* RandomizedSet\* obj = randomizedSetCreate();

 \* bool param\_1 = randomizedSetInsert(obj, val);

 \* bool param\_2 = randomizedSetRemove(obj, val);

 \* int param\_3 = randomizedSetGetRandom(obj);

 \* randomizedSetFree(obj);

\*/

## Daily Temperatures

Given an array of integers temperatures represents the daily temperatures, return *an array* answer *such that* answer[i] *is the number of days you have to wait after the* ith *day to get a warmer temperature*. If there is no future day for which this is possible, keep answer[i] == 0 instead.

**Example 1:**

**Input:** temperatures = [73,74,75,71,69,72,76,73]

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

**Example 2:**

**Input:** temperatures = [30,40,50,60]

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

**Example 3:**

**Input:** temperatures = [30,60,90]

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

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

int\* dailyTemperatures(int\* temperatures, int temperaturesSize, int\* returnSize) {

    int\* result = (int\*)malloc(sizeof(int) \* temperaturesSize);

    \*returnSize = temperaturesSize;

    int\* stack = (int\*)malloc(sizeof(int) \* temperaturesSize);

    int top = -1;

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

        // Check if the current temperature is greater than the temperature at the top of the stack

        while (top >= 0 && temperatures[i] > temperatures[stack[top]]) {

            int prevIndex = stack[top--];

            result[prevIndex] = i - prevIndex;

        }

        // Push the current index onto the stack

        stack[++top] = i;

    }

    // Remaining elements in the stack have no warmer days

    while (top >= 0) {

        result[stack[top--]] = 0;

    }

    free(stack);

    return result;

}

int main() {

    int temperatures[] = {73, 74, 75, 71, 69, 72, 76, 73};

    int temperaturesSize = sizeof(temperatures) / sizeof(temperatures[0]);

    int returnSize = 0;

    int\* result = dailyTemperatures(temperatures, temperaturesSize, &returnSize);

    printf("Result: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    free(result);

    return 0;

}

## Reverse Integer

Given a signed 32-bit integer x, return x*with its digits reversed*. If reversing x causes the value to go outside the signed 32-bit integer range [-231, 231 - 1], then return 0.

**Assume the environment does not allow you to store 64-bit integers (signed or unsigned).**

**Example 1:**

**Input:** x = 123

**Output:** 321

**Example 2:**

**Input:** x = -123

**Output:** -321

**Example 3:**

**Input:** x = 120

**Output:** 21

class Solution {

    public int reverse(int x) {

        int rev = 0;

        while (x != 0) {

            int pop = x % 10;

            x /= 10;

            if (rev > Integer.MAX\_VALUE/10 || (rev == Integer.MAX\_VALUE / 10 && pop > 7)) return 0;

            if (rev < Integer.MIN\_VALUE/10 || (rev == Integer.MIN\_VALUE / 10 && pop < -8)) return 0;

            rev = rev \* 10 + pop;

        }

        return rev;

    }

}

int main() {

    int num = 123;

    int reversed = reverse(num);

    printf("Reversed: %d\n", reversed);

    return 0;

}

## Symmetric Tree

Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

**Example 1:**

A diagram of a tree

Description automatically generated

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

**Output:** true

**Example 2:**

A diagram of a network

Description automatically generated

**Input:** root = [1,2,2,null,3,null,3]

**Output:** false

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

bool isSymmetric(struct TreeNode\* root){

 if (root == NULL) {

        return true;

    }

    struct TreeNode\* stack[200];

    int top = -1;

    stack[++top] = root;

    stack[++top] = root;

    while (top >= 0) {

        struct TreeNode\* left = stack[top--];

        struct TreeNode\* right = stack[top--];

        if (left == NULL && right == NULL) {

            continue;

        }

        if (left == NULL || right == NULL) {

            return false;

        }

        if (left->val != right->val) {

            return false;

        }

        stack[++top] = left->left;

        stack[++top] = right->right;

        stack[++top] = left->right;

        stack[++top] = right->left;

    }

    return true;

}

int main() {

    // Create a test binary tree

    struct TreeNode\* root = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->val = 1;

    root->left = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->left->val = 2;

    root->right = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->right->val = 2;

    root->left->left = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->left->left->val = 3;

    root->left->right = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->left->right->val = 4;

    root->right->left = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->right->left->val = 4;

    root->right->right = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    root->right->right->val = 3;

    root->left->left->left = NULL;

    root->left->left->right = NULL;

    root->left->right->left = NULL;

    root->left->right->right = NULL;

    root->right->left->left = NULL;

    root->right->left->right = NULL;

    root->right->right->left = NULL;

    root->right->right->right = NULL;

    // Check if the binary tree is symmetric

    bool symmetric = isSymmetric(root);

    if (symmetric) {

        printf("The binary tree is symmetric.\n");

    } else {

        printf("The binary tree is not symmetric.\n");

    }

    // Free memory allocated for the binary tree

    free(root->left->left);

    free(root->left->right);

    free(root->right->left);

    free(root->right->right);

    free(root->left);

    free(root->right);

    free(root);

    return 0;

}

## Lowest Common Ancestor Of A binary tree

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**).”

**Example 1:**

A diagram of a tree

Description automatically generated

**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:**

A diagram of a tree

Description automatically generated

**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.

**Example 3:**

**Input:** root = [1,2], p = 1, q = 2

**Output:** 1

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

struct TreeNode\* lowestCommonAncestor(struct TreeNode\* root, struct TreeNode\* p, struct TreeNode\* q) {

    // Base case: If the current node is null or matches either p or q, return the current node

    if (!root || root == p || root == q) {

        return root;

    }

    // Recursive calls to find the LCA in the left and right subtrees

    struct TreeNode\* leftLCA = lowestCommonAncestor(root->left, p, q);

    struct TreeNode\* rightLCA = lowestCommonAncestor(root->right, p, q);

    // If both left and right subtrees have a result, the current node is the LCA

    if (leftLCA && rightLCA) {

        return root;

    }

    // If only one subtree has a result, propagate it upward

    return (leftLCA != NULL) ? leftLCA : rightLCA;

}

int main() {

    // Create a sample binary tree

    struct TreeNode\* root = newNode(3);

    root->left = newNode(5);

    root->right = newNode(1);

    root->left->left = newNode(6);

    root->left->right = newNode(2);

    root->left->right->left = newNode(7);

    root->left->right->right = newNode(4);

    root->right->left = newNode(0);

    root->right->right = newNode(8);

    // Define the nodes p and q

    struct TreeNode\* p = root->left;

    struct TreeNode\* q = root->right;

    // Find the lowest common ancestor of p and q

    struct TreeNode\* lca = lowestCommonAncestor(root, p, q);

    // Print the value of the lowest common ancestor

    printf("Lowest Common Ancestor of %d and %d is %d\n", p->val, q->val, lca->val);

    // Free memory allocated for the binary tree

    free(root->left->left);

    free(root->left->right->left);

    free(root->left->right->right);

    free(root->left->right);

    free(root->left);

    free(root->right->left);

    free(root->right->right);

    free(root->right);

    free(root);

    return 0;

}

## Merge K Sorted Lists

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

*Merge all the linked-lists into one sorted linked-list and return it.*

**Example 1:**

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

**Output:** [1,1,2,3,4,4,5,6]

**Explanation:** The linked-lists are:

[

1->4->5,

1->3->4,

2->6

]

merging them into one sorted list:

1->1->2->3->4->4->5->6

**Example 2:**

**Input:** lists = []

**Output:** []

**Example 3:**

**Input:** lists = [[]]

**Output:** []

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     struct ListNode \*next;

 \* };

 \*/

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     struct ListNode \*next;

 \* };

 \*/

struct ListNode\* mergeKLists(struct ListNode\*\* lists, int listsSize){

    int i = 0;

    int count = 0;

    int arr[10000];

    int temp = 0;

    int j = 0;

    for (i = 0; i < listsSize; i++) {

        while(lists[i]!=NULL) {

            arr[count] = lists[i]->val;

            count++;

            lists[i] = lists[i]->next;

        }

    }

    if (count == 0) return NULL;

    for(i = 0 ; i < count; i++) {

        for (j = i+1; j < count; j++) {

            if (arr[i] > arr[j]) {

                temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

    }

    struct ListNode \*result = (struct ListNode\*)malloc(sizeof(struct ListNode));

    struct ListNode \*head;

    head = result;

    for(i = 0 ; i < count; i++) {

        result->val = arr[i];

        result->next = NULL;

        if (i!=count-1) {

            result->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

            result = result->next;

        }

        else {

            result->next = NULL;

        }

    }

    return head;

}

int main() {

    // Create three sorted linked lists

    struct ListNode\* list1 = newNode(1);

    list1->next = newNode(4);

    list1->next->next = newNode(5);

    struct ListNode\* list2 = newNode(1);

    list2->next = newNode(3);

    list2->next->next = newNode(4);

    struct ListNode\* list3 = newNode(2);

    list3->next = newNode(6);

    // Store the lists in an array

    struct ListNode\* lists[] = {list1, list2, list3};

    // Merge the lists

    struct ListNode\* mergedList = mergeKLists(lists, 3);

    // Print the merged list

    printf("Merged List: ");

    struct ListNode\* temp = mergedList;

    while (temp != NULL) {

        printf("%d ", temp->val);

        temp = temp->next;

    }

    printf("\n");

    // Free memory allocated for the lists

    free(list1->next->next);

    free(list1->next);

    free(list1);

    free(list2->next->next);

    free(list2->next);

    free(list2);

    free(list3->next);

    free(list3);

    // Free memory allocated for the merged list

    while (mergedList != NULL) {

        struct ListNode\* nextNode = mergedList->next;

        free(mergedList);

        mergedList = nextNode;

    }

    return 0;

}

## SubSets

Given an integer array nums of **unique** elements, return *all possible*

*subsets*

*(the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

**Example 1:**

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

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

**Example 2:**

**Input:** nums = [0]

**Output:** [[],[0]]

// Function to count the number of set bits in an integer

int countSetBits(int n) {

    int count = 0;

    while (n) {

        count += n & 1;

        n >>= 1;

    }

    return count;

}

int\*\* subsets(int\* nums, int numsSize, int\* returnSize, int\*\* returnColumnSizes) {

    // Calculate the total number of subsets (2^n)

    int totalSubsets = 1 << numsSize;

    // Allocate memory for the result array

    int\*\* result = (int\*\*)malloc(totalSubsets \* sizeof(int\*));

    \*returnColumnSizes = (int\*)malloc(totalSubsets \* sizeof(int));

    \*returnSize = 0;

    // Iterate through all numbers from 0 to 2^n - 1

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

        // Calculate the size of the current subset

        int subsetSize = countSetBits(i);

        // Allocate memory for the current subset

        result[\*returnSize] = (int\*)malloc(subsetSize \* sizeof(int));

        (\*returnColumnSizes)[\*returnSize] = subsetSize;

        // Fill in the subset based on set bits in the binary representation of i

        int index = 0;

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

            if ((i >> j) & 1) {

                result[\*returnSize][index++] = nums[j];

            }

        }

        (\*returnSize)++;

    }

    return result;

}

int main() {

    int nums[] = {1, 2, 3};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int returnSize;

    int\* returnColumnSizes;

    int\*\* result = subsets(nums, numsSize, &returnSize, &returnColumnSizes);

    printf("Number of subsets: %d\n", returnSize);

    printf("Subsets:\n");

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

        printf("[");

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

            printf("%d", result[i][j]);

            if (j < returnColumnSizes[i] - 1) {

                printf(", ");

            }

        }

        printf("]\n");

    }

    // Free memory allocated for subsets

    freeSubsets(result, returnSize, returnColumnSizes);

    return 0;

}

## Can Place Flowers

You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in **adjacent** plots.

Given an integer array flowerbed containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer n, return true *if* n *new flowers can be planted in the* flowerbed *without violating the no-adjacent-flowers rule and* false *otherwise*.

**Example 1:**

**Input:** flowerbed = [1,0,0,0,1], n = 1

**Output:** true

**Example 2:**

**Input:** flowerbed = [1,0,0,0,1], n = 2

**Output:** false

bool canPlaceFlowers(int\* flowerbed, int flowerbedSize, int n) {

    int count = 0;

    if (flowerbedSize == 0) return false;

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

        if (flowerbed[i] == 0) {

            if (i == 0) {

                if (i == flowerbedSize-1) {

                    count++;

                    break;

                }

                if (flowerbed[i+1] != 1) {

                    flowerbed[i] = 1;

                    count++;

                    //i++;

                }

            } else if (i != flowerbedSize-1) {

                if (flowerbed[i-1] != 1 && flowerbed[i+1] != 1) {

                    flowerbed[i] = 1;

                    count++;

                    //i++;

                }

            } else {

                if (flowerbed[i-1] != 1) {

                    flowerbed[i] = 1;

                    count++;

                    //i++;

                }

            }

        }

    }

    if (count >= n) return true;

    else return false;

}

int main() {

    int flowerbed[] = {1, 0, 0, 0, 1};

    int flowerbedSize = sizeof(flowerbed) / sizeof(flowerbed[0]);

    int n = 1;

    bool result = canPlaceFlowers(flowerbed, flowerbedSize, n);

    printf("Can place %d flowers: %s\n", n, result ? "true" : "false");

    return 0;

}

## Roman To Integer

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

**Example 1:**

**Input:** s = "III"

**Output:** 3

**Explanation:** III = 3.

**Example 2:**

**Input:** s = "LVIII"

**Output:** 58

**Explanation:** L = 50, V= 5, III = 3.

**Example 3:**

int romanToInt(char \* s){

    int result = 0;

    int i = 0;

    // Create a lookup table for Roman numerals

    char roman[7] = {'I', 'V', 'X', 'L', 'C', 'D', 'M'};

    int value[7] = {1, 5, 10, 50, 100, 500, 1000};

    int length = strlen(s);

    for (i = 0; i < length; i++) {

        // Find the corresponding value for the current Roman numeral

        int current = 0;

        for (int j = 0; j < 7; j++) {

            if (s[i] == roman[j]) {

                current = value[j];

                break;

            }

        }

        // Check if subtraction is required

        if (i < length - 1) {

            int next = 0;

            for (int j = 0; j < 7; j++) {

                if (s[i+1] == roman[j]) {

                    next = value[j];

                    break;

                }

            }

            if (current < next) {

                result += (next - current);

                i++;

                continue;

            }

        }

        // Add the current value to the result

        result += current;

    }

    return result;

}

int main() {

    char romanNumeral[] = "MCMXCIV";

    int result = romanToInt(romanNumeral);

    printf("The integer value of %s is: %d\n", romanNumeral, result);

    return 0;

}

## Find The Duplicate Number

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only **one repeated number** in nums, return *this repeated number*.

You must solve the problem **without** modifying the array nums and uses only constant extra space.

**Example 1:**

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

**Output:** 2

**Example 2:**

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

**Output:** 3

int findDuplicate(int\* nums, int numsSize) {

        if (nums == NULL || numsSize <= 1) {

        // Handle invalid input

        return -1;

    }

    int slow = nums[0];

    int fast = nums[0];

    // Phase 1: Find the intersection point of the two pointers

    do {

        slow = nums[slow];

        fast = nums[nums[fast]];

    } while (slow != fast);

    // Phase 2: Find the entrance to the cycle

    slow = nums[0];

    while (slow != fast) {

        slow = nums[slow];

        fast = nums[fast];

    }

    return slow;

}

int main() {

    int nums[] = {1, 3, 4, 2, 2};

    int duplicate = findDuplicate(nums, sizeof(nums) / sizeof(nums[0]));

    printf("The duplicate number is: %d\n", duplicate);

    return 0;

}

## Product Of Array Except Self

Given an integer array nums, return *an array* answer *such that* answer[i] *is equal to the product of all the elements of* nums *except* nums[i].

The product of any prefix or suffix of nums is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

**Example 1:**

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

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

**Example 2:**

**Input:** nums = [-1,1,0,-3,3]

**Output:** [0,0,9,0,0]

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

int\* productExceptSelf(int\* nums, int numsSize, int\* returnSize) {

    if (nums == NULL || numsSize == 0) {

        \*returnSize = 0;

        return NULL;

    }

    // Initialize arrays to store products to the left and right of each element

    int\* leftProducts = (int\*)malloc(numsSize \* sizeof(int));

    int\* rightProducts = (int\*)malloc(numsSize \* sizeof(int));

    // Initialize return array

    int\* result = (int\*)malloc(numsSize \* sizeof(int));

    // Calculate products to the left of each element

    int leftProduct = 1;

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

        leftProducts[i] = leftProduct;

        leftProduct \*= nums[i];

    }

    // Calculate products to the right of each element

    int rightProduct = 1;

    for (int i = numsSize - 1; i >= 0; i--) {

        rightProducts[i] = rightProduct;

        rightProduct \*= nums[i];

    }

    // Calculate the final result by multiplying left and right products

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

        result[i] = leftProducts[i] \* rightProducts[i];

    }

    // Set the return size

    \*returnSize = numsSize;

    // Free allocated memory for intermediate arrays

    free(leftProducts);

    free(rightProducts);

    return result;

}

int main() {

    int nums[] = {1, 2, 3, 4};

    int returnSize;

    int\* result = productExceptSelf(nums, sizeof(nums) / sizeof(nums[0]), &returnSize);

    printf("Product of array elements except self:\n");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    // Free memory allocated for result array

    free(result);

    return 0;

}

## Coin Change

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

**Example 1:**

**Input:** coins = [1,2,5], amount = 11

**Output:** 3

**Explanation:** 11 = 5 + 5 + 1

**Example 2:**

**Input:** coins = [2], amount = 3

**Output:** -1

**Example 3:**

**Input:** coins = [1], amount = 0

**Output:** 0

public static int coinChange(int[] S, int sum)

{

if (sum == 0) {

return 0;

}

if (sum < 0) {

return Integer.MAX\_VALUE;

}

int coins = Integer.MAX\_VALUE;

for (int c: S)

{

int result = coinChange(S, sum - c);

if (result != Integer.MAX\_VALUE) {

coins = Integer.min(coins, result + 1);

}

}

return coins;

}

#define min(a,b) a < b ? a:b

int coinChange(int\* coins, int coinsSize, int amount) {

    // Create an array to store the minimum number of coins needed for each amount

    int dp[amount + 1];

    // Initialize the array with a value larger than the maximum possible number of coins

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

        dp[i] = INT\_MAX;

    }

    // Base case: 0 coins needed for amount 0

    dp[0] = 0;

    // Iterate through each coin denomination

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

        // Update dp array for each possible amount

        for (int j = coins[i]; j <= amount; j++) {

            // If using the current coin leads to a smaller number of coins needed, update dp

            if (dp[j - coins[i]] != INT\_MAX) {

                dp[j] = min(dp[j], dp[j - coins[i]] + 1);

            }

        }

    }

    // If dp[amount] is still INT\_MAX, no valid combination was found

    return (dp[amount] == INT\_MAX) ? -1 : dp[amount];

}

int main() {

    int coins[] = {1, 2, 5};

    int coinsSize = sizeof(coins) / sizeof(coins[0]);

    int amount = 11;

    int minCoins = coinChange(coins, coinsSize, amount);

    printf("Minimum number of coins needed to make up %d: %d\n", amount, minCoins);

    return 0;

}

## Contains Duplicate

Given an integer array nums, return true if any value appears **at least twice** in the array, and return false if every element is distinct.

**Example 1:**

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

**Output:** true

**Example 2:**

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

**Output:** false

**Example 3:**

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

**Output:** true

bool containsDuplicate(int\* nums, int numsSize) {

    // Create a hash set using an array of flags

    bool\* set = (bool\*)malloc(100000 \* sizeof(bool));

    // Initialize the set to false

    for (int i = 0; i < 100000; i++) {

        set[i] = false;

    }

    // Iterate through the array

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

        // If the element is already in the set, it's a duplicate

        if (set[nums[i]]) {

            free(set);

            return true;

        }

        // Mark the element as seen in the set

        set[nums[i]] = true;

    }

    // No duplicates found

    free(set);

    return false;

}

int main() {

    int nums[] = {1, 2, 3, 1};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    bool hasDuplicate = containsDuplicate(nums, numsSize);

    if (hasDuplicate) {

        printf("The array contains duplicates.\n");

    } else {

        printf("The array does not contain duplicates.\n");

    }

    return 0;

}

## Search A 2D Matrix

You are given an m x n integer matrix matrix with the following two properties:

* Each row is sorted in non-decreasing order.
* The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true *if* target *is in* matrix *or* false *otherwise*.

You must write a solution in O(log(m \* n)) time complexity.

**Example 1:**

A number grid with numbers

Description automatically generated with medium confidence

**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

**Output:** true

**Example 2:**

A number grid with black numbers

Description automatically generated

**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

**Output:** false

bool searchMatrix(int\*\* matrix, int matrixSize, int\* matrixColSize, int target){

    int row = matrixSize - 1;

    int col = 0;

    while (row >= 0 && col < \*matrixColSize) {

        if (matrix[row][col] > target) {

            row--;

        } else if (matrix[row][col] < target) {

            col++;

        } else { //a found it

            return true;

        }

    }

    return false;

}

int main() {

    int matrix[][4] = {

        {1, 4, 7, 11},

        {2, 5, 8, 12},

        {3, 6, 9, 16},

        {10, 13, 14, 17},

        {18, 21, 23, 26}

    };

    int matrixSize = 5;

    int matrixColSize[] = {4, 4, 4, 4, 4};

    int target = 5;

    bool found = searchMatrix((int\*\*)matrix, matrixSize, matrixColSize, target);

    if (found) {

        printf("The target %d is found in the matrix.\n", target);

    } else {

        printf("The target %d is not found in the matrix.\n", target);

    }

    return 0;

}

## Validate Binary Search Tree

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid 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:**

A diagram of a connection

Description automatically generated

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

**Output:** true

**Example 2:**

A diagram of a network

Description automatically generated

**Input:** root = [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.

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

bool isValidBSTHelper(struct TreeNode\* node, int \*min, int \*max) {

    if (node == NULL) return true;

    if ((min != NULL && node->val <= \*min) || (max != NULL && node->val >= \*max)) return false;

    return isValidBSTHelper(node->left,min,&node->val) && isValidBSTHelper(node->right, &node->val, max);

}

bool isValidBST(struct TreeNode\* root){

    return isValidBSTHelper(root,NULL,NULL);

}

int main() {

    // Create a sample binary search tree

    struct TreeNode\* root = createNode(2);

    root->left = createNode(1);

    root->right = createNode(3);

    // Check if the tree is a valid binary search tree

    bool isValid = isValidBST(root);

    // Print the result

    if (isValid) {

        printf("The given tree is a valid binary search tree.\n");

    } else {

        printf("The given tree is not a valid binary search tree.\n");

    }

    // Free the memory allocated for the tree

    freeTree(root);

    return 0;

}

## Plus One

You are given a **large integer** represented as an integer array digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return *the resulting array of digits*.

**Example 1:**

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

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

**Explanation:** The array represents the integer 123.

Incrementing by one gives 123 + 1 = 124.

Thus, the result should be [1,2,4].

**Example 2:**

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

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

**Explanation:** The array represents the integer 4321.

Incrementing by one gives 4321 + 1 = 4322.

Thus, the result should be [4,3,2,2].

**Example 3:**

**Input:** digits = [9]

**Output:** [1,0]

**Explanation:** The array represents the integer 9.

Incrementing by one gives 9 + 1 = 10.

Thus, the result should be [1,0].

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

int\* plusOne(int\* digits, int digitsSize, int\* returnSize) {

    int i = 0;

    int \*return\_arr = NULL;

    for (i = digitsSize-1; i >= 0; i-- ) {

        if (digits[i] == 9) digits[i] = 0;

        else {

            digits[i]+=1;

            \*returnSize = digitsSize;

            return digits;

        }

    }

    return\_arr = (int\*)malloc((digitsSize+1)\*sizeof(int));

    return\_arr[0] = 1;

    for (i = 0; i <= digitsSize-1; i++ ) {

      return\_arr[i+1] = digits[i];

    }

    \*returnSize = digitsSize+1;

    return return\_arr;

}

int main() {

    int digits[] = {1, 2, 3}; // Example array

    int digitsSize = sizeof(digits) / sizeof(digits[0]);

    int returnSize = 0;

    // Call the plusOne function

    int\* result = plusOne(digits, digitsSize, &returnSize);

    // Print the result

    printf("Result after adding one: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    // Free the dynamically allocated memory

    free(result);

    return 0;

}

## Majority Element

Given an array nums of size n, return *the majority element*.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

**Example 1:**

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

**Output:** 3

**Example 2:**

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

**Output:** 2

int majorityElement(int\* nums, int numsSize) {

    int candidate = 0;

    int count = 0;

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

        if (count == 0) candidate = nums[i];

        count+=(nums[i] == candidate) ? 1:-1;

    }

    return candidate;

}

int main() {

    int nums[] = {3, 2, 3}; // Example array

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Call the majorityElement function

    int result = majorityElement(nums, numsSize);

    // Print the result

    printf("Majority Element: %d\n", result);

    return 0;

}

## Valid Palindrome

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true*if it is a****palindrome****, or*false*otherwise*.

**Example 1:**

**Input:** s = "A man, a plan, a canal: Panama"

**Output:** true

**Explanation:** "amanaplanacanalpanama" is a palindrome.

**Example 2:**

**Input:** s = "race a car"

**Output:** false

**Explanation:** "raceacar" is not a palindrome.

**Example 3:**

**Input:** s = " "

**Output:** true

**Explanation:** s is an empty string "" after removing non-alphanumeric characters.

Since an empty string reads the same forward and backward, it is a palindrome.

bool isPalindrome(char\* s) {

    int start = 0;

    int end = strlen(s);

    while (start < end) {

        if (!isalnum(s[start])) {

            start++;

            continue;

        }

        if (!isalnum(s[end])) {

            end--;

            continue;

        }

        if (tolower(s[start]) != tolower(s[end])) return false;

        else {

            start++;

            end--;

        }

    }

    return true;

}

int main() {

    char s[] = "A man, a plan, a canal: Panama"; // Example string

    // Call the isPalindrome function

    bool result = isPalindrome(s);

    // Print the result

    if (result) {

        printf("The string is a palindrome.\n");

    } else {

        printf("The string is not a palindrome.\n");

    }

    return 0;

}

## House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight****without alerting the police***.

**Example 1:**

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

**Output:** 4

**Explanation:** Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

**Example 2:**

**Input:** nums = [2,7,9,3,1]

**Output:** 12

**Explanation:** Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).

Total amount you can rob = 2 + 9 + 1 = 12.

int max(int a, int b) {

    return (a > b) ? a:b;

}

int rob(int\* nums, int numsSize){

    int i = 0;

    if (numsSize == 0) return 0;

    if (numsSize == 1) return nums[0];

    int\* result = malloc(numsSize \* sizeof(int));

    result[0] = nums[0];

    result[1] = max(nums[0],nums[1]);

    for (i = 2; i < numsSize; i++) {

        result[i] = max(result[i-1],result[i-2]+nums[i]);

    }

    return result[numsSize-1];

}

int main() {

    int nums[] = {2, 7, 9, 3, 1};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    int maxProfit = rob(nums, numsSize);

    printf("Maximum profit: %d\n", maxProfit);

    return 0;

}

## Next Greater Element

The **next greater element** of some element x in an array is the **first greater** element that is **to the right** of x in the same array.

You are given two **distinct 0-indexed** integer arrays nums1 and nums2, where nums1 is a subset of nums2.

For each 0 <= i < nums1.length, find the index j such that nums1[i] == nums2[j] and determine the **next greater element** of nums2[j] in nums2. If there is no next greater element, then the answer for this query is -1.

Return *an array*ans*of length*nums1.length*such that*ans[i]*is the****next greater element****as described above.*

**Example 1:**

**Input:** nums1 = [4,1,2], nums2 = [1,3,4,2]

**Output:** [-1,3,-1]

**Explanation:** The next greater element for each value of nums1 is as follows:

- 4 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answer is -1.

- 1 is underlined in nums2 = [1,3,4,2]. The next greater element is 3.

- 2 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answer is -1.

**Example 2:**

**Input:** nums1 = [2,4], nums2 = [1,2,3,4]

**Output:** [3,-1]

**Explanation:** The next greater element for each value of nums1 is as follows:

- 2 is underlined in nums2 = [1,2,3,4]. The next greater element is 3.

- 4 is underlined in nums2 = [1,2,3,4]. There is no next greater element, so the answer is -1.

 public class Solution {

    public int[] nextGreaterElement(int[] nums1, int[] nums2) {

        Stack<Integer> stack = new Stack<>();

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

        for (int i = 0; i < nums2.length; i++) {

            while (!stack.empty() && nums2[i] > stack.peek())

                map.put(stack.pop(), nums2[i]);

            stack.push(nums2[i]);

        }

        while (!stack.empty())

            map.put(stack.pop(), -1);

        int[] res = new int[nums1.length];

        for (int i = 0; i < nums1.length; i++) {

            res[i] = map.get(nums1[i]);

        }

        return res;

    }

}

## Shuffle An Array

Given an integer array nums, design an algorithm to randomly shuffle the array. All permutations of the array should be **equally likely** as a result of the shuffling.

Implement the Solution class:

* Solution(int[] nums) Initializes the object with the integer array nums.
* int[] reset() Resets the array to its original configuration and returns it.
* int[] shuffle() Returns a random shuffling of the array.

**Example 1:**

**Input**

["Solution", "shuffle", "reset", "shuffle"]

[[[1, 2, 3]], [], [], []]

**Output**

[null, [3, 1, 2], [1, 2, 3], [1, 3, 2]]

**Explanation**

Solution solution = new Solution([1, 2, 3]);

solution.shuffle(); // Shuffle the array [1,2,3] and return its result.

// Any permutation of [1,2,3] must be equally likely to be returned.

// Example: return [3, 1, 2]

solution.reset(); // Resets the array back to its original configuration [1,2,3]. Return [1, 2, 3]

solution.shuffle(); // Returns the random shuffling of array [1,2,3]. Example: return [1, 3, 2]

**C:**

#include <stdlib.h>

#include <string.h>

typedef struct {

int\* original;

int\* copy;

int\* temp;

int length;

} Solution;

void swap(int\* arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

Solution\* solutionCreate(int\* nums, int numsSize) {

Solution\* obj = (Solution\*)malloc(sizeof(Solution));

obj->original = (int\*)malloc(sizeof(int) \* numsSize);

obj->copy = (int\*)malloc(sizeof(int) \* numsSize);

obj->temp = (int\*)malloc(sizeof(int) \* numsSize);

obj->length = numsSize;

memcpy(obj->original, nums, sizeof(int) \* numsSize);

memcpy(obj->copy, nums, sizeof(int) \* numsSize);

return obj;

}

int\* solutionReset(Solution\* obj, int\* returnSize) {

memcpy(obj->copy, obj->original, sizeof(int) \* obj->length);

\*returnSize = obj->length;

return obj->original;

}

int\* solutionShuffle(Solution\* obj, int\* returnSize) {

memcpy(obj->temp, obj->copy, sizeof(int) \* obj->length);

for (int i = 0; i < obj->length; i++) {

int ra = obj->length - i;

int j = i + rand() % ra;

swap(obj->temp, i, j);

}

\*returnSize = obj->length;

return obj->temp;

}

void solutionFree(Solution\* obj) {

free(obj->original);

free(obj->copy);

free(obj->temp);

free(obj);

}

int main() {

    // Example usage

    int nums[] = {1, 2, 3, 4, 5};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Create the Solution object

    Solution\* obj = solutionCreate(nums, numsSize);

    // Reset the array

    int returnSize;

    int\* resetArray = solutionReset(obj, &returnSize);

    printf("Reset array: ");

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

        printf("%d ", resetArray[i]);

    }

    printf("\n");

    // Shuffle the array

    int\* shuffledArray = solutionShuffle(obj, &returnSize);

    printf("Shuffled array: ");

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

        printf("%d ", shuffledArray[i]);

    }

    printf("\n");

    // Free the Solution object

    solutionFree(obj);

    return 0;

}

## MinStack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

* MinStack() initializes the stack object.
* void push(int val) pushes the element val onto the stack.
* void pop() removes the element on the top of the stack.
* int top() gets the top element of the stack.
* int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

**Example 1:**

**Input**

["MinStack","push","push","push","getMin","pop","top","getMin"]

[[],[-2],[0],[-3],[],[],[],[]]

**Output**

[null,null,null,null,-3,null,0,-2]

**Explanation**

MinStack minStack = new MinStack();

minStack.push(-2);

minStack.push(0);

minStack.push(-3);

minStack.getMin(); // return -3

minStack.pop();

minStack.top(); // return 0

minStack.getMin(); // return -2

**C:**

typedef struct {

int\* stack;

int\* minStack;

int top;

int minTop;

int capacity;

} MinStack;

MinStack\* minStackCreate() {

MinStack\* stack = (MinStack\*)malloc(sizeof(MinStack));

stack->stack = (int\*)malloc(sizeof(int));

stack->minStack = (int\*)malloc(sizeof(int));

stack->top = -1;

stack->minTop = -1;

stack->capacity = 1;

return stack;

}

void minStackPush(MinStack\* obj, int val) {

if (obj->top == obj->capacity - 1) {

obj->capacity \*= 2;

obj->stack = (int\*)realloc(obj->stack, sizeof(int) \* obj->capacity);

}

obj->stack[++obj->top] = val;

if (obj->minTop == -1 || val <= obj->minStack[obj->minTop]) {

if (obj->minTop == obj->capacity - 1) {

obj->minStack = (int\*)realloc(obj->minStack, sizeof(int) \* obj->capacity);

}

obj->minStack[++obj->minTop] = val;

}

}

void minStackPop(MinStack\* obj) {

if (obj->stack[obj->top] == obj->minStack[obj->minTop]) {

obj->minTop--;

}

obj->top--;

}

int minStackTop(MinStack\* obj) {

return obj->stack[obj->top];

}

int minStackGetMin(MinStack\* obj) {

return obj->minStack[obj->minTop];

}

void minStackFree(MinStack\* obj) {

free(obj->stack);

free(obj->minStack);

free(obj);

}

int main() {

    // Example usage

    MinStack\* obj = minStackCreate();

    // Push elements onto the stack

    minStackPush(obj, -2);

    minStackPush(obj, 0);

    minStackPush(obj, -3);

    // Print the minimum element in the stack

    printf("Minimum element: %d\n", minStackGetMin(obj)); // Output: -3

    // Pop an element from the stack

    minStackPop(obj);

    // Print the top element of the stack

    printf("Top element: %d\n", minStackTop(obj)); // Output: 0

    // Print the minimum element in the stack

    printf("Minimum element: %d\n", minStackGetMin(obj)); // Output: -2

    // Free the memory allocated for the stack

    minStackFree(obj);

    return 0;

}

## Sum Of Two Integers

Given two integers a and b, return *the sum of the two integers without using the operators* + *and* -.

**Example 1:**

**Input:** a = 1, b = 2

**Output:** 3

**Example 2:**

**Input:** a = 2, b = 3

**Output:** 5

int getSum(int a, int b){

    while (b != 0) {

        unsigned int sum = a ^ b;

        unsigned int carry = (unsigned int)(a & b) << 1;

        a = sum;

        b = carry;

    }

    return a;

}

int main() {

    // Example usage of getSum function

    int a = 5;

    int b = 7;

    printf("Sum of %d and %d is: %d\n", a, b, getSum(a, b)); // Output: Sum of 5 and 7 is: 12

    return 0;

}

## Maximum Product SubArray

Given an integer array nums, find a

subarray

 that has the largest product, and return the product.

The test cases are generated so that the answer will fit in a **32-bit** integer.

**Example 1:**

**Input:** nums = [2,3,-2,4]

**Output:** 6

**Explanation:** [2,3] has the largest product 6.

**Example 2:**

**Input:** nums = [-2,0,-1]

**Output:** 0

**Explanation:** The result cannot be 2, because [-2,-1] is not a subarray.

class Solution {

    public int maxProduct(int[] nums) {

        int current\_array = nums[0];

        int maximum\_array = nums[0];

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

            current\_array = Math.max(current\_array\*nums[i],nums[i]);

            maximum\_array = Math.max(current\_array, maximum\_array);

        }

        return maximum\_array;

    }}

int max(int a, int b) {

    return a > b ? a : b;

}

int maxProduct(int\* nums, int numsSize) {

    int current\_array = nums[0];

    int maximum\_array = nums[0];

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

        current\_array = max(current\_array \* nums[i], nums[i]);

        maximum\_array = max(current\_array, maximum\_array);

    }

    return maximum\_array;

}

int main() {

    int nums[] = {2, 3, -2, 4};

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    printf("Maximum product: %d\n", maxProduct(nums, numsSize)); // Output: 6

    return 0;

}

## Number Of 1 Bits

Write a function that takes the binary representation of an unsigned integer and returns the number of '1' bits it has (also known as the [Hamming weight](http://en.wikipedia.org/wiki/Hamming_weight)).

**Note:**

* Note that in some languages, such as Java, there is no unsigned integer type. In this case, the input will be given as a signed integer type. It should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
* In Java, the compiler represents the signed integers using [2's complement notation](https://en.wikipedia.org/wiki/Two%27s_complement). Therefore, in **Example 3**, the input represents the signed integer. -3.

**Example 1:**

**Input:** n = 00000000000000000000000000001011

**Output:** 3

**Explanation:** The input binary string **00000000000000000000000000001011** has a total of three '1' bits.

**Example 2:**

**Input:** n = 00000000000000000000000010000000

**Output:** 1

**Explanation:** The input binary string **00000000000000000000000010000000** has a total of one '1' bit.

**Example 3:**

**Input:** n = 11111111111111111111111111111101

**Output:** 31

**Explanation:** The input binary string **11111111111111111111111111111101** has a total of thirty one '1' bits.

int hammingWeight(uint32\_t n) {

    int count = 0;

    while (n != 0) {

        n = (n & n-1);

        count++;

    }

    return count;

}

int main() {

    uint32\_t n = 0b00000000000000000000000000001011; // Example input: 11

    printf("Number of set bits: %d\n", hammingWeight(n)); // Output: 3

    return 0;

}

## Counting Bits

Given an integer n, return *an array*ans*of length*n + 1*such that for each*i(0 <= i <= n)*,*ans[i]*is the****number of***1***'s****in the binary representation of*i.

**Example 1:**

**Input:** n = 2

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

**Explanation:**

0 --> 0

1 --> 1

2 --> 10

**Example 2:**

**Input:** n = 5

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

**Explanation:**

0 --> 0

1 --> 1

2 --> 10

3 --> 11

4 --> 100

5 --> 101

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

int countOnes(int num) {

    int count = 0;

    while (num) {

        num=  num & (num -1);

        //num >>= 1;

        count++;

    }

    return count;

}

int\* countBits(int n, int\* returnSize) {

    // Allocate memory for the result array

    int\* result = (int\*)malloc((n + 1) \* sizeof(int));

    // Generate count of set bits for each number from 0 to n

    for (int i = 0; i <= n; ++i) {

        result[i] = countOnes(i);

    }

    // Set the return size

    \*returnSize = n + 1;

    return result;

}

int main() {

    int n = 5; // Example value of n

    int returnSize;

    int\* result = countBits(n, &returnSize);

    // Print the result

    printf("Result: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    // Free the allocated memory

    free(result);

    return 0;

}

## Missing Number

Given an array nums containing n distinct numbers in the range [0, n], return *the only number in the range that is missing from the array.*

**Example 1:**

**Input:** nums = [3,0,1]

**Output:** 2

**Explanation:** n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

**Example 2:**

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

**Output:** 2

**Explanation:** n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.

**Example 3:**

**Input:** nums = [9,6,4,2,3,5,7,0,1]

**Output:** 8

**Explanation:** n = 9 since there are 9 numbers, so all numbers are in the range [0,9]. 8 is the missing number in the range since it does not appear in nums.

int missingNumber(int\* nums, int numsSize){

    int sum = 0;

    int actual\_sum = 0;

    int i = 0;

    for (i = 0; i < numsSize; i++) {

        sum += nums[i];

    }

    actual\_sum = (numsSize\*(numsSize+1))/2;

    return actual\_sum - sum;

}

int main() {

    // Example input array

    int nums[] = {3, 0, 1, 4, 6, 2}; // Example array

    int numsSize = sizeof(nums) / sizeof(nums[0]);

    // Find the missing number

    int missing = missingNumber(nums, numsSize);

    // Print the missing number

    printf("The missing number is: %d\n", missing);

    return 0;

}

## Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

**Note:**

* Note that in some languages, such as Java, there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
* In Java, the compiler represents the signed integers using [2's complement notation](https://en.wikipedia.org/wiki/Two%27s_complement). Therefore, in **Example 2** above, the input represents the signed integer -3 and the output represents the signed integer -1073741825.

**Example 1:**

**Input:** n = 00000010100101000001111010011100

**Output:** 964176192 (00111001011110000010100101000000)

**Explanation:** The input binary string **00000010100101000001111010011100** represents the unsigned integer 43261596, so return 964176192 which its binary representation is **00111001011110000010100101000000**.

**Example 2:**

**Input:** n = 11111111111111111111111111111101

**Output:** 3221225471 (10111111111111111111111111111111)

**Explanation:** The input binary string **11111111111111111111111111111101** represents the unsigned integer 4294967293, so return 3221225471 which its binary representation is **10111111111111111111111111111111**.

uint32\_t reverseBits(uint32\_t n) {

    int result = 0;

    for (int i = 31; i >=0; i--) {

        result |= (((n >> (31-i)) & 0x1) << i);

    }

    return result;

}

int main() {

    // Example input

    uint32\_t n = 43261596; // Binary: 00000010100101000001111010011100

    // Reverse the bits

    uint32\_t reversed = reverseBits(n);

    // Print the result

    printf("Reversed bits: %u\n", reversed);

    return 0;

}

## Detect Cycle in Linked List

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.

Return true*if there is a cycle in the linked list*. Otherwise, return false.

**Example 1:**

A diagram of a number and two circles

Description automatically generated

**Input:** head = [3,2,0,-4], pos = 1

**Output:** true

**Explanation:** There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

**Example 2:**

A diagram of a diagram

Description automatically generated

**Input:** head = [1,2], pos = 0

**Output:** true

**Explanation:** There is a cycle in the linked list, where the tail connects to the 0th node.

**Example 3:**

A number in a circle

Description automatically generated

**Input:** head = [1], pos = -1

**Output:** false

**Explanation:** There is no cycle in the linked list.

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     struct ListNode \*next;

 \* };

 \*/

bool hasCycle(struct ListNode \*head) {

    if (head == NULL) return false;

    struct ListNode \*slow = head;

    struct ListNode \*fast = head;

    while (fast != NULL && fast->next != NULL) {

        slow = slow->next;

        fast = fast->next->next;

        if (slow == fast) return true;

    }

    return false;

}

int main() {

    // Create a linked list with a cycle

    struct ListNode \*head = (struct ListNode \*)malloc(sizeof(struct ListNode));

    head->val = 3;

    head->next = (struct ListNode \*)malloc(sizeof(struct ListNode));

    head->next->val = 2;

    head->next->next = (struct ListNode \*)malloc(sizeof(struct ListNode));

    head->next->next->val = 0;

    head->next->next->next = (struct ListNode \*)malloc(sizeof(struct ListNode));

    head->next->next->next->val = -4;

    head->next->next->next->next = head->next; // Create a cycle

    // Check if the linked list has a cycle

    bool has\_cycle = hasCycle(head);

    // Print the result

    if (has\_cycle) {

        printf("The linked list has a cycle.\n");

    } else {

        printf("The linked list does not have a cycle.\n");

    }

    // Free the memory allocated for the linked list

    free(head->next->next->next);

    free(head->next->next);

    free(head->next);

    free(head);

    return 0;

}

## Reorder List

You are given the head of a singly linked-list. The list can be represented as:

L0 → L1 → … → Ln - 1 → Ln

*Reorder the list to be on the following form:*

L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …

You may not modify the values in the list's nodes. Only nodes themselves may be changed.

**Example 1:**

A diagram of a diagram

Description automatically generated

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

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

**Example 2:**

A diagram of a diagram

Description automatically generated

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

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

**Constraints:**

void reorderList(struct ListNode\* head){

struct ListNode \*stack[50000];

struct ListNode \*temp=head;

int i=0;

while(temp)

{

    stack[i]=temp;

    temp=temp->next;

    i++;

}

temp=head;

struct ListNode \*sec=NULL;

int k=i;

for(int j=0;j<i/2;j++)

{

 sec=temp->next;

 temp->next=stack[--k];

 temp->next->next=sec;

 temp=sec;

}

temp->next=NULL;

}

int main() {

    // Create a sample linked list

    struct ListNode\* head = (struct ListNode\*)malloc(sizeof(struct ListNode));

    head->val = 1;

    head->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    head->next->val = 2;

    head->next->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    head->next->next->val = 3;

    head->next->next->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

    head->next->next->next->val = 4;

    head->next->next->next->next = NULL;

       reorderList(head);

    // Print the reordered list

    printf("Reordered list: ");

    struct ListNode\* temp = head;

    while (temp != NULL) {

        printf("%d ", temp->val);

        temp = temp->next;

    }

    printf("\n");

    // Free the memory allocated for the linked list

    while (head != NULL) {

        struct ListNode\* temp = head;

        head = head->next;

        free(temp);

    }

    return 0;

}

class Solution {

public void reorderList(ListNode head) {

if (head == null) return;

// find the middle of linked list [Problem 876]

// in 1->2->3->4->5->6 find 4

ListNode slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

// reverse the second part of the list [Problem 206]

// convert 1->2->3->4->5->6 into 1->2->3->4 and 6->5->4

// reverse the second half in-place

ListNode prev = null, curr = slow, tmp;

while (curr != null) {

tmp = curr.next;

curr.next = prev;

prev = curr;

curr = tmp;

}

// merge two sorted linked lists [Problem 21]

// merge 1->2->3->4 and 6->5->4 into 1->6->2->5->3->4

ListNode first = head, second = prev;

while (second.next != null) {

tmp = first.next;

first.next = second;

first = tmp;

tmp = second.next;

second.next = first;

second = tmp;

}

}

}

## Set Matrix Zeros

Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's.

You must do it [in place](https://en.wikipedia.org/wiki/In-place_algorithm).

**Example 1:**

A white background with black arrows and numbers

Description automatically generated

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

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

**Example 2:**

A number and arrow with numbers

Description automatically generated with medium confidence

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

**Output:** [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

void setZeroes(int\*\* matrix, int matrixSize, int\* matrixColSize) {

    // Variables to track whether the first row and first column contain zeros

    bool first\_row = false;

    bool first\_col = false;

    // Check if the first column contains zeros

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

        if (matrix[i][0] == 0) {

            first\_col = true;

            break;

        }

    }

    // Check if the first row contains zeros

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

        if (matrix[0][i] == 0) {

            first\_row = true;

            break;

        }

    }

    // Mark the first element of each row and column if the element is zero

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

        for (int j = 1; j < \*matrixColSize; j++) {

            if (matrix[i][j] == 0) {

                matrix[i][0] = 0;

                matrix[0][j] = 0;

            }

        }

    }

    // Set the entire row to zero if the first element of the row is zero

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

        if (matrix[i][0] == 0) {

            for (int j = 1; j < \*matrixColSize; j++) {

                matrix[i][j] = 0;

            }

        }

    }

    // Set the entire column to zero if the first element of the column is zero

    for (int j = 1; j < \*matrixColSize; j++) {

        if (matrix[0][j] == 0) {

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

                matrix[i][j] = 0;

            }

        }

    }

    // Set the first row to zero if it originally contained a zero

    if (first\_row) {

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

            matrix[0][j] = 0;

        }

    }

    // Set the first column to zero if it originally contained a zero

    if (first\_col) {

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

            matrix[i][0] = 0;

        }

    }

}

int main() {

    // Example matrix

    int numRows = 3;

    int numCols = 4;

    int \*\*matrix = (int\*\*)malloc(numRows \* sizeof(int\*));

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

        matrix[i] = (int\*)malloc(numCols \* sizeof(int));

    }

    // Initialize the matrix with sample values

    matrix[0][0] = 1;

    matrix[0][1] = 1;

    matrix[0][2] = 1;

    matrix[0][3] = 1;

    matrix[1][0] = 1;

    matrix[1][1] = 0;

    matrix[1][2] = 1;

    matrix[1][3] = 1;

    matrix[2][0] = 1;

    matrix[2][1] = 1;

    matrix[2][2] = 1;

    matrix[2][3] = 1;

    // Printing the original matrix

    printf("Original Matrix:\n");

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

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

            printf("%d ", matrix[i][j]);

        }

        printf("\n");

    }

    // Call the setZeroes function

    setZeroes(matrix, numRows, &numCols);

    // Printing the modified matrix

    printf("\nMatrix After Setting Zeroes:\n");

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

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

            printf("%d ", matrix[i][j]);

        }

        printf("\n");

    }

    // Free memory allocated for the matrix

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

        free(matrix[i]);

    }

    free(matrix);

}

/\*

0 1 2 0    0 0 0 0   0 0 0 0

3 4 5 2    0 4 5 0   3 4 5 0

1 3 1 5    0 3 1 0   1 3 1 0

0 0 0 5    0 0 0 0   0 0 0 0

4 3 1 4    0 0 0 4   0 0 0 4

0 1 1 4    0 0 0 0   0 0 0 4

1 2 1 3    0 0 0 3   0 0 0 3

0 0 1 1    0 0 0 0   0 0 0 0

\*/

## Insert Intervals

You are given an array of non-overlapping intervals intervals where intervals[i] = [starti, endi] represent the start and the end of the ith interval and intervals is sorted in ascending order by starti. You are also given an interval newInterval = [start, end] that represents the start and end of another interval.

Insert newInterval into intervals such that intervals is still sorted in ascending order by starti and intervals still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return intervals*after the insertion*.

**Example 1:**

**Input:** intervals = [[1,3],[6,9]], newInterval = [2,5]

**Output:** [[1,5],[6,9]]

**Example 2:**

**Input:** intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]

**Output:** [[1,2],[3,10],[12,16]]

**Explanation:** Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

/\*

 \* 57. Insert Interval

 \*

 \* You are given an array of non-overlapping intervals intervals

 \* where intervals[i] = [starti, endi] represent the start and the

 \* end of the ith interval and intervals is sorted in ascending

 \* order by starti. You are also given an interval newInterval =

 \* [start, end] that represents the start and end of another interval.

 \*

 \* Insert newInterval into intervals such that intervals is still sorted

 \* in ascending order by starti and intervals still does not have any

 \* overlapping intervals (merge overlapping intervals if necessary).

 \*

 \* Return intervals after the insertion.

 \*

 \* 0 <= intervals.length <= 104

 \* intervals[i].length == 2

 \* 0 <= starti <= endi <= 105

 \* intervals is sorted by starti in ascending order.

 \* newInterval.length == 2

 \* 0 <= start <= end <= 10^5

 \*/

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

#define MAX(a, b) ((a) > (b) ? (a) : (b))

int cmp(const void \*a, const void \*b)

{

    return (\*((int \*\*)a))[0] - (\*((int \*\*)b))[0];

}

int\*\* insert(int\*\* intervals, int intervalsSize, int\* intervalsColSize, int\* newInterval, int newIntervalSize, int\* returnSize, int\*\* returnColumnSizes){

    /\*

     \* Input:

     \*  intervals,

     \*  intervalsSize

     \*  intervalsColSize

     \*  newInterval

     \*  newIntervalSize

     \*/

    int seg\_start = newInterval[0], seg\_end = newInterval[1];

    int \*\*ans = (int \*\*)malloc(sizeof(int \*) \* (intervalsSize + 1));

    int \*\*tmp = (int \*\*)malloc(sizeof(int \*) \* (intervalsSize + 1));

    (\*returnColumnSizes) = (int \*)malloc(sizeof(int) \* (intervalsSize + 1));

    \*returnSize = 0;

    /\* Copy intervals to tmp buffer \*/

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

        tmp[i] = (int \*)malloc(sizeof(int) \* 2);

        tmp[i][0] = intervals[i][0];

        tmp[i][1] = intervals[i][1];

    }

    /\* Append newInterval to tmp buffer \*/

    tmp[intervalsSize] = (int \*)malloc(sizeof(int) \* 2);

    ans[intervalsSize] = (int \*)malloc(sizeof(int) \* 2);

    tmp[intervalsSize][0] = newInterval[0];

    tmp[intervalsSize][1] = newInterval[1];

    /\* Sort by start \*/

    qsort(tmp, intervalsSize + 1, sizeof(int \*), cmp);

    seg\_start = tmp[0][0];

    seg\_end = tmp[0][1];

    /\* Merge the interval \*/

    for (int i = 0; i < intervalsSize + 1; i++) {

        /\* Check if segements can be merged, and store the start and end position of merged segment \*/

        if (tmp[i][0] >= seg\_start && tmp[i][0] <= seg\_end) {

            seg\_end = MAX(seg\_end, tmp[i][1]);

        }

        /\* Complete current segment \*/

        else {

            ans[\*returnSize] = (int \*)malloc(sizeof(int) \* 2);

            ans[\*returnSize][0] = seg\_start;

            ans[\*returnSize][1] = seg\_end;

            (\*returnColumnSizes)[\*returnSize] = 2;

            \*returnSize += 1;

            seg\_start = tmp[i][0];

            seg\_end = tmp[i][1];

        }

        /\* Add last one segment \*/

        if (i == intervalsSize) {

            ans[\*returnSize] = (int \*)malloc(sizeof(int) \* 2);

            ans[\*returnSize][0] = seg\_start;

            ans[\*returnSize][1] = seg\_end;

            (\*returnColumnSizes)[\*returnSize] = 2;

            \*returnSize += 1;

        }

    }

    for (int i = 0; i < intervalsSize + 1; i++) {

        free(tmp[i]);

    }

    free(tmp);

    /\*

     \* Output:

     \*  returnSize

     \*  returnColumnSizes

     \*/

    return ans;

}

int main() {

    // Example input intervals

    int intervalsSize = 3;

    int intervalsColSize[] = {2, 2, 2};

    int\* intervals[] = {{1, 3}, {6, 9}, {12, 17}};

    // Example new interval

    int newInterval[] = {4, 10};

    int newIntervalSize = 2;

    // Variables to store output

    int returnSize = 0;

    int\* returnColumnSizes = NULL;

    int\*\* result = insert(intervals, intervalsSize, intervalsColSize, newInterval, newIntervalSize, &returnSize, &returnColumnSizes);

    // Print input intervals

    printf("Input intervals: ");

    printIntervals(intervals, intervalsSize, intervalsColSize);

    // Print new interval

    printf("New interval: [%d, %d]\n", newInterval[0], newInterval[1]);

    // Print merged intervals

    printf("Merged intervals: ");

    printIntervals(result, returnSize, returnColumnSizes);

    // Free memory

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

        free(result[i]);

    }

    free(result);

    free(returnColumnSizes);

    return 0;

}

## Non Overlapping Intervals

Given an array of intervals intervals where intervals[i] = [starti, endi], return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

**Example 1:**

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

**Output:** 1

**Explanation:** [1,3] can be removed and the rest of the intervals are non-overlapping.

**Example 2:**

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

**Output:** 2

**Explanation:** You need to remove two [1,2] to make the rest of the intervals non-overlapping.

**Example 3:**

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

**Output:** 0

**Explanation:** You don't need to remove any of the intervals since they're already non-overlapping.

// Interval structure definition

struct Interval {

    int start;

    int end;

};

// Function to compare intervals for sorting

int compareIntervals(const void\* a, const void\* b) {

    return ((struct Interval\*)a)->end - ((struct Interval\*)b)->end;

}

// Function to find the minimum number of intervals to remove

int eraseOverlapIntervals(int\*\* intervals, int intervalsSize, int\* intervalsColSize) {

    if (intervals == NULL || intervalsSize <= 1) {

        return 0;

    }

    // Convert 2D array to array of Interval structures

    struct Interval\* intervalsArr = (struct Interval\*)malloc(intervalsSize \* sizeof(struct Interval));

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

        intervalsArr[i].start = intervals[i][0];

        intervalsArr[i].end = intervals[i][1];

    }

    // Sort intervals by end time in ascending order

    qsort(intervalsArr, intervalsSize, sizeof(struct Interval), compareIntervals);

    int ans = 0;

    int k = INT\_MIN;

    // Iterate through intervals

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

        int x = intervalsArr[i].start;

        int y = intervalsArr[i].end;

        if (x >= k) {

            // Case 1

            k = y;

        } else {

            // Case 2

            ans++;

        }

    }

    // Free allocated memory

    free(intervalsArr);

    return ans;

}

int main() {

    // Example intervals

    int intervalsSize = 4;

    int intervalsColSize[] = {2, 2, 2, 2};

    int\* intervals[] = {{1, 2}, {2, 3}, {3, 4}, {1, 3}};

    // Find the minimum number of intervals to remove

    int minToRemove = eraseOverlapIntervals(intervals, intervalsSize, intervalsColSize);

    // Print the result

    printf("Minimum number of intervals to remove: %d\n", minToRemove);

    return 0;

}

## Meeting Rooms (Can attend all meetings)

Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.

**Example 1:**

**Input:** intervals = [[0,30],[5,10],[15,20]]

**Output:** false

**Example 2:**

**Input:** intervals = [[7,10],[2,4]]

**Output:** true

int compareIntervals(const void\* a, const void\* b) {

    return (\*((int\*\*)a))[0] - (\*((int\*\*)b))[0];

}

int canAttendMeetings(int\*\* intervals, int intervalsSize, int\* intervalsColSize) {

    if (intervalsSize <= 1) {

        return 1;  // A person can attend all meetings if there is only one or zero meetings.

    }

    // Sort intervals based on the start time.

    qsort(intervals, intervalsSize, sizeof(int\*), compareIntervals);

    // Check if there is any overlap between consecutive intervals.

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

        if (intervals[i - 1][1] > intervals[i][0]) {

            return 0;  // There is an overlap, so the person cannot attend all meetings.

        }

    }

    return 1;  // No overlap found, person can attend all meetings.

}

int main() {

    // Example intervals

    int intervalsSize = 3;

    int intervalsColSize[] = {2, 2, 2};

    int\* intervals[] = {{0, 5}, {5, 10}, {15, 20}};

    // Check if a person can attend all meetings

    int canAttendAllMeetings = canAttendMeetings(intervals, intervalsSize, intervalsColSize);

    // Print the result

    if (canAttendAllMeetings) {

        printf("The person can attend all meetings without overlapping.\n");

    } else {

        printf("The person cannot attend all meetings without overlapping.\n");

    }

    return 0;

}

## Longest Repeating Character Replacement

You are given a string s and an integer k. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most k times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations*.

**Example 1:**

**Input:** s = "ABAB", k = 2

**Output:** 4

**Explanation:** Replace the two 'A's with two 'B's or vice versa.

**Example 2:**

**Input:** s = "AABABBA", k = 1

**Output:** 4

**Explanation:** Replace the one 'A' in the middle with 'B' and form "AABBBBA".

The substring "BBBB" has the longest repeating letters, which is 4.

There may exists other ways to achieve this answer too.

class Solution {

public int characterReplacement(String s, int k) {

// Initialising an empty array to store the count of the

// characters in the given string s

int[] arr = new int[26];

int res = 0;

int max = 0;

// The left pointer for the sliding window is l AND r is the

// right pointer

int l = 0;

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

// Counting the number of each character in the string s

arr[s.charAt(r) - 'A']++;

// Checking the character with max number of occurrence

max = Math.max(max, arr[s.charAt(r) - 'A']);

// Now we check if our current window is valid or not

if (r - l + 1 - max > k) {

// this means the no. of replacements is more than

// allowed (k)

// Decrementing the count of the character which was

// at l because it is no longer in the window

arr[s.charAt(l) - 'A']--;

l++;

}

// The max our window can be

res = Math.max(res, r - l + 1);

}

return res;

}

}

int characterReplacement(char \*s, int k) {

    int n = strlen(s);

    int max\_count = 0;  // Maximum repeating character count in the window

    int start = 0;      // Start index of the window

    int max\_length = 0; // Maximum length of the substring with repeating characters

    // Array to store the count of each character in the window

    int char\_count[26] = {0};

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

        char\_count[s[end] - 'A']++;

        max\_count = (max\_count > char\_count[s[end] - 'A']) ? max\_count : char\_count[s[end] - 'A'];

        // If the window size exceeds the maximum count + k, shrink the window

        if ((end - start + 1) - max\_count > k) {

            char\_count[s[start] - 'A']--;

            start++;

        }

        // Update the maximum length

        max\_length = (max\_length > (end - start + 1)) ? max\_length : (end - start + 1);

    }

    return max\_length;

}

int main() {

    char s[] = "ABAB";

    int k = 2;

    int maxLength = characterReplacement(s, k);

    printf("The maximum length of the substring with repeating characters after replacement is: %d\n", maxLength);

    return 0;

}

## Minimum Window SubString

Given two strings s and t of lengths m and n respectively, return *the****minimum window***

***substring***

*of*s*such that every character in*t*(****including duplicates****) is included in the window*. If there is no such substring, return *the empty string*"".

The testcases will be generated such that the answer is **unique**.

**Example 1:**

**Input:** s = "ADOBECODEBANC", t = "ABC"

**Output:** "BANC"

**Explanation:** The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.

**Example 2:**

**Input:** s = "a", t = "a"

**Output:** "a"

**Explanation:** The entire string s is the minimum window.

**Example 3:**

**Input:** s = "a", t = "aa"

**Output:** ""

**Explanation:** Both 'a's from t must be included in the window.

Since the largest window of s only has one 'a', return empty string.

class Solution {

public String minWindow(String s, String t)

{

if(s.length() == 0 || t.length() == 0)

return "";

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

int n = s.length(),m=t.length();

for(int i=0; i<m;i++)

map.put(t.charAt(i) , map.getOrDefault(t.charAt(i), 0)+1);

int count = map.size();

int start =0,end =0,min =Integer.MAX\_VALUE;

String substring = "";

while(end < n)

{

char ch = s.charAt(end);

if(map.containsKey(ch))

{

map.put(ch , map.get(ch) -1);

if(map.get(ch) == 0)

count--;

}

if(count > 0)

end++;

else if(count == 0)

{

while(count == 0)

{

if(end-start +1 < min)

{

min = end-start+1;

substring = s.substring(start,end+1);

}

char temp = s.charAt(start);

if(map.containsKey(temp))

{

map.put(temp , map.get(temp)+1);

if(map.get(temp) == 1)

count++;

}

start++;

}

end++;

}

}

return substring;

}

}

char\* minWindow(char\* s, char\* t) {

    int sLen = strlen(s);

    int tLen = strlen(t);

    // Initialize arrays to store character frequencies

    int sFreq[128] = {0};

    int tFreq[128] = {0};

    // Count frequencies of characters in string t

    for (int i = 0; i < tLen; i++) {

        tFreq[t[i]]++;

    }

    int left = 0; // Left pointer of the sliding window

    int right = 0; // Right pointer of the sliding window

    int minLen = sLen + 1; // Initialize to a value greater than sLen

    int minStart = 0; // Start index of the minimum window

    int requiredChars = tLen; // Number of characters still required to match t

    while (right < sLen) {

        // Expand the window to the right

        if (tFreq[s[right]] > 0) {

            // This character is required

            requiredChars--;

        }

        tFreq[s[right]]--;

        right++;

        // Check if all characters from t are included in the window

        while (requiredChars == 0) {

            // Update the minimum window

            if (right - left < minLen) {

                minLen = right - left;

                minStart = left;

            }

            // Shrink the window from the left

            tFreq[s[left]]++;

            if (tFreq[s[left]] > 0) {

                // This character is no longer covered

                requiredChars++;

            }

            left++;

        }

    }

    // Check if a valid window was found

    if (minLen == sLen + 1) {

        return "";

    }

    // Allocate memory for the result and copy the substring

    char\* result = malloc((minLen + 1) \* sizeof(char));

    strncpy(result, s + minStart, minLen);

    result[minLen] = '\0';

    return result;

}

int main() {

    char s[] = "ADOBECODEBANC";

    char t[] = "ABC";

    char\* minWindowStr = minWindow(s, t);

    printf("Minimum window substring: %s\n", minWindowStr);

    free(minWindowStr); // Free allocated memory

    return 0;

}

## Palindromic Substrings

Given a string s, return *the number of****palindromic substrings****in it*.

A string is a **palindrome** when it reads the same backward as forward.

A **substring** is a contiguous sequence of characters within the string.

**Example 1:**

**Input:** s = "abc"

**Output:** 3

**Explanation:** Three palindromic strings: "a", "b", "c".

**Example 2:**

**Input:** s = "aaa"

**Output:** 6

**Explanation:** Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

   // Function to expand around the center for odd length palindromes

    int expandAroundCenterOdd(int left, int right, int n, char \*s) {

        int cnt = 0;

        while (left >= 0 && right < n && s[left] == s[right]) {

            cnt++;

            left--;

            right++;

        }

        return cnt;

    }

    // Function to expand around the center for even length palindromes

    int expandAroundCenterEven(int left, int right, int n, char \*s) {

        int cnt = 0;

        while (left >= 0 && right < n && s[left] == s[right]) {

            cnt++;

            left--;

            right++;

        }

        return cnt;

    }

int countSubstrings(char \*s) {

    int n = strlen(s);

    int count = 0;

    for (int i = 0; i < n; i++) {

        // Expand around the center for odd length palindromes

        count += expandAroundCenterOdd(i, i, n, s);

        // Expand around the center for even length palindromes

        count += expandAroundCenterEven(i, i + 1, n, s);

    }

    return count;

}

int main() {

    char s[] = "abc";

    int numSubstrings = countSubstrings(s);

    printf("Number of palindromic substrings: %d\n", numSubstrings);

    return 0;

}

## Encode And Decode Strings

Design an algorithm to encode **a list of strings** to **a string**. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

string encode(vector<string> strs) {

// ... your code

return encoded\_string;

}

Machine 2 (receiver) has the function:

vector<string> decode(string s) {

//... your code

return strs;

}

So Machine 1 does:

string encoded\_string = encode(strs);

and Machine 2 does:

vector<string> strs2 = decode(encoded\_string);

strs2 in Machine 2 should be the same as strs in Machine 1.

Implement the encode and decode methods.

You are not allowed to solve the problem using any serialize methods (such as eval).

**Example 1:**

**Input:** dummy\_input = ["Hello","World"]

**Output:** ["Hello","World"]

**Explanation:**

Machine 1:

Codec encoder = new Codec();

String msg = encoder.encode(strs);

Machine 1 ---msg---> Machine 2

Machine 2:

Codec decoder = new Codec();

String[] strs = decoder.decode(msg);

**Example 2:**

**Input:** dummy\_input = [""]

**Output:** [""]

/\*\* Encodes a list of strings to a single string \*/

char\* encode(char\*\* strs, int strsSize) {

    char \*ret = malloc(strsSize \* 200);

    char \*ret\_start = ret;

    /\* Build string in the following format:

       <string\_size>#<string> \*/

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

        ret += sprintf(ret, "%d#%s", (int) strlen(strs[i]), strs[i]);

    }

    printf("%s",ret\_start);

    return ret\_start;

}

/\*\*

 \* Decodes a single string to a list of strings.

 \*

 \* Return an array of size \*returnSize.

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

char\*\* decode(char\* s, int\* returnSize) {

    int i = 0;

    int j;

    char \*\*ret = malloc(sizeof (char \*) \* 200);

    char len[4];

    int curr\_char, str\_len;

    int curr\_str = 0;

    while (s[i] != '\0') {

        /\* Find the length of this string \*/

        j = 0;

        memset(len, 0, sizeof(len));

        while (s[i] != '#') {

            len[j++] = s[i++];

        }

        str\_len = atoi(len);

        i++;

        /\* Extract the string \*/

        ret[curr\_str] = malloc(str\_len + 1);

        curr\_char = 0;

        while (str\_len > 0) {

            ret[curr\_str][curr\_char++] = s[i++];

            str\_len--;

        }

        ret[curr\_str][curr\_char] = '\0';

        curr\_str++;

    }

    \*returnSize = curr\_str;

    return ret;

}

int main() {

    char\* strs[] = {"hello", "world", "foo", "bar"};

    int strsSize = 4;

    // Encode the list of strings

    char\* encoded = encode(strs, strsSize);

    printf("\nEncoded string: %s\n", encoded);

    // Decode the encoded string

    int returnSize;

    char\*\* decoded = decode(encoded, &returnSize);

    // Print the decoded strings

    printf("Decoded strings:\n");

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

        printf("%s\n", decoded[i]);

        free(decoded[i]); // Free memory allocated for each decoded string

    }

    free(decoded); // Free memory allocated for the decoded array

    // Free memory allocated for the encoded string

    free(encoded);

    return 0;

}

// Your functions will be called as such:

// char\* s = encode(strs, strsSize);

// decode(s, &returnSize);

## Longest Common Subsequence

Given two strings text1 and text2, return *the length of their longest****common subsequence****.*If there is no **common subsequence**, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

* For example, "ace" is a subsequence of "abcde".

A **common subsequence** of two strings is a subsequence that is common to both strings.

**Example 1:**

**Input:** text1 = "abcde", text2 = "ace"

**Output:** 3

**Explanation:** The longest common subsequence is "ace" and its length is 3.

**Example 2:**

**Input:** text1 = "abc", text2 = "abc"

**Output:** 3

**Explanation:** The longest common subsequence is "abc" and its length is 3.

**Example 3:**

**Input:** text1 = "abc", text2 = "def"

**Output:** 0

**Explanation:** There is no such common subsequence, so the result is 0.

int max(int a, int b) {

    return a > b ? a : b;

}

int longestCommonSubsequence(char\* text1, char\* text2) {

    int len1 = strlen(text1);

    int len2 = strlen(text2);

    // Create a 2D array to store the lengths of LCS for subproblems

    int dp[len1 + 1][len2 + 1];

    // Initialize the first row and column to 0

    for (int i = 0; i <= len1; i++) {

        for (int j = 0; j <= len2; j++) {

            if (i == 0 || j == 0) {

                dp[i][j] = 0;

            }

        }

    }

    // Fill the dp array to find the length of LCS

    for (int i = 1; i <= len1; i++) {

        for (int j = 1; j <= len2; j++) {

            if (text1[i - 1] == text2[j - 1]) {

                dp[i][j] = dp[i - 1][j - 1] + 1;

            } else {

                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);

            }

        }

    }

    // The length of the LCS is stored in the bottom-right cell of dp array

    return dp[len1][len2];

}

#include <stdio.h>

#include <string.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

void findLCS(char\* str1, char\* str2) {

int m = strlen(str1);

int n = strlen(str2);

int lcs[m + 1][n + 1];

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

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

if (i == 0 || j == 0)

lcs[i][j] = 0;

else if (str1[i - 1] == str2[j - 1])

lcs[i][j] = lcs[i - 1][j - 1] + 1;

else

lcs[i][j] = max(lcs[i - 1][j], lcs[i][j - 1]);

}

}

int length = lcs[m][n];

char result[length + 1];

result[length] = '\0';

int i = m, j = n;

while (i > 0 && j > 0) {

if (str1[i - 1] == str2[j - 1]) {

result[length - 1] = str1[i - 1];

i--;

j--;

length--;

} else if (lcs[i - 1][j] > lcs[i][j - 1]) {

i--;

} else {

j--;

}

}

printf("Longest Common Subsequence: %s\n", result);

}

int main() {

char str1[] = "ABCBDAB";

char str2[] = "BDCAB";

findLCS(str1, str2);

return 0;

}

## Word Break

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

**Note** that the same word in the dictionary may be reused multiple times in the segmentation.

**Example 1:**

**Input:** s = "leetcode", wordDict = ["leet","code"]

**Output:** true

**Explanation:** Return true because "leetcode" can be segmented as "leet code".

**Example 2:**

**Input:** s = "applepenapple", wordDict = ["apple","pen"]

**Output:** true

**Explanation:** Return true because "applepenapple" can be segmented as "apple pen apple".

Note that you are allowed to reuse a dictionary word.

**Example 3:**

**Input:** s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]

**Output:** false

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

int n = s.length();

Set<String> words = new HashSet<>(wordDict);

boolean[] dp = new boolean[n + 1];

dp[0] = true;

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

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

if (dp[j] && words.contains(s.substring(j, i))) {

dp[i] = true;

break;

}

}

}

return dp[n];

}

}

## Combination Sum IV

Given an array of **distinct** integers nums and a target integer target, return *the number of possible combinations that add up to* target.

The test cases are generated so that the answer can fit in a **32-bit** integer.

**Example 1:**

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

**Output:** 7

**Explanation:**

The possible combination ways are:

(1, 1, 1, 1)

(1, 1, 2)

(1, 2, 1)

(1, 3)

(2, 1, 1)

(2, 2)

(3, 1)

Note that different sequences are counted as different combinations.

class Solution {

public int combinationSum4(int[] nums, int target) {

// minor optimization

// Arrays.sort(nums);

int[] dp = new int[target + 1];

dp[0] = 1;

for (int combSum = 1; combSum < target + 1; ++combSum) {

for (int num : nums) {

if (combSum - num >= 0)

dp[combSum] += dp[combSum - num];

// minor optimizaton, early stopping

// else

// break;

}

}

return dp[target];

}

}

## House Robber II

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight****without alerting the police***.

**Example 1:**

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

**Output:** 3

**Explanation:** You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.

**Example 2:**

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

**Output:** 4

**Explanation:** Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

**Example 3:**

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

**Output:** 3

class Solution {

public int rob(int[] nums) {

if (nums.length == 0)

return 0;

if (nums.length == 1)

return nums[0];

int max1 = rob\_simple(nums, 0, nums.length - 2);

int max2 = rob\_simple(nums, 1, nums.length - 1);

return Math.max(max1, max2);

}

public int rob\_simple(int[] nums, int start, int end) {

int t1 = 0;

int t2 = 0;

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

int temp = t1;

int current = nums[i];

t1 = Math.max(current + t2, t1);

t2 = temp;

}

return t1;

}

}

## Decode Ways

A message containing letters from A-Z can be encoded into numbers using the following mapping:

'A' -> "1"

'B' -> "2"

...

'Z' -> "26"

To decode an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:

• "AAJF" with the grouping (1 1 10 6)

• "KJF" with the grouping (11 10 6)

Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".

Given a string s containing only digits, return the number of ways to decode it.

The test cases are generated so that the answer fits in a 32-bit integer.

Example 1:

Input: s = "12"

Output: 2

Explanation: "12" could be decoded as "AB" (1 2) or "L" (12).

Example 2:

Input: s = "226"

Output: 3

Explanation: "226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

Example 3:

Input: s = "06"

Output: 0

Explanation: "06" cannot be mapped to "F" because of the leading zero ("6" is different from "06").

class Solution {

public int numDecodings(String s) {

if (s.charAt(0) == '0') {

return 0;

}

int n = s.length();

int twoBack = 1;

int oneBack = 1;

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

int current = 0;

if (s.charAt(i) != '0') {

current = oneBack;

}

int twoDigit = Integer.parseInt(s.substring(i - 1, i + 1));

if (twoDigit >= 10 && twoDigit <= 26) {

current += twoBack;

}

twoBack = oneBack;

oneBack = current;

}

return oneBack;

}

}

## Find Median From Data Stream

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

* For example, for arr = [2,3,4], the median is 3.
* For example, for arr = [2,3], the median is (2 + 3) / 2 = 2.5.

Implement the MedianFinder class:

* MedianFinder() initializes the MedianFinder object.
* void addNum(int num) adds the integer num from the data stream to the data structure.
* double findMedian() returns the median of all elements so far. Answers within 10-5 of the actual answer will be accepted.

**Example 1:**

**Input**

["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]

[[], [1], [2], [], [3], []]

**Output**

[null, null, null, 1.5, null, 2.0]

**Explanation**

MedianFinder medianFinder = new MedianFinder();

medianFinder.addNum(1); // arr = [1]

medianFinder.addNum(2); // arr = [1, 2]

medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)

medianFinder.addNum(3); // arr[1, 2, 3]

medianFinder.findMedian(); // return 2.0

class MedianFinder {

    private PriorityQueue<Integer> small = new PriorityQueue<>(Collections.reverseOrder());

    private PriorityQueue<Integer> large = new PriorityQueue<>();

    private boolean even = true;

    public MedianFinder() {

    }

    public void addNum(int num) {

          if (even) {

        large.offer(num);

        small.offer(large.poll());

    } else {

        small.offer(num);

        large.offer(small.poll());

    }

    even = !even;

    }

    public double findMedian() {

         if (even)

        return (small.peek() + large.peek()) / 2.0;

    else

        return small.peek();

    }

}

/\*\*

 \* Your MedianFinder object will be instantiated and called as such:

 \* MedianFinder obj = new MedianFinder();

 \* obj.addNum(num);

 \* double param\_2 = obj.findMedian();

 \*/

class MedianFinder {

ArrayList<Integer> list;

public MedianFinder() {

list = new ArrayList<>();

}

public void addNum(int num) {

int i;

if(list.size() > 0){

for (i = 0; (i < list.size() && list.get(i) < num); i++);

list.add(i , num);

}else{

list.add(num);

}

}

public double findMedian() {

// System.out.println(list);

int index = list.size()/2;

if(list.size() % 2 == 0){

return (double) (list.get(index) + list.get(index - 1))/2;

}else{

return list.get(index);

}

}

}

## Maximum Depth Of Binary Tree

Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Example 1:**

A diagram of a network

Description automatically generated

**Input:** root = [3,9,20,null,null,15,7]

**Output:** 3

**Example 2:**

**Input:** root = [1,null,2]

**Output:** 2

int max(int a, int b) {

    return (a > b) ? a : b;

}

int maxDepth(struct TreeNode\* root){

    if (root == NULL) {

        return 0;

    }

    return max(1 + maxDepth(root -> left), 1 + maxDepth(root -> right));

}

int main() {

    // Create a sample binary tree

    struct TreeNode\* root = newNode(1);

    root->left = newNode(2);

    root->right = newNode(3);

    root->left->left = newNode(4);

    root->left->right = newNode(5);

    // Calculate the maximum depth of the binary tree

    int depth = maxDepth(root);

    // Print the result

    printf("Maximum depth of the binary tree: %d\n", depth);

    // Free dynamically allocated memory

    // In a real-world scenario, you should implement a function to free the entire tree

    free(root->left->right);

    free(root->left->left);

    free(root->right);

    free(root->left);

    free(root);

    return 0;

}

## Same Tree

Given the roots of two binary trees p and q, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

**Example 1:**

A black circle with black lines and numbers

Description automatically generated with medium confidence

**Input:** p = [1,2,3], q = [1,2,3]

**Output:** true

**Example 2:**

A diagram of a number one and two circles

Description automatically generated

**Input:** p = [1,2], q = [1,null,2]

**Output:** false

**Example 3:**

A black circle with white circles and black lines

Description automatically generated with medium confidence

**Input:** p = [1,2,1], q = [1,1,2]

**Output:** false

bool isSameTree(struct TreeNode\* p, struct TreeNode\* q) {

    if (p == NULL && q == NULL) {

        return true;

    }

    else if ((p == NULL && q != NULL) ||

            (q == NULL && p != NULL)) {

                return false;

            }

    if (p->val != q->val) {

        return false;

    }

    return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

}

int main() {

    // Create two sample binary trees

    struct TreeNode\* p = newNode(1);

    p->left = newNode(2);

    p->right = newNode(3);

    struct TreeNode\* q = newNode(1);

    q->left = newNode(2);

    q->right = newNode(3);

    // Check if the two trees are the same

    bool result = isSameTree(p, q);

    // Print the result

    if (result) {

        printf("The two binary trees are the same.\n");

    } else {

        printf("The two binary trees are not the same.\n");

    }

    // Free dynamically allocated memory

    free(p->right);

    free(p->left);

    free(p);

    free(q->right);

    free(q->left);

    free(q);

    return 0;

}

## Invert Binary Tree

Given the root of a binary tree, invert the tree, and return *its root*.

**Example 1:**

A diagram of a diagram

Description automatically generated

**Input:** root = [4,2,7,1,3,6,9]

**Output:** [4,7,2,9,6,3,1]

**Example 2:**

A couple of round objects with a arrow pointing to the left

Description automatically generated with medium confidence

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

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

**Example 3:**

**Input:** root = []

**Output:** []

struct TreeNode\* invertTree(struct TreeNode\* root) {

    if (root == NULL) {

        return NULL;

    }

    // Swap left and right subtrees

    struct TreeNode\* temp = root->left;

    root->left = root->right;

    root->right = temp;

    // Invert left and right subtrees recursively

    invertTree(root->left);

    invertTree(root->right);

    return root;

}

int main() {

    // Create a sample binary tree

    struct TreeNode\* root = newNode(4);

    root->left = newNode(2);

    root->right = newNode(7);

    root->left->left = newNode(1);

    root->left->right = newNode(3);

    root->right->left = newNode(6);

    root->right->right = newNode(9);

    // Print the original binary tree

    printf("Original binary tree (inorder traversal): ");

    printInorder(root);

    printf("\n");

    // Invert the binary tree

    struct TreeNode\* invertedRoot = invertTree(root);

    // Print the inverted binary tree

    printf("Inverted binary tree (inorder traversal): ");

    printInorder(invertedRoot);

    printf("\n");

    // Free dynamically allocated memory

    // Note: Implementing a function to free the binary tree nodes is recommended for larger trees.

    return 0;

}

## Binary Tree Maximum Path Sum

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the root of a binary tree, return *the maximum****path sum****of any****non-empty****path*.

**Example 1:**

A diagram of a triangle

Description automatically generated

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

**Output:** 6

**Explanation:** The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.

**Example 2:**

A diagram of a network

Description automatically generated

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

 #define max(a,b) (a > b)?a:b

 int helper(struct TreeNode\* root, int \*ans){

        // Escape condition

        if(!root) return 0;

        // DFS

        int left = max(0, helper(root->left,ans));

        int right = max(0, helper(root->right,ans));

        // There is also possibility left+parent+right > ans

        \*ans = max(\*ans, root->val + left + right);

        // Inorder to make the path continous we can send either right or the left side,

        // So we will send the maximum side

        return root->val + max(left, right);

    }

    int maxPathSum(struct TreeNode\* root) {

        // The answer

        int ans = INT\_MIN;

        if(!root) return 0;

        // Recurssive function

        helper(root, &ans);

        return ans;

    }

int main() {

    // Create a sample binary tree

    struct TreeNode\* root = newNode(10);

    root->left = newNode(2);

    root->right = newNode(10);

    root->left->left = newNode(20);

    root->left->right = newNode(1);

    root->right->right = newNode(-25);

    root->right->right->left = newNode(3);

    root->right->right->right = newNode(4);

    // Calculate the maximum path sum

    int maxSum = maxPathSum(root);

    // Print the maximum path sum

    printf("Maximum path sum in the binary tree: %d\n", maxSum);

    // Free dynamically allocated memory

    // Note: Implementing a function to free the binary tree nodes is recommended for larger trees.

    return 0;

}

class Solution {

    int max = Integer.MIN\_VALUE;

    public int maxPath(TreeNode root) {

        if(root == null) return 0;

        int value = root.val;

        int left\_sum = Math.max(maxPath(root.left),0);

        int right\_sum = Math.max(maxPath(root.right),0);

        max = Math.max(max, left\_sum + right\_sum + value);

        return Math.max(left\_sum, right\_sum) + value;

    }

    public int maxPathSum(TreeNode root) {

        maxPath(root);

        return max;

    }

}

## Binary Tree Level Order Traversal

Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

**Example 1:**

A diagram of a triangle

Description automatically generated

**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[3],[9,20],[15,7]]

**Example 2:**

**Input:** root = [1]

**Output:** [[1]]

**Example 3:**

**Input:** root = []

**Output:** []

int\*\* levelOrder(struct TreeNode\* root, int\* returnSize, int\*\* returnColumnSizes) {

    // Check if the tree is empty

    if (root == NULL) {

        \*returnSize = 0;

        \*returnColumnSizes = NULL;

        return NULL;

    }

    // Initialize a queue for BFS

    struct TreeNode\*\* queue = (struct TreeNode\*\*)malloc(sizeof(struct TreeNode\*) \* 10000);

    int front = 0, rear = -1;

    // Initialize 2D array to store level-order traversal

    int\*\* result = (int\*\*)malloc(sizeof(int\*) \* 10000);

    \*returnColumnSizes = (int\*)malloc(sizeof(int) \* 10000);

    int levelSize = 0;

    \*returnSize = 0;

    // Enqueue the root node

    queue[++rear] = root;

    while (front <= rear) {

        levelSize = rear - front + 1;

        (\*returnColumnSizes)[\*returnSize] = levelSize;

        // Allocate memory for the current level

        result[\*returnSize] = (int\*)malloc(sizeof(int) \* levelSize);

        // Process nodes at the current level

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

            struct TreeNode\* node = queue[front++];

            result[\*returnSize][i] = node->val;

            // Enqueue left and right children, if present

            if (node->left) queue[++rear] = node->left;

            if (node->right) queue[++rear] = node->right;

        }

        (\*returnSize)++;

    }

    // Free the queue memory

    free(queue);

    return result;

}

int main() {

    // Create a sample binary tree

    struct TreeNode\* root = newNode(3);

    root->left = newNode(9);

    root->right = newNode(20);

    root->right->left = newNode(15);

    root->right->right = newNode(7);

    // Variables to store the level-order traversal result

    int returnSize;

    int\* returnColumnSizes;

    // Call the levelOrder function to get the level-order traversal

    int\*\* result = levelOrder(root, &returnSize, &returnColumnSizes);

    // Print the level-order traversal result

    printf("Level-order traversal:\n");

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

        printf("Level %d: ", i + 1);

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

            printf("%d ", result[i][j]);

        }

        printf("\n");

        // Free memory allocated for each level

        free(result[i]);

    }

    // Free memory allocated for the result and column sizes

    free(result);

    free(returnColumnSizes);

    // Free memory allocated for the binary tree

    freeBinaryTree(root);

    return 0;

}

## Kth Smallest Element In BST

Given the root of a binary search tree, and an integer k, return *the* kth *smallest value (****1-indexed****) of all the values of the nodes in the tree*.

**Example 1:**

A diagram of a network

Description automatically generated

**Input:** root = [3,1,4,null,2], k = 1

**Output:** 1

**Example 2:**

A diagram of a network

Description automatically generated

**Input:** root = [5,3,6,2,4,null,null,1], k = 3

**Output:** 3

class Solution {

    public ArrayList<Integer> inorder(TreeNode root, ArrayList<Integer> arr) {

        if (root == null) return arr;

        inorder(root.left, arr);

        arr.add(root.val);

        inorder(root.right, arr);

        return arr;

    }

    public int kthSmallest(TreeNode root, int k) {

        ArrayList<Integer> nums = inorder(root, new ArrayList<Integer>());

        return nums.get(k - 1);

    }

}

#include <stdio.h>

#include <stdlib.h>

// Definition for a binary tree node.

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Function to create a new binary tree node

struct TreeNode\* newNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

// Function to free the memory allocated for a binary tree

void freeBinaryTree(struct TreeNode\* root) {

    if (root == NULL) return;

    freeBinaryTree(root->left);

    freeBinaryTree(root->right);

    free(root);

}

// Function to perform inorder traversal and store values in an array

void inorder(struct TreeNode\* root, int\* arr, int\* index) {

    if (root == NULL) return;

    inorder(root->left, arr, index);

    arr[(\*index)++] = root->val;

    inorder(root->right, arr, index);

}

// Function to find the kth smallest element in a BST

int kthSmallest(struct TreeNode\* root, int k) {

    int arr[10000]; // Assuming a maximum of 10000 nodes

    int index = 0;

    inorder(root, arr, &index);

    return arr[k - 1];

}

int main() {

    // Create a sample binary search tree

    struct TreeNode\* root = newNode(3);

    root->left = newNode(1);

    root->right = newNode(4);

    root->left->right = newNode(2);

    // Find the kth smallest element

    int k = 1; // Example: find the 1st smallest element

    int kth = kthSmallest(root, k);

    printf("The %dth smallest element is: %d\n", k, kth);

    // Free memory allocated for the binary tree

    freeBinaryTree(root);

    return 0;

}

## SubTree Of Another Tree

Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

**Example 1:**

A diagram of a tree

Description automatically generated

**Input:** root = [3,4,5,1,2], subRoot = [4,1,2]

**Output:** true

**Example 2:**

A diagram of a tree

Description automatically generated

**Input:** root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]

**Output:** false

#include <stdio.h>

#include <stdbool.h>

// Definition for a binary tree node

struct TreeNode {

    int val;

    struct TreeNode \*left;

    struct TreeNode \*right;

};

// Function to check if two trees are identical

bool isSame(struct TreeNode\* s, struct TreeNode\* t) {

    if (s == NULL && t == NULL) return true;

    if (s == NULL || t == NULL) return false;

    if (s->val != t->val) return false;

    return isSame(s->left, t->left) && isSame(s->right, t->right);

}

// Function to check if t is a subtree of s

bool isSubtree(struct TreeNode\* s, struct TreeNode\* t) {

    if (s == NULL) return false;

    if (isSame(s, t)) return true;

    return isSubtree(s->left, t) || isSubtree(s->right, t);

}

int main() {

    // Example trees

    struct TreeNode node1 = {1, NULL, NULL};

    struct TreeNode node2 = {2, NULL, NULL};

    struct TreeNode node3 = {3, NULL, NULL};

    struct TreeNode node4 = {4, NULL, NULL};

    struct TreeNode node5 = {5, NULL, NULL};

    struct TreeNode node6 = {6, NULL, NULL};

    // Constructing the tree structure

    node3.left = &node4;

    node3.right = &node5;

    node2.left = &node3;

    node2.right = &node6;

    node1.left = &node2;

    // Example subtree

    struct TreeNode subRoot = {2, NULL, NULL};

    subRoot.left = &node3;

    subRoot.right = &node6;

    // Test the isSubtree function

    if (isSubtree(&node1, &subRoot)) {

        printf("subRoot is a subtree of node1.\n");

    } else {

        printf("subRoot is not a subtree of node1.\n");

    }

    return 0;

}

public class Solution {

    public boolean isSubtree(TreeNode s, TreeNode t) {

        if (s == null) return false;

        if (isSame(s, t)) return true;

        return isSubtree(s.left, t) || isSubtree(s.right, t);

    }

    private boolean isSame(TreeNode s, TreeNode t) {

        if (s == null && t == null) return true;

        if (s == null || t == null) return false;

        if (s.val != t.val) return false;

        return isSame(s.left, t.left) && isSame(s.right, t.right);

    }

}

## Word Search II

Given an m x n board of characters and a list of strings words, return *all words on the board*.

Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

**Example 1:**

A close-up of a crossword

Description automatically generated

**Input:** board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]

**Output:** ["eat","oath"]

**Example 2:**

A white square with black letters

Description automatically generated

**Input:** board = [["a","b"],["c","d"]], words = ["abcb"]

**Output:** []

class TrieNode {

HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();

String word = null;

public TrieNode() {}

}

class Solution {

char[][] \_board = null;

ArrayList<String> \_result = new ArrayList<String>();

public List<String> findWords(char[][] board, String[] words) {

// Step 1). Construct the Trie

TrieNode root = new TrieNode();

for (String word : words) {

TrieNode node = root;

for (Character letter : word.toCharArray()) {

if (node.children.containsKey(letter)) {

node = node.children.get(letter);

} else {

TrieNode newNode = new TrieNode();

node.children.put(letter, newNode);

node = newNode;

}

}

node.word = word; // store words in Trie

}

this.\_board = board;

// Step 2). Backtracking starting for each cell in the board

for (int row = 0; row < board.length; ++row) {

for (int col = 0; col < board[row].length; ++col) {

if (root.children.containsKey(board[row][col])) {

backtracking(row, col, root);

}

}

}

return this.\_result;

}

private void backtracking(int row, int col, TrieNode parent) {

Character letter = this.\_board[row][col];

TrieNode currNode = parent.children.get(letter);

// check if there is any match

if (currNode.word != null) {

this.\_result.add(currNode.word);

currNode.word = null;

}

// mark the current letter before the EXPLORATION

this.\_board[row][col] = '#';

// explore neighbor cells in around-clock directions: up, right, down, left

int[] rowOffset = {-1, 0, 1, 0};

int[] colOffset = {0, 1, 0, -1};

for (int i = 0; i < 4; ++i) {

int newRow = row + rowOffset[i];

int newCol = col + colOffset[i];

if (newRow < 0 || newRow >= this.\_board.length || newCol < 0

|| newCol >= this.\_board[0].length) {

continue;

}

if (currNode.children.containsKey(this.\_board[newRow][newCol])) {

backtracking(newRow, newCol, currNode);

}

}

// End of EXPLORATION, restore the original letter in the board.

this.\_board[row][col] = letter;

// Optimization: incrementally remove the leaf nodes

if (currNode.children.isEmpty()) {

parent.children.remove(letter);

}

}

}

## Clone Graph

Given a reference of a node in a [**connected**](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)#Connected_graph) undirected graph.

Return a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

class Node {

public int val;

public List<Node> neighbors;

}

**Test case format:**

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with val == 1, the second node with val == 2, and so on. The graph is represented in the test case using an adjacency list.

**An adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the **copy of the given node** as a reference to the cloned graph.

**Example 1:**

A diagram of a graph

Description automatically generated

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

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

**Explanation:** There are 4 nodes in the graph.

1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

**Example 2:**

A black circle with a number in it

Description automatically generated

#include <stdlib.h>

#include <stdio.h>

struct Node {

    int val;

    struct Node\*\* neighbors;

    int numNeighbors;

};

struct Node\* cloneGraphUtil(struct Node\* node, struct Node\*\* visited) {

    if (node == NULL) {

        return NULL;

    }

    // If the node was already visited before.

    // Return the clone from the visited array.

    if (visited[node->val] != NULL) {

        return visited[node->val];

    }

    // Create a clone for the given node.

    // Note that we don't have cloned neighbors as of now, hence NULL.

    struct Node\* cloneNode = (struct Node\*)malloc(sizeof(struct Node));

    cloneNode->val = node->val;

    cloneNode->neighbors = (struct Node\*\*)malloc(node->numNeighbors \* sizeof(struct Node\*));

    cloneNode->numNeighbors = node->numNeighbors;

    // Mark the original node as visited and store its clone.

    visited[node->val] = cloneNode;

    // Iterate through the neighbors to generate their clones

    // and prepare an array of cloned neighbors to be added to the cloned node.

    for (int i = 0; i < node->numNeighbors; i++) {

        cloneNode->neighbors[i] = cloneGraphUtil(node->neighbors[i], visited);

    }

    return cloneNode;

}

struct Node\* cloneGraph(struct Node\* node) {

    if (node == NULL) {

        return NULL;

    }

    // Allocate memory for visited array and initialize it to NULL

    struct Node\*\* visited = (struct Node\*\*)calloc(101, sizeof(struct Node\*));

    // Call the utility function to clone the graph

    struct Node\* clonedGraph = cloneGraphUtil(node, visited);

    // Free the memory allocated for the visited array

    free(visited);

    return clonedGraph;

}

int main() {

    // Create a graph for testing

    struct Node\* originalGraph = createGraph();

    // Clone the graph

    struct Node\* clonedGraph = cloneGraph(originalGraph);

    // Print the cloned graph

    printf("Original Graph:\n");

    printGraph(originalGraph);

    printf("\nCloned Graph:\n");

    printGraph(clonedGraph);

    // Free the memory allocated for the original and cloned graphs

    // (Assuming a function to free the graph nodes is implemented)

    // freeGraph(originalGraph);

    // freeGraph(clonedGraph);

    return 0;

}

**Input:** adjList = [[]]

**Output:** [[]]

**Explanation:** Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

**Example 3:**

**Input:** adjList = []

**Output:** []

**Explanation:** This an empty graph, it does not have any nodes.

class Solution {

private HashMap <Node, Node> visited = new HashMap <> ();

public Node cloneGraph(Node node) {

if (node == null) {

return node;

}

// If the node was already visited before.

// Return the clone from the visited dictionary.

if (visited.containsKey(node)) {

return visited.get(node);

}

// Create a clone for the given node.

// Note that we don't have cloned neighbors as of now, hence [].

Node cloneNode = new Node(node.val, new ArrayList());

// The key is original node and value being the clone node.

visited.put(node, cloneNode);

// Iterate through the neighbors to generate their clones

// and prepare a list of cloned neighbors to be added to the cloned node.

for (Node neighbor: node.neighbors) {

cloneNode.neighbors.add(cloneGraph(neighbor));

}

return cloneNode;

}

}

## Course Schedule

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you **must** take course bi first if you want to take course ai.

* For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

**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.

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

int[] indegree = new int[numCourses];

List<List<Integer>> adj = new ArrayList<>(numCourses);

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

adj.add(new ArrayList<>());

}

for (int[] prerequisite : prerequisites) {

adj.get(prerequisite[1]).add(prerequisite[0]);

indegree[prerequisite[0]]++;

}

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

// Push all the nodes with indegree zero in the queue.

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

if (indegree[i] == 0) {

queue.offer(i);

}

}

int nodesVisited = 0;

while (!queue.isEmpty()) {

int node = queue.poll();

nodesVisited++;

for (int neighbor : adj.get(node)) {

// Delete the edge "node -> neighbor".

indegree[neighbor]--;

if (indegree[neighbor] == 0) {

queue.offer(neighbor);

}

}

}

return nodesVisited == numCourses;

}

}

class Solution {

public boolean dfs(int node, List<List<Integer>> adj, boolean[] visit, boolean[] inStack) {

// If the node is already in the stack, we have a cycle.

if (inStack[node]) {

return true;

}

if (visit[node]) {

return false;

}

// Mark the current node as visited and part of current recursion stack.

visit[node] = true;

inStack[node] = true;

for (int neighbor : adj.get(node)) {

if (dfs(neighbor, adj, visit, inStack)) {

return true;

}

}

// Remove the node from the stack.

inStack[node] = false;

return false;

}

public boolean canFinish(int numCourses, int[][] prerequisites) {

List<List<Integer>> adj = new ArrayList<>(numCourses);

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

adj.add(new ArrayList<>());

}

for (int[] prerequisite : prerequisites) {

adj.get(prerequisite[1]).add(prerequisite[0]);

}

boolean[] visit = new boolean[numCourses];

boolean[] inStack = new boolean[numCourses];

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

if (dfs(i, adj, visit, inStack)) {

return false;

}

}

return true;

}

## Pacific Atlantic Water Flow

There is an m x n rectangular island that borders both the **Pacific Ocean** and **Atlantic Ocean**. The **Pacific Ocean** touches the island's left and top edges, and the **Atlantic Ocean** touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an m x n integer matrix heights where heights[r][c] represents the **height above sea level** of the cell at coordinate (r, c).

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is **less than or equal to** the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return *a****2D list****of grid coordinates*result*where*result[i] = [ri, ci]*denotes that rain water can flow from cell*(ri, ci)*to****both****the Pacific and Atlantic oceans*.

**Example 1:**

A blue and white square with black numbers

Description automatically generated

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

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

**Explanation:** The following cells can flow to the Pacific and Atlantic oceans, as shown below:

[0,4]: [0,4] -> Pacific Ocean

  [0,4] -> Atlantic Ocean

[1,3]: [1,3] -> [0,3] -> Pacific Ocean

  [1,3] -> [1,4] -> Atlantic Ocean

[1,4]: [1,4] -> [1,3] -> [0,3] -> Pacific Ocean

  [1,4] -> Atlantic Ocean

[2,2]: [2,2] -> [1,2] -> [0,2] -> Pacific Ocean

  [2,2] -> [2,3] -> [2,4] -> Atlantic Ocean

[3,0]: [3,0] -> Pacific Ocean

  [3,0] -> [4,0] -> Atlantic Ocean

[3,1]: [3,1] -> [3,0] -> Pacific Ocean

  [3,1] -> [4,1] -> Atlantic Ocean

[4,0]: [4,0] -> Pacific Ocean

[4,0] -> Atlantic Ocean

Note that there are other possible paths for these cells to flow to the Pacific and Atlantic oceans.

**Example 2:**

**Input:** heights = [[1]]

**Output:** [[0,0]]

**Explanation:** The water can flow from the only cell to the Pacific and Atlantic oceans.

class Solution {

private static final int[][] DIRECTIONS = new int[][]{{0, 1}, {1, 0}, {-1, 0}, {0, -1}};

private int numRows;

private int numCols;

private int[][] landHeights;

public List<List<Integer>> pacificAtlantic(int[][] matrix) {

// Check if input is empty

if (matrix.length == 0 || matrix[0].length == 0) {

return new ArrayList<>();

}

// Save initial values to parameters

numRows = matrix.length;

numCols = matrix[0].length;

landHeights = matrix;

boolean[][] pacificReachable = new boolean[numRows][numCols];

boolean[][] atlanticReachable = new boolean[numRows][numCols];

// Loop through each cell adjacent to the oceans and start a DFS

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

dfs(i, 0, pacificReachable);

dfs(i, numCols - 1, atlanticReachable);

}

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

dfs(0, i, pacificReachable);

dfs(numRows - 1, i, atlanticReachable);

}

// Find all cells that can reach both oceans

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

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

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

if (pacificReachable[i][j] && atlanticReachable[i][j]) {

commonCells.add(List.of(i, j));

}

}

}

return commonCells;

}

private void dfs(int row, int col, boolean[][] reachable) {

// This cell is reachable, so mark it

reachable[row][col] = true;

for (int[] dir : DIRECTIONS) { // Check all 4 directions

int newRow = row + dir[0];

int newCol = col + dir[1];

// Check if new cell is within bounds

if (newRow < 0 || newRow >= numRows || newCol < 0 || newCol >= numCols) {

continue;

}

// Check that the new cell hasn't already been visited

if (reachable[newRow][newCol]) {

continue;

}

// Check that the new cell has a higher or equal height,

// So that water can flow from the new cell to the old cell

if (landHeights[newRow][newCol] < landHeights[row][col]) {

continue;

}

// If we've gotten this far, that means the new cell is reachable

dfs(newRow, newCol, reachable);

}

}

## Alien Dictionary

There is a new alien language that uses the English alphabet. However, the order of the letters is unknown to you.

You are given a list of strings words from the alien language's dictionary. Now it is claimed that the strings in words are

**sorted lexicographically**

 by the rules of this new language.

If this claim is incorrect, and the given arrangement of string in words cannot correspond to any order of letters, return "".

Otherwise, return *a string of the unique letters in the new alien language sorted in****lexicographically increasing order****by the new language's rules.*If there are multiple solutions, return***any of them***.

**Example 1:**

**Input:** words = ["wrt","wrf","er","ett","rftt"]

**Output:** "wertf"

**Example 2:**

**Input:** words = ["z","x"]

**Output:** "zx"

**Example 3:**

**Input:** words = ["z","x","z"]

**Output:** ""

**Explanation:** The order is invalid, so return "".

class Solution {

    public String alienOrder(String[] words) {

        if(words.length==0)

            return "";

        Map<Character,Integer>inDegree=new HashMap<>();

        Map<Character,List<Character>>graph=new HashMap<>();

        //a. Initialise adjacency list and in degree map

        for (String word : words) {

            for (char character : word.toCharArray()) {

                inDegree.put(character, 0);

                graph.put(character, new ArrayList<Character>());

            }

        }

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

            String w1 = words[i], w2 = words[i + 1];

            //Checks if str1 starts wtih str2

            if (w1.length() > w2.length() && w1 .startsWith(w2)) {

                return "";

            }

            //b. Build the graph and in degree map

            for (int j = 0; j < Math.min(w1.length(), w2.length()); j++) {

                char parent = w1.charAt(j), child = w2.charAt(j);

                if (parent != child) { // if the two characters are different

                  graph.get(parent).add(child); // put the child into it's parent's list

                  inDegree.put(child, inDegree.get(child) + 1); // increment child's inDegree

                  break; // only the first different character between the two words will help us find the order

                }

            }

        }

            // c. Find all sources i.e., all vertices with 0 in-degrees

            Queue<Character> sources = new LinkedList<>();

            for (Map.Entry<Character, Integer> entry : inDegree.entrySet()) {

              if (entry.getValue() == 0)

                sources.add(entry.getKey());

            }

        // d. For each source, add it to the sortedOrder and subtract one from all of its children's in-degrees

        // if a child's in-degree becomes zero, add it to the sources queue

        StringBuilder sortedOrder = new StringBuilder();

        while (!sources.isEmpty()) {

          Character vertex = sources.poll();

          sortedOrder.append(vertex);

          List<Character> children = graph.get(vertex); // get the node's children to decrement their in-degrees

          for (Character child : children) {

            inDegree.put(child, inDegree.get(child) - 1);

            if (inDegree.get(child) == 0)

              sources.add(child);

          }

        }

        // if sortedOrder doesn't contain all characters, there is a cyclic dependency between characters, therefore, we

        // will not be able to find the correct ordering of the characters

        if (sortedOrder.length() != inDegree.size())

          return "";

        return sortedOrder.toString();

    }

}

## Graph Valid Tree

You have a graph of n nodes labeled from 0 to n - 1. You are given an integer n and a list of edges where edges[i] = [ai, bi] indicates that there is an undirected edge between nodes ai and bi in the graph.

Return true *if the edges of the given graph make up a valid tree, and* false *otherwise*.

**Example 1:**

A diagram of a tree

Description automatically generated

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

**Output:** true

**Example 2:**

A diagram of a network

Description automatically generated

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

**Output:** false

class Solution {

// A graph is considered a tree if it is connected and the number of edges = n-1 nodes

public boolean validTree(int n, int[][] edges) {

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

//create graph

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

graph.add(new ArrayList<>());

}

//since this is an undirected graph, make each edge go both ways

for (int[] edge : edges) {

graph.get(edge[1]).add(edge[0]);

graph.get(edge[0]).add(edge[1]);

}

HashSet<Integer> visited = new HashSet<>();

//if graph is connected then we would be able to start from any node and reach all the other nodes using dfs

//we can dfs starting from first node since this is guaranteed to exist according to constraints

dfs(visited, 0, graph);

//if we see an unvisited node,

//the graph is not connected and therefore cannot be a tree

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

if(!visited.contains(i)) return false;

}

//also check number of edges = n-1 nodes

return (edges.length == n-1);

}

public void dfs(HashSet<Integer> visited, int currNode, List<List<Integer>> graph) {

visited.add(currNode);

for (Integer node : graph.get(currNode)) {

if(!visited.contains(node)) {

dfs(visited, node, graph);

}

}

}

}

## Number Of Connected Components In an Undirected Graph

You have a graph of n nodes. You are given an integer n and an array edges where edges[i] = [ai, bi] indicates that there is an edge between ai and bi in the graph.

Return *the number of connected components in the graph*.

**Example 1:**

A diagram of a network

Description automatically generated

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

**Output:** 2

**Example 2:**

A diagram of a connection

Description automatically generated

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

**Output:** 1

class Solution {

   public int countComponents(int n, int[][] edges) {

    int[] roots = new int[n];

    for(int i = 0; i < n; i++) roots[i] = i;

    for(int[] e : edges) {

        int root1 = find(roots, e[0]);

        int root2 = find(roots, e[1]);

        if(root1 != root2) {

            roots[root1] = root2;  // union

            n--;

        }

    }

    return n;

}

public int find(int[] roots, int id) {

    while(roots[id] != id) {

        roots[id] = roots[roots[id]];  // optional: path compression

        id = roots[id];

    }

    return id;

}

}

## Longest Prefix Match – Trie (Store and Query IP address efficiently)

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

// Define the structure for a trie node

typedef struct TrieNode {

struct TrieNode\* children[2]; // Assuming binary trie for IPv4 addresses

int isLeaf;

} TrieNode;

// Function to create a new trie node

TrieNode\* createNode() {

TrieNode\* node = (TrieNode\*)malloc(sizeof(TrieNode));

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

node->children[i] = NULL;

}

node->isLeaf = 0;

return node;

}

// Function to insert an IP address prefix into the trie

void insert(TrieNode\* root, uint32\_t prefix, int length) {

TrieNode\* current = root;

for (int i = 31; i >= 32 - length; i--) {

int bit = (prefix >> i) & 1;

if (!current->children[bit]) {

current->children[bit] = createNode();

}

current = current->children[bit];

}

current->isLeaf = 1;

}

// Function to perform longest prefix match and return the matched prefix

uint32\_t longestPrefixMatch(TrieNode\* root, uint32\_t ip) {

TrieNode\* current = root;

uint32\_t match = 0;

for (int i = 31; i >= 0; i--) {

int bit = (ip >> i) & 1;

if (current->children[bit]) {

current = current->children[bit];

match = (match << 1) | bit; // Append the current bit to the matched prefix

} else {

break;

}

}

return match;

}

int main() {

// Example usage

TrieNode\* root = createNode();

// Insert some prefixes into the trie

insert(root, 0xC0A80100, 24); // 192.168.1.0/24

insert(root, 0xC0A80180, 26); // 192.168.1.128/26

insert(root, 0xAC100001, 16); // 172.16.0.0/16

// Test longest prefix match

uint32\_t ipAddress = 0xC0A80145; // 192.168.1.69

uint32\_t matchedPrefix = longestPrefixMatch(root, ipAddress);

if (matchedPrefix > 0) {

printf("Longest prefix matched: %0x%x\n", matchedPrefix);

} else {

printf("No matching prefix found.\n");

}

return 0;

}

Next Solution:

#include <stdio.h>

#include <stdint.h>

#include <stdlib.h>

#define IPV4\_OCTETS 4

// Trie Node Definition

typedef struct TrieNode {

struct TrieNode\* children[2];

int isIPPrefix;

int prefixLength;

} TrieNode;

// Function to create a new trie node

TrieNode\* createNode() {

TrieNode\* newNode = (TrieNode\*)malloc(sizeof(TrieNode));

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

newNode->children[i] = NULL;

}

newNode->isIPPrefix = 0;

newNode->prefixLength = 0;

return newNode;

}

// Function to insert an IP prefix into the trie

void insertIPPrefix(TrieNode\* root, uint8\_t\* ipAddress, int prefixLength) {

TrieNode\* current = root;

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

int bit = (ipAddress[i / 8] >> (7 - (i % 8))) & 1;

if (current->children[bit] == NULL) {

current->children[bit] = createNode();

}

current = current->children[bit];

}

current->isIPPrefix = 1;

current->prefixLength = prefixLength;

}

// Function to perform longest prefix match using the trie

int longestPrefixMatch(TrieNode\* root, uint8\_t\* ipAddress) {

TrieNode\* current = root;

int lastMatchedPrefix = -1;

for (int i = 0; i < IPV4\_OCTETS \* 8; ++i) {

int bit = (ipAddress[i / 8] >> (7 - (i % 8))) & 1;

if (current->children[bit] == NULL) {

break;

}

current = current->children[bit];

if (current->isIPPrefix) {

lastMatchedPrefix = i;

}

}

return lastMatchedPrefix;

}

// Function to free memory allocated for the trie

void freeTrie(TrieNode\* root) {

if (root == NULL) {

return;

}

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

freeTrie(root->children[i]);

}

free(root);

}

int main() {

TrieNode\* root = createNode();

// Sample IP prefixes

uint8\_t ipPrefix1[IPV4\_OCTETS] = {192, 168, 1, 0};

uint8\_t ipPrefix2[IPV4\_OCTETS] = {192, 168, 1, 0};

uint8\_t ipPrefix3[IPV4\_OCTETS] = {192, 168, 2, 0};

insertIPPrefix(root, ipPrefix1, 24); // Insert IP prefix 192.168.1.0/24

insertIPPrefix(root, ipPrefix2, 16); // Insert IP prefix 192.168.1.0/16

insertIPPrefix(root, ipPrefix3, 24); // Insert IP prefix 192.168.2.0/24

// Sample IP addresses to match

uint8\_t ipAddress1[IPV4\_OCTETS] = {192, 168, 1, 42};

uint8\_t ipAddress2[IPV4\_OCTETS] = {192, 168, 2, 42};

uint8\_t ipAddress3[IPV4\_OCTETS] = {192, 169, 1, 42};

// Perform longest prefix match

int matchIndex1 = longestPrefixMatch(root, ipAddress1);

int matchIndex2 = longestPrefixMatch(root, ipAddress2);

int matchIndex3 = longestPrefixMatch(root, ipAddress3);

// Print results

if (matchIndex1 != -1) {

printf("Longest prefix match for IP 192.168.1.42: Prefix %d\n", matchIndex1);

} else {

printf("No match for IP 192.168.1.42\n");

}

if (matchIndex2 != -1) {

printf("Longest prefix match for IP 192.168.2.42: Prefix %d\n", matchIndex2);

} else {

printf("No match for IP 192.168.2.42\n");

}

if (matchIndex3 != -1) {

printf("Longest prefix match for IP 192.169.1.42: Prefix %d\n", matchIndex3);

} else {

printf("No match for IP 192.169.1.42\n");

}

// Free allocated memory

freeTrie(root);

return 0;

}

## UDP Socket Communication Client and Server

**Server:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

int serverSocket;

struct sockaddr\_in serverAddress, clientAddress;

char buffer[BUFFER\_SIZE];

// Creating UDP socket

if ((serverSocket = socket(AF\_INET, SOCK\_DGRAM, 0)) == -1) {

perror("Socket creation failed");

exit(EXIT\_FAILURE);

}

memset(&serverAddress, 0, sizeof(serverAddress));

memset(&clientAddress, 0, sizeof(clientAddress));

// Server address configuration

serverAddress.sin\_family = AF\_INET; // IPv4

serverAddress.sin\_addr.s\_addr = INADDR\_ANY;

serverAddress.sin\_port = htons(PORT);

// Binding the socket

if (bind(serverSocket, (const struct sockaddr\*)&serverAddress, sizeof(serverAddress)) == -1) {

perror("Bind failed");

exit(EXIT\_FAILURE);

}

printf("Server listening on port %d...\n", PORT);

while (1) {

socklen\_t len = sizeof(clientAddress);

// Receiving message from the client

ssize\_t dataSize = recvfrom(serverSocket, (char\*)buffer, BUFFER\_SIZE, 0, (struct sockaddr\*)&clientAddress, &len);

buffer[dataSize] = '\0';

printf("Received message from client: %s\n", buffer);

// Sending acknowledgment back to the client

sendto(serverSocket, "Acknowledgment: Message received", strlen("Acknowledgment: Message received"), 0, (const struct sockaddr\*)&clientAddress, len);

}

close(serverSocket);

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

**Client:**

#define PORT 8080

#define SERVER\_IP "127.0.0.1"

#define BUFFER\_SIZE 1024

int main() {

int clientSocket;

struct sockaddr\_in serverAddress;

char buffer[BUFFER\_SIZE];

// Creating UDP socket

if ((clientSocket = socket(AF\_INET, SOCK\_DGRAM, 0)) == -1) {

perror("Socket creation failed");

exit(EXIT\_FAILURE);

}

memset(&serverAddress, 0, sizeof(serverAddress));

// Server address configuration

serverAddress.sin\_family = AF\_INET; // IPv4

serverAddress.sin\_port = htons(PORT);

if (inet\_pton(AF\_INET, SERVER\_IP, &serverAddress.sin\_addr) <= 0) {

perror("Invalid address/ Address not supported");

exit(EXIT\_FAILURE);

}

while (1) {

printf("Enter message for the server (or type 'exit' to quit): ");

fgets(buffer, BUFFER\_SIZE, stdin);

// Sending message to the server

sendto(clientSocket, (const char\*)buffer, strlen(buffer), 0, (const struct sockaddr\*)&serverAddress, sizeof(serverAddress));

if (strcmp(buffer, "exit\n") == 0) {

break;

}

memset(buffer, 0, BUFFER\_SIZE);

// Receiving acknowledgment from the server

ssize\_t dataSize = recvfrom(clientSocket, (char\*)buffer, BUFFER\_SIZE, 0, NULL, NULL);

buffer[dataSize] = '\0';

printf("Server acknowledgment: %s\n", buffer);

}

close(clientSocket);

return 0;

}

## TCP Socket Communication Client and Server

**Server:**

#include<stdio.h>

#include<arpa/inet.h>

#include<string.h>

#include<stdlib.h>

#include<pthread.h>

#define MAX 10

void \*handle\_client(void \*arg) {

int client\_fd = \*((int \*)arg);

char buffer[100];

printf("\nClient connected!");

send(client\_fd, "Hi from Server!", strlen("Hi from Server!"), 0);

recv(client\_fd, buffer, sizeof(buffer),0);

printf("\nClient sent : %s", buffer);

close(client\_fd);

pthread\_exit(NULL);

}

void main() {

printf("\nSERVER INITIALIZING!");

int server\_fd = 0;

int connected\_fd = 0;

struct sockaddr\_in server\_addr = {0};

pthread\_t tid[MAX];

static int i = 0;

int opt = 1;

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0) {

perror("\nServer Socket creation failed!");

return;

}

if (setsockopt(server\_fd, SOL\_SOCKET, SO\_REUSEADDR | SO\_REUSEPORT, &opt, sizeof(opt))) {

perror("\nSocket option set failed!");

return;

}

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(8888);

struct sockaddr\_in client\_addr;

if (bind(server\_fd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {

perror("\nServer Bind failed!");

return;

}

listen(server\_fd, MAX);

int addlen = sizeof(struct sockaddr\_in);

while(1) {

if ((connected\_fd = accept(server\_fd, (struct sockaddr \*)&client\_addr, &addlen)) < 0) {

perror("\nClient connection failed!");

return;

}

pthread\_create(&tid[i], NULL, handle\_client, (void \*)&connected\_fd);

i = (i+1)%MAX;

}

close(server\_fd);

}

**Client:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<arpa/inet.h>

void main() {

int socket\_fd = 0;

struct sockaddr\_in server = {0};

char buf[100];

if ((socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0) {

perror("\nClient socket creation failed!");

return;

}

server.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

server.sin\_port = htons(8888);

server.sin\_family = AF\_INET;

if (connect(socket\_fd, (struct sockaddr\*)&server, sizeof(server)) < 0) {

perror("\nClient to Server connect failed!");

return;

}

send(socket\_fd, "Hi from Client", strlen("Hi from Client"),0);

recv(socket\_fd, &buf, sizeof(buf),0);

printf("\nReceived from Server: %s",buf);

close(socket\_fd);

}

## Shared Memory

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/shm.h>

#include<string.h>

int main()

{

int i;

void \*shared\_memory;

char buff[100];

int shmid;

shmid=shmget((key\_t)2345, 1024, 0666);

shared\_memory=shmat(shmid,NULL,0); //process attached to shared memory segment

printf("Data read from shared memory is : %s\n",(char \*)shared\_memory);

}

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/shm.h>

#include<string.h>

int main()

{

int i;

void \*shared\_memory;

char buff[100];

int shmid;

shmid=shmget((key\_t)2345, 1024, 0666|IPC\_CREAT);

//creates shared memory segment with key 2345, having size 1024 bytes. IPC\_CREAT is used to create the shared segment if it does not exist. 0666 are the permissions on the shared segment

shared\_memory=shmat(shmid,NULL,0);

//process attached to shared memory segment

//this prints the address where the segment is attached with this process

printf("Enter some data to write to shared memory\n");

fgets(buff,100,stdin); //get some input from user

strcpy(shared\_memory,buff); //data written to shared memory

}

## Message Queues

#include<stdio.h>

#include<sys/ipc.h>

#include<sys/msg.h>

typedef struct msg\_buf {

long type;

char msg[100];

}msg\_t;

void main() {

msg\_t ms;

key\_t key = ftok("Allah",123);

int msg\_id = msgget(key, 0666|IPC\_CREAT);

msgrcv(msg\_id,&ms, sizeof(ms),1,0);

printf("%s",ms.msg);

}

#include<stdio.h>

#include<sys/ipc.h>

#include<sys/msg.h>

typedef struct msg\_buffer {

long type;

char buffer[100];

}msg\_t;

void main() {

msg\_t ms;

key\_t key = ftok("Allah",123);

int msg\_id = 0;

msg\_id = msgget(key,0666|IPC\_CREAT);

ms.type = 1;

strncpy(ms.buffer,"HELLO DUDE!", sizeof("HELLO DUDE!"));

msgsnd(msg\_id, &ms, sizeof(ms), 0);

}

## Dequeue

#include <stdio.h>

#include <stdlib.h>

// Node structure of a doubly-linked list

struct Node

{

int data;

struct Node\* prev;

struct Node\* next;

};

// A structure for Deque

struct Deque

{

struct Node\* head;

struct Node\* tail;

int size;

};

// Function prototypes

void initDeque(struct Deque\* dq);

int isEmpty(struct Deque\* dq);

int size(struct Deque\* dq);

void insertFront(struct Deque\* dq, int data);

void insertRear(struct Deque\* dq, int data);

void deleteFront(struct Deque\* dq);

void deleteRear(struct Deque\* dq);

int getFront(struct Deque\* dq);

int getRear(struct Deque\* dq);

// Function to initialize the deque

void initDeque(struct Deque\* dq)

{

dq->head = dq->tail = NULL;

dq->size = 0;

}

// Function to check if the deque is empty

int isEmpty(struct Deque\* dq)

{

return (dq->head == NULL);

}

// Function to get the size of the deque

int size(struct Deque\* dq)

{

return dq->size;

}

// Function to insert an element at the front of the deque

void insertFront(struct Deque\* dq, int data)

{

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL)

{

printf("Memory allocation failed (Overflow)\n");

return;

}

newNode->data = data;

newNode->prev = NULL;

newNode->next = dq->head;

if (dq->head == NULL)

{

dq->tail = newNode;

}

else

{

dq->head->prev = newNode;

}

dq->head = newNode;

dq->size++;

}

// Function to insert an element at the rear of the deque

void insertRear(struct Deque\* dq, int data)

{

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL)

{

printf("Memory allocation failed (Overflow)\n");

return;

}

newNode->data = data;

newNode->prev = dq->tail;

newNode->next = NULL;

if (dq->tail == NULL)

{

dq->head = newNode;

}

else

{

dq->tail->next = newNode;

}

dq->tail = newNode;

dq->size++;

}

// Function to delete an element from the front of the deque

void deleteFront(struct Deque\* dq)

{

if (isEmpty(dq))

{

printf("Underflow\n");

return;

}

struct Node\* temp = dq->head;

dq->head = dq->head->next;

if (dq->head == NULL)

{

dq->tail = NULL;

}

else

{

dq->head->prev = NULL;

}

free(temp);

dq->size--;

}

// Function to delete an element from the rear of the deque

void deleteRear(struct Deque\* dq)

{

if (isEmpty(dq))

{

printf("Underflow\n");

return;

}

struct Node\* temp = dq->tail;

dq->tail = dq->tail->prev;

if (dq->tail == NULL)

{

dq->head = NULL;

}

else

{

dq->tail->next = NULL;

}

free(temp);

dq->size--;

}

// Function to get the front element of the deque

int getFront(struct Deque\* dq)

{

if (isEmpty(dq))

{

return -1;

}

return dq->head->data;

}

// Function to get the rear element of the deque

int getRear(struct Deque\* dq)

{

if (isEmpty(dq))

{

return -1;

}

return dq->tail->data;

}

int main()

{

struct Deque dq;

initDeque(&dq);

printf("Insert element '2' at rear end\n");

insertRear(&dq, 2);

printf("Insert element '0' at rear end\n");

insertRear(&dq, 0);

printf("Rear end element: %d\n", getRear(&dq));

deleteRear(&dq);

printf("After deleting rear element new rear is: %d\n", getRear(&dq));

printf("Inserting element '27' at front end\n");

insertFront(&dq, 27);

printf("Front end element: %d\n", getFront(&dq));

printf("Number of elements in Deque: %d\n", size(&dq));

deleteFront(&dq);

printf("After deleting front element new front is: %d\n", getFront(&dq));

return 0;

}

## Regex Search Target String

#define MAX\_BUFFER\_SIZE 1024

// Function to search for a target string using regex

int searchTargetString(const char \*output, const char \*target) {

regex\_t regex;

int reti;

char buffer[MAX\_BUFFER\_SIZE];

// Compile the regular expression

reti = regcomp(&regex, target, REG\_EXTENDED);

if (reti) {

fprintf(stderr, "Could not compile regex\n");

return -1;

}

// Execute the regular expression

reti = regexec(&regex, output, 0, NULL, 0);

if (!reti) {

printf("Target string found in the output\n");

return 1;

} else if (reti == REG\_NOMATCH) {

printf("Target string not found in the output\n");

return 0;

} else {

regerror(reti, &regex, buffer, sizeof(buffer));

fprintf(stderr, "Regex match failed: %s\n", buffer);

return -1;

}

// Free the compiled regular expression

regfree(&regex);

return 0;

}

int main() {

FILE \*fp;

char buffer[MAX\_BUFFER\_SIZE];

// Command to execute (replace with your network command)

const char \*command = "ping -c 4 google.com";

// Open a pipe to execute the command and read its output

fp = popen(command, "r");

if (fp == NULL) {

fprintf(stderr, "Failed to execute command\n");

return 1;

}

// Read the command output

while (fgets(buffer, sizeof(buffer), fp) != NULL) {

// Search for the target string in the output

if (searchTargetString(buffer, "time=") > 0) {

// Do something when the target string is found

printf("Found target string in the line: %s", buffer);

}

}

// Close the pipe

pclose(fp);

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function to execute a command and capture its output

char\* executeCommand(const char\* command) {

FILE\* fp;

char buffer[128];

char\* result = malloc(4096); // Allocate a buffer for the command output

result[0] = '\0'; // Initialize the result string

// Open the command for reading

fp = popen(command, "r");

if (fp == NULL) {

printf("Failed to run command\n");

exit(1);

}

// Read the output line by line and concatenate it to the result string

while (fgets(buffer, sizeof(buffer) - 1, fp) != NULL) {

strcat(result, buffer);

}

// Close the pipe and return the result

pclose(fp);

return result;

}

// Function to extract the time value from the ping output

int extractPingTime(const char\* pingOutput) {

const char\* timeString = strstr(pingOutput, "time=");

if (timeString != NULL) {

// Extract the time value as an integer

int timeValue;

sscanf(timeString, "time=%d", &timeValue);

return timeValue;

}

return -1; // Return -1 if the time value is not found

}

int main() {

const char\* command = "ping -c 4 google.com"; // Your ping command

char\* pingOutput = executeCommand(command);

int pingTime = extractPingTime(pingOutput);

if (pingTime != -1) {

printf("Ping time: %d ms\n", pingTime);

} else {

printf("Unable to extract ping time\n");

}

// Free the allocated memory

free(pingOutput);

return 0;

}

## Quick Sort

void asc(int \*arr, int start, int size) {

int pivot = start;

int i = start+1;

int j = size;

int temp = 0;

if (start >= size) return;

while (i <= j) {

while ((j >= start) && arr[j] >= arr[pivot]) j--;

while ((i <= size) && arr[i] <= arr[pivot]) i++;

if (i < j) {

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

} else {

temp = arr[pivot];

arr[pivot] = arr[j];

arr[j] = temp;

}

}

asc(arr,0, j-1);

asc(arr,j+1, size);

}

void quick\_sort(int \*arr, int size, void (\*fn)(int\*, int, int)) {

fn(arr, 0, size-1);

printf("\nSorted array is:");

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

printf("%d ",arr[i]);

}

}

## Find repeating elements from two linked lists

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

// Structure to represent a node in the hash table

struct Node {

    int data;

    struct Node\* next;

};

// Function to initialize a hash table

struct Node\*\* initializeHashTable() {

    struct Node\*\* hashTable = (struct Node\*\*)malloc(MAX\_SIZE \* sizeof(struct Node\*));

    for (int i = 0; i < MAX\_SIZE; i++) {

        hashTable[i] = NULL;

    }

    return hashTable;

}

// Function to insert an element into the hash table

void insert(struct Node\*\* hashTable, int key) {

    int index = abs(key) % MAX\_SIZE;

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = key;

    newNode->next = hashTable[index];

    hashTable[index] = newNode;

}

// Function to check if an element exists in the hash table

int contains(struct Node\*\* hashTable, int key) {

    int index = abs(key) % MAX\_SIZE;

    struct Node\* current = hashTable[index];

    while (current != NULL) {

        if (current->data == key) {

            return 1; // Element found

        }

        current = current->next;

    }

    return 0; // Element not found

}

// Function to find repeated elements in two lists

void findRepeats(int\* list1, int size1, int\* list2, int size2) {

    // Initialize hash table

    struct Node\*\* hashTable = initializeHashTable();

    // Insert elements of list1 into hash table

    for (int i = 0; i < size1; i++) {

        insert(hashTable, list1[i]);

    }

    // Check elements of list2 against hash table

    printf("Repeated elements: ");

    for (int i = 0; i < size2; i++) {

        if (contains(hashTable, list2[i])) {

            printf("%d ", list2[i]);

        }

    }

    printf("\n");

    // Free memory allocated for the hash table

    for (int i = 0; i < MAX\_SIZE; i++) {

        struct Node\* current = hashTable[i];

        while (current != NULL) {

            struct Node\* temp = current;

            current = current->next;

            free(temp);

        }

    }

    free(hashTable);

}

int main() {

    int list1[] = {1, 2, 3, 4, 5};

    int list2[] = {3, 4, 5, 6, 7};

    int size1 = sizeof(list1) / sizeof(list1[0]);

    int size2 = sizeof(list2) / sizeof(list2[0]);

    findRepeats(list1, size1, list2, size2);

    return 0;

}

## Merge Sort

void merge\_asc(int \*arr, int start, int middle, int end) {

int n1 = middle-start+1;

int n2 = end-middle;

int left[n1];

int right[n2];

int i = 0;

int j = 0;

int k = 0;

for (i = 0; i < n1; i++) {

left[i] = arr[start+i];

}

for (i = 0; i < n2; i++) {

right[i] = arr[middle + i + 1];

}

i = 0;

k = start;

while (i < n1 && j < n2) {

if (left[i] <= right[j]) {

arr[k] = left[i];

i++;

} else {

arr[k] = right[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = left[i];

i++;

k++;

}

while (j < n2) {

arr[k] = right[j];

j++;

k++;

}

}

void asc(int \*arr, int start, int end) {

if (start < end) {

int middle = (start + end)/2;

asc(arr, start, middle);

asc(arr, middle+1, end);

merge\_asc(arr,start,middle,end);

}

}

## Bloom Filter

#include <stdio.h>

#include <stdbool.h>

#include <math.h>

#define ll long long

// hash 1

int h1(int num, int arrSize)

{

ll int hash = num % arrSize;

return hash;

}

// hash 2

int h2(int num, int arrSize)

{

ll int hash = 1;

while (num > 0)

{

hash = (hash + num % 10);

num /= 10;

}

hash = hash % arrSize;

return hash;

}

// hash 3

int h3(int num, int arrSize)

{

ll int hash = 7;

while (num > 0)

{

hash = (hash \* 31 + num % 10) % arrSize;

num /= 10;

}

return hash;

}

// hash 4

int h4(int num, int arrSize)

{

ll int hash = 3;

int p = 7;

while (num > 0)

{

hash += hash \* 7 + (num % 10) \* pow(p, num % 10);

hash = hash % arrSize;

num /= 10;

} return hash;

}

// lookup operation

bool lookup(int \*bitarray, int arrSize, int num)

{

int a = h1(num, arrSize);

int b = h2(num, arrSize);

int c = h3(num, arrSize);

int d = h4(num, arrSize);

if (bitarray[a] && bitarray[b] && bitarray[c] && bitarray[d])

return true;

else

return false;

}

// insert operation

void insert(int \*bitarray, int arrSize, int num)

{

// check if the element is already present or not

if (lookup(bitarray, arrSize, num))

printf("%d is Probably already present\n", num);

else

{

int a = h1(num, arrSize);

int b = h2(num, arrSize);

int c = h3(num, arrSize);

int d = h4(num, arrSize);

bitarray[a] = 1;

bitarray[b] = 1;

bitarray[c] = 1;

bitarray[d] = 1;

printf("%d inserted\n", num);

}

}

// Driver Code

int main()

{

int bitarray[100] = {0};

int arrSize = 100;

int numArray[33] = {123, 456, 789, 321, 654, 987, 111, 222, 333,

444, 555, 666, 777, 888, 999, 101, 202, 303,

404, 505, 606, 707, 808, 909, 1111, 2222, 3333,

4444, 5555, 6666, 7777, 8888, 9999};

for (int i = 0; i < 33; i++)

{

insert(bitarray, arrSize, numArray[i]);

}

return 0;

}

## Hash Table

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define the size of the hash table

#define TABLE\_SIZE 10

// Define a structure for a node in the linked list

struct Node {

int key;

char data[50];

struct Node\* next;

};

// Define the hash table as an array of linked lists

struct Node\* hashTable[TABLE\_SIZE] = {NULL};

// Hash function

int hash(int key) {

return key % TABLE\_SIZE;

}

// Insert a key-value pair into the hash table

void insert(int key, const char\* data) {

// Create a new node

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL) {

fprintf(stderr, "Memory allocation failed.\n");

exit(1);

}

// Set the key and data for the new node

newNode->key = key;

strncpy(newNode->data, data, sizeof(newNode->data));

newNode->next = NULL;

// Get the hash index

int index = hash(key);

// Insert the new node at the beginning of the linked list

newNode->next = hashTable[index];

hashTable[index] = newNode;

}

// Search for a key in the hash table

const char\* search(int key) {

// Get the hash index

int index = hash(key);

// Traverse the linked list at the hash index

struct Node\* current = hashTable[index];

while (current != NULL) {

if (current->key == key) {

return current->data; // Key found

}

current = current->next;

}

return NULL; // Key not found

}

// Print the contents of the hash table

void printHashTable() {

for (int i = 0; i < TABLE\_SIZE; i++) {

printf("Bucket %d: ", i);

struct Node\* current = hashTable[i];

while (current != NULL) {

printf("(%d, %s) ", current->key, current->data);

current = current->next;

}

printf("\n");

}

}

// Main function for testing the hash table

int main() {

// Insert key-value pairs

insert(1, "John");

insert(2, "Alice");

insert(11, "Bob");

insert(12, "Charlie");

insert(21, "David");

// Print the initial hash table

printf("Initial Hash Table:\n");

printHashTable();

// Search for keys

printf("\nSearch results:\n");

printf("Key 2: %s\n", search(2));

printf("Key 12: %s\n", search(12));

printf("Key 5: %s\n", search(5));

return 0;

}

## Endianess

#include<stdio.h>

#if 0

void print\_bytes(void \*ptr, int size)

{

    unsigned char \*p = ptr;

    int i;

    for (i=0; i<size; i++) {

        printf("%02hhX ", p[i]);

    }

    printf("\n");

}

#endif

//12345678 -> 0th byte 12 --> big endian, 0th byte 78 --> little endian(lowest address - least significant byte)

void main() {

    int data = 12345678;

    char \*d = (char\*)&data;

    printf("\n %p",&d);

    if (\*d == 78)

        printf("\nLittle Endian");

    else

        printf("\nBig Endian");

    d = (char\*)&data+3;

    printf("\n%x",\*d);

    //print\_bytes(&data,sizeof(int));

}

#if 0

#include <stdio.h>

/\* function to show bytes in memory, from location start to start+n\*/

void show\_mem\_rep(char \*start, int n)

{

    int i;

    printf("\nn = %d",n);

    for (i = 0; i < n; i++)

        printf("%d %.2x", i,start[i]);

    printf("\n");

}

/\*Main function to call above function for 0x01234567\*/

int main()

{

int i = 0x01234567;

show\_mem\_rep((char \*)&i, sizeof(i));

return 0;

}

#endif

#include<stdio.h>

//12345678 -> 0th byte 12 --> little endian, 0th byte 78 --> big endian

void main() {

int data = 12345678;

char \*d = (char\*)&data;

printf("\n %u",\*d);

if (\*d == 78)

printf("\nLittle Endian");

else

printf("\nBig Endian");

}

## MALLOC Implementation

#include <stdio.h>

#include <stdlib.h>

#define BUFFER\_SIZE 1000 // Adjust the size as needed

// Global buffer to simulate memory allocation

char buffer[BUFFER\_SIZE];

int next\_free\_offset = 0;

void\* my\_malloc(size\_t size) {

if (next\_free\_offset + size <= BUFFER\_SIZE) {

void\* allocated\_memory = buffer + next\_free\_offset;

next\_free\_offset += size;

return allocated\_memory;

} else {

fprintf(stderr, "Error: Insufficient memory in the fixed buffer.\n");

return NULL;

}

}

void my\_free(void\* ptr) {

// In a simple fixed buffer scenario, freeing memory can be a no-op

// or could be used to reset the allocation pointer.

// For demonstration purposes, we'll reset the allocation pointer here.

next\_free\_offset = (int)((char\*)ptr - buffer);

}

int main() {

// Example usage

int\* int\_ptr = (int\*)my\_malloc(sizeof(int));

if (int\_ptr != NULL) {

\*int\_ptr = 42;

printf("Allocated integer value: %d\n", \*int\_ptr);

// Freeing the allocated memory

my\_free(int\_ptr);

}

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <pthread.h>

#define MEMORY\_POOL\_SIZE 1000

// Block structure to represent allocated memory blocks

typedef struct Block {

size\_t size;

struct Block\* next;

} Block;

// Memory pool

static char memory\_pool[MEMORY\_POOL\_SIZE];

// Mutex for thread safety

static pthread\_mutex\_t memory\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

// Pointer to the first block in the memory pool

static Block\* memory\_head = NULL;

// Function to initialize the memory manager

void initialize\_memory\_manager() {

memory\_head = (Block\*)memory\_pool;

memory\_head->size = MEMORY\_POOL\_SIZE - sizeof(Block);

memory\_head->next = NULL;

}

// Function to allocate memory

void\* my\_malloc(size\_t size) {

pthread\_mutex\_lock(&memory\_mutex);

if (memory\_head == NULL) {

initialize\_memory\_manager();

}

size\_t total\_size = size + sizeof(Block);

Block\* current\_block = memory\_head;

Block\* previous\_block = NULL;

while (current\_block != NULL) {

if (current\_block->size >= total\_size) {

// Allocate from this block

if (current\_block->size > total\_size + sizeof(Block)) {

// Split the block

Block\* new\_block = (Block\*)((char\*)current\_block + total\_size);

new\_block->size = current\_block->size - total\_size;

new\_block->next = current\_block->next;

current\_block->size = size;

current\_block->next = new\_block;

} else {

// Use the entire block

total\_size = current\_block->size;

}

// Update the linked list

if (previous\_block == NULL) {

memory\_head = current\_block->next;

} else {

previous\_block->next = current\_block->next;

}

pthread\_mutex\_unlock(&memory\_mutex);

return (char\*)current\_block + sizeof(Block);

}

previous\_block = current\_block;

current\_block = current\_block->next;

}

pthread\_mutex\_unlock(&memory\_mutex);

fprintf(stderr, "Error: Insufficient memory.\n");

return NULL;

}

// Function to free memory

void my\_free(void\* ptr) {

pthread\_mutex\_lock(&memory\_mutex);

if (ptr != NULL) {

Block\* block = (Block\*)((char\*)ptr - sizeof(Block));

block->next = memory\_head;

memory\_head = block;

}

pthread\_mutex\_unlock(&memory\_mutex);

}

// Function to print memory blocks for debugging

void print\_memory\_blocks() {

Block\* current\_block = memory\_head;

while (current\_block != NULL) {

printf("Block: %p, Size: %zu\n", (void\*)current\_block, current\_block->size);

current\_block = current\_block->next;

}

}

int main() {

// Example usage

initialize\_memory\_manager();

int\* int\_ptr = (int\*)my\_malloc(sizeof(int));

if (int\_ptr != NULL) {

\*int\_ptr = 42;

printf("Allocated integer value: %d\n", \*int\_ptr);

print\_memory\_blocks();

// Freeing the allocated memory

my\_free(int\_ptr);

print\_memory\_blocks();

}

return 0;

}

## Single Linked List

#include<stdio.h>

#include<stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head;

struct node\* allocate() {

struct node \*new\_node = (struct node \*)malloc(sizeof(struct node));

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

}

return new\_node;

}

void main() {

printf("\nSingle Linked List!");

int choice = 0;

int position = 0;

int count = 0;

int value = 0;

struct node \*new\_node = NULL;

while (1) {

printf("\n1.Create\n2.Insert at beginning\n3.Insert at end\n4.Insert at position\n5.Delete from first\n6.Delete at end\n7.Delete at position\n8.Search\n9.Print\n10.End");

printf("\nSelect a choice:");

scanf("%d",&choice);

switch (choice) {

case 1:

printf("\nCreate!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

new\_node->next = NULL;

head = new\_node;

}

break;

case 2:

printf("\nInsert at beginning!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

new\_node->next = head;

head = new\_node;

}

break;

case 3:

printf("\nInsert at end!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

struct node\* cur = head;

while (cur->next != NULL) {

cur = cur->next;

}

cur->next = new\_node;

new\_node->next = NULL;

}

break;

case 4:

printf("\nInsert at position!");

printf("\nTell me a position!");

scanf("%d",&position);

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

struct node\* curr = head;

while (curr != NULL) {

count++;

if (position-1 > count) {

curr = curr->next;

} else {

break;

}

}

new\_node->next = curr->next;

curr->next = new\_node;

}

break;

case 5:

printf("\nDelete from first!");

if (head == NULL) {

printf("\nList empty!");

}

struct node \*ele = head;

struct node \*next = head->next;

free(ele);

ele = NULL;

head = next;

break;

case 6:

printf("\nDelete at end!");

struct node\* h = head;

struct node\* p = head;

if (h == NULL) {

printf("\nList is empty!");

}

if (h->next == NULL) {

free(h);

h = NULL;

head = NULL;

}

else {

while (h->next != NULL) {

p = h;

h = h->next;

}

p->next = NULL;

free(h);

}

break;

case 7:

count = 0;

printf("\nDelete at position!");

printf("\nEnter position to delete from");

scanf("%d",&position);

struct node\* cr = head;

struct node\* pr = head;

while (cr != NULL) {

count++;

pr = cr;

if (count == position-1) {

pr->next = pr->next->next;

free(cr->next);

cr->next = NULL;

break;

}

cr = cr->next;

}

break;

case 8:

printf("\nSearch!");

printf("\nEnter the value to search!");

scanf("%d",&value);

count = 0;

struct node\* c = head;

while (c != NULL) {

count++;

if (c->data == value) {

printf("\nValue %d present in %d position",value,count);

break;

}

if (c->next == NULL) {

printf("\nValue %d not found!",value);

}

c = c->next;

}

break;

case 9:

printf("\nList!");

struct node\* current = head;

printf("\n");

while (current != NULL) {

printf("%d->",current->data);

current = current->next;

}

printf("NULL");

break;

case 10:

printf("\nThank you!");

exit(1);

break;

default:

printf("\nTry again!");

break;

}

choice = 0;

}

}

## Double Linked List

#include<stdio.h>

#include<stdlib.h>

struct node {

int data;

struct node \*prev;

struct node \*next;

};

struct node \*head;

struct node\* allocate() {

struct node \*new\_node = (struct node \*)malloc(sizeof(struct node));

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

}

return new\_node;

}

void main() {

printf("\nSingle Linked List!");

int choice = 0;

int position = 0;

int count = 0;

int value = 0;

struct node \*new\_node = NULL;

while (1) {

printf("\n1.Create\n2.Insert at beginning\n3.Insert at end\n4.Insert at position\n5.Delete from first\n6.Delete at end\n7.Delete at position\n8.Search\n9.Print\n10.Reverse Print\n11.End");

printf("\nSelect a choice:");

scanf("%d",&choice);

switch (choice) {

case 1:

printf("\nCreate!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

new\_node->prev = NULL;

new\_node->next = NULL;

head = new\_node;

}

break;

case 2:

printf("\nInsert at beginning!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

new\_node->prev = NULL;

new\_node->next = head;

head->prev = new\_node;

head = new\_node;

}

break;

case 3:

printf("\nInsert at end!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

struct node\* cur = head;

while (cur->next != NULL) {

cur = cur->next;

}

cur->next = new\_node;

new\_node->prev = cur;

new\_node->next = NULL;

}

break;

case 4:

printf("\nInsert at position!");

printf("\nTell me a position!");

scanf("%d",&position);

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

struct node\* curr = head;

while (curr != NULL) {

count++;

if (position-1 > count) {

curr = curr->next;

} else {

break;

}

}

new\_node->next = curr->next;

curr->next->prev = new\_node;

new\_node->prev = curr;

curr->next = new\_node;

}

break;

case 5:

printf("\nDelete from first!");

if (head == NULL) {

printf("\nList empty!");

}

struct node \*ele = head;

struct node \*next = head->next;

free(ele);

ele = NULL;

next->prev = NULL;

head = next;

break;

case 6:

printf("\nDelete at end!");

struct node\* h = head;

struct node\* p = head;

if (h == NULL) {

printf("\nList is empty!");

}

if (h->next == NULL) {

free(h);

h = NULL;

head = NULL;

}

else {

while (h->next != NULL) {

p = h;

h = h->next;

}

p->next = NULL;

free(h);

}

break;

case 7:

count = 0;

printf("\nDelete at position!");

printf("\nEnter position to delete from");

scanf("%d",&position);

struct node\* cr = head;

struct node\* nx = NULL;

while (cr != NULL) {

count++;

//1 2 3 4 5

if (count == position-1) {

nx = cr->next;

cr->next->next->prev = cr;

cr->next = cr->next->next;

free(nx);

nx->next = NULL;

break;

}

cr = cr->next;

}

break;

case 8:

printf("\nSearch!");

printf("\nEnter the value to search!");

scanf("%d",&value);

count = 0;

struct node\* c = head;

while (c != NULL) {

count++;

if (c->data == value) {

printf("\nValue %d present in %d position",value,count);

break;

}

if (c->next == NULL) {

printf("\nValue %d not found!",value);

}

c = c->next;

}

break;

case 9:

printf("\nList!");

struct node\* current = head;

printf("\n");

while (current != NULL) {

printf("%d->",current->data);

current = current->next;

}

printf("NULL");

break;

case 10:

printf("\nList reverse print!");

struct node\* currnt = head;

printf("\n");

while (currnt->next != NULL) {

currnt = currnt->next;

}

struct node\* end = currnt;

while (end != NULL) {

printf("%d->",end->data);

end = end->prev;

}

printf("NULL");

case 11:

printf("\nThank you!");

exit(1);

break;

default:

printf("\nTry again!");

break;

}

choice = 0;

}

}

## Stack

#include<stdio.h>

#include<stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head;

struct node\* allocate() {

struct node \*new\_node = (struct node \*)malloc(sizeof(struct node));

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

}

return new\_node;

}

void main() {

printf("\nStack!");

int choice = 0;

int position = 0;

int count = 0;

int value = 0;

struct node \*new\_node = NULL;

while (1) {

printf("\n1.Create\n2.Push\n3.Pop\n4.Print\n5.End");

printf("\nSelect a choice:");

scanf("%d",&choice);

switch (choice) {

case 1:

printf("\nCreate!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

new\_node->next = NULL;

head = new\_node;

}

break;

case 2:

printf("\nPush!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

new\_node->next = head;

head = new\_node;

}

break;

case 3:

printf("\nPop!");

if (head == NULL) {

printf("\nList empty!");

}

struct node \*ele = head;

struct node \*next = head->next;

free(ele);

ele = NULL;

head = next;

break;

case 4:

printf("\nStack Contents!");

struct node\* current = head;

printf("\n");

while (current != NULL) {

printf("%d->",current->data);

current = current->next;

}

printf("NULL");

break;

case 5:

printf("\nThank you!");

exit(1);

break;

default:

printf("\nTry again!");

break;

}

choice = 0;

}

}

## Queue

#include<stdio.h>

#include<stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head;

struct node\* allocate() {

struct node \*new\_node = (struct node \*)malloc(sizeof(struct node));

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

}

return new\_node;

}

void main() {

printf("\nQueue!");

int choice = 0;

int position = 0;

int count = 0;

int value = 0;

struct node \*new\_node = NULL;

while (1) {

printf("\n1.Create\n2.Enqueue\n3.Dequeue\n4.Print\n5.End");

printf("\nSelect a choice:");

scanf("%d",&choice);

switch (choice) {

case 1:

printf("\nCreate!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

new\_node->next = NULL;

head = new\_node;

}

break;

case 2:

printf("\nEnqueue!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

struct node\* cur = head;

while (cur->next != NULL) {

cur = cur->next;

}

cur->next = new\_node;

new\_node->next = NULL;

}

break;

case 3:

printf("\nDequeue!");

if (head == NULL) {

printf("\nList empty!");

}

struct node \*ele = head;

struct node \*next = head->next;

free(ele);

ele = NULL;

head = next;

break;

case 4:

printf("\nList!");

struct node\* current = head;

printf("\n");

while (current != NULL) {

printf("%d->",current->data);

current = current->next;

}

printf("NULL");

break;

case 5:

printf("\nThank you!");

exit(1);

break;

default:

printf("\nTry again!");

break;

}

choice = 0;

}

}

## Binary Search Tree

#include<stdio.h>

#include<stdlib.h>

struct node {

int data;

struct node \*left;

struct node \*right;

};

struct node \*root;

struct node\* allocate() {

struct node \*new\_node = (struct node \*)malloc(sizeof(struct node));

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

}

return new\_node;

}

struct node\* insert(struct node \*temp, struct node \*root) {

if (root == NULL) {

root = temp;

} else {

if (temp->data < root->data) {

root->left = insert(temp,root->left);

} else if (temp->data > root->data){

root->right = insert(temp,root->right);

}

}

return root;

}

void print\_inorder(struct node \*t) {

if (t == NULL) return;

if (t != NULL) {

print\_inorder(t->left);

printf("%d:",t->data);

print\_inorder(t->right);

}

}

void print\_preorder(struct node \*t) {

if (t == NULL) return;

if (t != NULL) {

printf("%d:",t->data);

print\_preorder(t->left);

print\_preorder(t->right);

}

}

void print\_postorder(struct node \*t) {

if (t == NULL) return;

if (t != NULL) {

print\_postorder(t->left);

print\_postorder(t->right);

printf("%d:",t->data);

}

}

void search(struct node \*root, int value) {

if (root == NULL) {

printf("\nValue not found!");

return;

}

else {

if (root->data == value) {

printf("\nValue found!");

return;

} else {

if (value < root->data)

search(root->left,value);

else

search(root->right,value);

}

}

}

struct node \*delete(struct node\*root, int value) {

if (root == NULL) {

printf("\nValue not found to be deleted!");

return NULL;

}

if (value < root->data) {

root->left = delete(root->left, value);

} else if (value > root->data) {

root->right = delete(root->right, value);

} else {

if (root->left == NULL && root->right == NULL) {

free(root);

return NULL;

} else if (root->left != NULL && root->right == NULL) {

struct node \* y = root->left;

free(root);

return y;

} else if (root->right != NULL && root->left == NULL) {

struct node \*z = root->right;

free(root);

return z;

} else {

struct node \*succ\_parent = root;

struct node \*succ = root->right;

while (succ->left != NULL) {

succ\_parent = succ;

succ = succ->left;

}

if (succ\_parent == root) {

root->right = succ->right;

} else {

root->left = succ->right;

}

root->data = succ->data;

free(succ);

}

}

return root;

}

void main() {

printf("\nBinary Search Tree!");

int choice = 0;

int position = 0;

int count = 0;

int value = 0;

struct node \*new\_node = NULL;

struct node \*nn = NULL;

while (1) {

printf("\n1.Create\n2.Insert\n3.Delete\n4.Search\n5.Print Inorder\n6.Print PreOrder\n7.Print PostOrder\n6..End");

printf("\nSelect a choice:");

scanf("%d",&choice);

switch (choice) {

case 1:

printf("\nCreate!");

new\_node = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&new\_node->data);

new\_node->left = NULL;

new\_node->right = NULL;

root = new\_node;

}

break;

case 2:

printf("\nInsert!");

nn = allocate();

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

printf("\nEnter the value for node!");

scanf("%d",&nn->data);

insert(nn,root);

}

break;

case 3:

printf("\nDelete!");

printf("\nEnter value to delete!");

scanf("%d",&value);

delete(root,value);

break;

case 4:

printf("\nSearch!");

printf("\nEnter the value to search!");

scanf("%d",&value);

search(root,value);

break;

case 5:

printf("\nBST Inorder!");

printf("\n");

print\_inorder(root);

break;

case 6:

printf("\nBST Preorder!");

printf("\n");

print\_preorder(root);

break;

case 7:

printf("\nBST Postorder!");

printf("\n");

print\_postorder(root);

break;

case 8:

printf("\nThank you!");

exit(1);

break;

default:

printf("\nTry again!");

break;

}

choice = 0;

}

}

## Trie Complete Code

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

#include<string.h>

#define N 26

struct node {

char data;

struct node \*children[N];

int is\_leaf;

};

struct node \*root;

struct node\* allocate(char data) {

struct node \*new\_node = (struct node \*)malloc(sizeof(struct node));

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

}

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

new\_node->children[i] = NULL;

}

new\_node->data = data;

new\_node->is\_leaf = 0;

return new\_node;

}

struct node\* insert(struct node \*root, char \*input) {

struct node \*temp = root;

int position = 0;

printf("\nWord input is :%s",input);

for (int i = 0; input[i] != '\0'; i++) {

position = (int)input[i] - 'a';

printf("\nPosition is :%d",position);

if (temp->children[position] == NULL) {

temp->children[position] = allocate(input[i]);

printf("\nCharacter : %c",temp->data);

} else {

//Do nothing as character is already present

}

temp = temp->children[position];

}

temp->is\_leaf = 1;

return root;

}

void print(struct node\* root) {

// Prints the nodes of the trie

if (!root)

return;

struct node\* temp = root;

printf("%c -> ", temp->data);

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

print(temp->children[i]);

}

}

void search(struct node \*root, char \*word) {

struct node \*temp = root;

int position = 0;

for (int i = 0; word[i] != '\0'; i++) {

position = (int)word[i] - 'a';

if (temp->children[position] == NULL) {

printf("\nWord not found");

return;

} else {

temp = temp->children[position];

}

}

if (temp->is\_leaf == 1) {

printf("\nWord found!");

} else {

printf("\nIncomplete word found!");

}

}

bool is\_complete\_word(char \*word) {

struct node \*temp = root;

int position = 0;

for (int i = 0; word[i] != '\0'; i++) {

position = (int)word[i] - 'a';

if (temp->children[position] != NULL) {

temp = temp->children[position];

}

}

return (temp->is\_leaf == 1) ? true:false;

}

int check\_divergence(struct node \*root, char \*prefix) {

struct node \*temp = root;

int c\_index = 0;

for (int i= 0; i < strlen(prefix); i++) {

int position = prefix[i] - 'a';

if (temp->children[position]) {

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

if (j != position && temp->children[j]) {

c\_index = i;

break;

}

}

temp = temp->children[position];

}

}

return c\_index;

}

char\* common\_prefix (struct node \*root, char \*word) {

char \*prefix = (char \*)malloc((1 + strlen(word))\*sizeof(char));

strcpy(prefix, word);

int index = check\_divergence(root, prefix);

if (index >= 0) {

prefix[index] = '\0';

prefix = (char \*)realloc(prefix, (1+index) \* sizeof(char));

}

return prefix;

}

void delete(struct node \*root, char \*word) {

printf("\nDeletion in progress!");

if (!is\_complete\_word(word)) {

printf("\nIncomplete word cant be deleted!");

return;

}

struct node \*temp = root;

char\* cp = common\_prefix(root, word);

int i = 0;

for (i = 0; cp[i] != '\0'; i++) {

int position = (int)cp[i] - 'a';

if (temp->children[position] != NULL) {

temp = temp->children[position];

} else {

return;

}

}

int len = strlen(word);

for (; i < len; i++) {

int position = (int)word[i] - 'a';

if (temp->children[position] != NULL) {

struct node\* rem = temp->children[position];

temp->children[position] = NULL;

free(rem);

}

}

}

void main() {

printf("\nTrie!");

int choice = 0;

int position = 0;

int count = 0;

int value = 0;

struct node \*new\_node = NULL;

char input[100];

while (1) {

printf("\n1.Create\n2.Insert Word\n3.Delete Word\n4.Search Word\n5.Print Dictionary\n6.End");

printf("\nSelect a choice:");

scanf("%d",&choice);

switch (choice) {

case 1:

printf("\nCreate!");

new\_node = allocate('\0');

if (new\_node == NULL) {

printf("\nMalloc failure!");

exit(1);

} else {

root = new\_node;

}

break;

case 2:

printf("\nInsert Word!");

printf("\nEnter a word to add to dictionary!");

scanf("%s",input);

insert(root, input);

break;

case 3:

printf("\nDelete Word!");

printf("\nEnter a word to delete from dictionary!");

scanf("%s",input);

delete(root, input);

break;

case 4:

printf("\nSearch Word!");

printf("\nEnter a word to search in dictionary!");

scanf("%s",input);

search(root, input);

break;

case 5:

printf("\nPrint Dictionary!");

print(root);

break;

case 6:

printf("\nThank you!");

exit(1);

break;

default:

printf("\nTry again!");

break;

}

choice = 0;

}

}

## Graph Shortest Path

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

typedef struct {

    int \*\*adj;

    int n;

    int \*cost;

    bool \*visited;

} Graph;

int minCost(int \*cost, bool \*visited, int n) {

    int min = INT\_MAX, min\_node;

    for (int i = 0; i < n; i++) {

        if (visited[i] == false && cost[i] <= min) {

            min = cost[i];

            min\_node = i;

        }   else {

            printf("\nNOP");

        }

    }

    printf("\nmin cost = %d and min node = %d", min,min\_node);

    return min\_node;

}

Graph\* graphCreate(int n, int\*\* edges, int edgesSize, int\* edgesColSize) {

    Graph \*obj = malloc(sizeof(Graph));

    obj->n = n;

    obj->adj = (int \*\*)calloc(n, sizeof(int \*));

    obj->cost = (int \*)calloc(n, sizeof(int));

    obj->visited = (bool \*)calloc(n, sizeof(bool));

    for (int i = 0; i < n; i++) {

        obj->adj[i] = (int \*)calloc(n, sizeof(int));

    }

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

        graphAddEdge(obj, edges[i], edgesColSize[i]);

    }

    return obj;

}

void graphAddEdge(Graph\* obj, int\* edge, int edgeSize) {

    obj->adj[edge[0]][edge[1]] = edge[2];

}

int graphShortestPath(Graph\* obj, int node1, int node2) {

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

        obj->cost[i] = INT\_MAX,

        obj->visited[i] = false;

    }

    obj->cost[node1] = 0;

    /\* Djikstra's Algorithm \*/

    for (int i = 0; i < obj->n - 1; i++) {

        printf("\ni = %d",i);

        int src = minCost(obj->cost, obj->visited, obj->n);

        printf("\nsrc = %d",src);

        obj->visited[src] = true;

        for (int dst = 0; dst < obj->n; dst++)

            if (obj->visited[dst] == false && obj->adj[src][dst]

                && obj->cost[src] != INT\_MAX

                && obj->cost[src] + obj->adj[src][dst] < obj->cost[dst]) {

                obj->cost[dst] = obj->cost[src] + obj->adj[src][dst];

                printf("\nobj->cost[%d] = %d obj->cost[%d] = %d",src,obj->cost[src],dst,obj->cost[dst]);

            } else {

                printf("\nElse");

            }

    }

    return obj->cost[node2] == INT\_MAX ? -1 : obj->cost[node2];

}

void graphFree(Graph\* obj) {

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

        free(obj->adj[i]);

    }

    free(obj->adj);

    free(obj->cost);

    free(obj->visited);

    free(obj);

}

int main() {

    int n = 4;

    int edgesSize = 4;

    int edgesColSize[] = {3, 3, 3, 3};

    int\*\* edges = (int\*\*)malloc(edgesSize \* sizeof(int\*));

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

        edges[i] = (int\*)malloc(edgesColSize[i] \* sizeof(int));

    }

    edges[0][0] = 0;

    edges[0][1] = 2;

    edges[0][2] = 5;

    edges[1][0] = 0;

    edges[1][1] = 1;

    edges[1][2] = 2;

    edges[2][0] = 1;

    edges[2][1] = 2;

    edges[2][2] = 1;

    edges[3][0] = 3;

    edges[3][1] = 0;

    edges[3][2] = 3;

    Graph\* obj = graphCreate(n, edges, edgesSize, edgesColSize);

    printf("\nShortet path from 3->2 is %d\n", graphShortestPath(obj, 3, 2)); // 6

    printf("\nShortest path from 0->3 is %d\n", graphShortestPath(obj, 0, 3)); // -1

    int newEdge[] = {1, 3, 4};

    graphAddEdge(obj, newEdge, 3);

    printf("\nShortest path from 0->3 is: %d\n", graphShortestPath(obj, 0, 3)); // 6

    graphFree(obj);

    return 0;

}

Time/Space Complexity: O(n2)/O(1)s

## Semaphore

#include <stdio.h>

#include <pthread.h>

typedef struct {

pthread\_mutex\_t mutex;

pthread\_cond\_t condition;

int value;

} semaphore;

void semaphore\_init(semaphore\* sem, int initial\_value) {

pthread\_mutex\_init(&(sem->mutex), NULL);

pthread\_cond\_init(&(sem->condition), NULL);

sem->value = initial\_value;

}

void semaphore\_wait(semaphore\* sem) {

pthread\_mutex\_lock(&(sem->mutex));

while (sem->value <= 0) {

pthread\_cond\_wait(&(sem->condition), &(sem->mutex));

}

sem->value--;

pthread\_mutex\_unlock(&(sem->mutex));

}

void semaphore\_signal(semaphore\* sem) {

pthread\_mutex\_lock(&(sem->mutex));

sem->value++;

pthread\_cond\_signal(&(sem->condition));

pthread\_mutex\_unlock(&(sem->mutex));

}

void\* thread\_function(void\* arg) {

semaphore\* sem = (semaphore\*)arg;

printf("Thread waiting\n");

semaphore\_wait(sem);

printf("Thread acquired the semaphore\n");

// Perform some critical section or shared resource access here

printf("Thread releasing the semaphore\n");

semaphore\_signal(sem);

return NULL;

}

int main() {

semaphore sem;

semaphore\_init(&sem, 1); // Initialize the semaphore with initial value 1

pthread\_t thread;

pthread\_create(&thread, NULL, thread\_function, (void\*)&sem);

// Main thread also performs some work

printf("Main thread performing some work\n");

// Main thread waits for the semaphore

printf("Main thread waiting\n");

semaphore\_wait(&sem);

printf("Main thread acquired the semaphore\n");

// Perform some critical section or shared resource access here

printf("Main thread releasing the semaphore\n");

semaphore\_signal(&sem);

pthread\_join(thread, NULL);

return 0;

}

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t mutex;

void\* thread(void\* arg)

{

//wait

sem\_wait(&mutex);

printf("\nEntered..\n");

//critical section

sleep(4);

//signal

printf("\nJust Exiting...\n");

sem\_post(&mutex);

}

int main()

{

sem\_init(&mutex, 0, 1);

pthread\_t t1,t2;

pthread\_create(&t1,NULL,thread,NULL);

sleep(2);

pthread\_create(&t2,NULL,thread,NULL);

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

sem\_destroy(&mutex);

return 0;

}

## Mutex

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

pthread\_t tid[2];

int counter;

pthread\_mutex\_t lock;

void\* trythis(void\* arg)

{

pthread\_mutex\_lock(&lock);

unsigned long i = 0;

counter += 1;

printf("\n Job %d has started\n", counter);

for (i = 0; i < (0xFFFFFFFF); i++)

;

printf("\n Job %d has finished\n", counter);

pthread\_mutex\_unlock(&lock);

return NULL;

}

int main(void)

{

int i = 0;

int error;

if (pthread\_mutex\_init(&lock, NULL) != 0) {

printf("\n mutex init has failed\n");

return 1;

}

while (i < 2) {

error = pthread\_create(&(tid[i]),

NULL,

&trythis, NULL);

if (error != 0)

printf("\nThread can't be created :[%s]",

strerror(error));

i++;

}

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

pthread\_mutex\_destroy(&lock);

return 0;

}

## Blocking Queue

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define QUEUE\_SIZE 5

typedef struct {

    int\* buffer;

    int size;

    int front;

    int rear;

    pthread\_mutex\_t mutex;            // Mutex for mutual exclusion

    pthread\_cond\_t not\_empty;        // Condition variable for not empty

    pthread\_cond\_t not\_full;         // Condition variable for not full

} CircularQueue;

// Initialize the circular queue

void initializeQueue(CircularQueue\* queue, int size) {

    queue->buffer = (int\*)malloc(sizeof(int) \* size);

    queue->size = size;

    queue->front = 0;

    queue->rear = 0;

    pthread\_mutex\_init(&queue->mutex, NULL);

    pthread\_cond\_init(&queue->not\_empty, NULL);

    pthread\_cond\_init(&queue->not\_full, NULL);

}

// Enqueue an item into the circular queue

void enqueue(CircularQueue\* queue, int item) {

    pthread\_mutex\_lock(&queue->mutex);

    while ((queue->rear + 1) % queue->size == queue->front) {

        // Queue is full, wait for space

        pthread\_cond\_wait(&queue->not\_full, &queue->mutex);

    }

    queue->buffer[queue->rear] = item;

    queue->rear = (queue->rear + 1) % queue->size;

    pthread\_cond\_signal(&queue->not\_empty);  // Notify that the queue is not empty

    pthread\_mutex\_unlock(&queue->mutex);

}

// Dequeue an item from the circular queue

int dequeue(CircularQueue\* queue) {

    pthread\_mutex\_lock(&queue->mutex);

    while (queue->front == queue->rear) {

        // Queue is empty, wait for items

        pthread\_cond\_wait(&queue->not\_empty, &queue->mutex);

    }

    int item = queue->buffer[queue->front];

    queue->front = (queue->front + 1) % queue->size;

    pthread\_cond\_signal(&queue->not\_full);  // Notify that the queue is not full

    pthread\_mutex\_unlock(&queue->mutex);

    return item;

}

// Clean up the circular queue

void cleanupQueue(CircularQueue\* queue) {

    free(queue->buffer);

    pthread\_mutex\_destroy(&queue->mutex);

    pthread\_cond\_destroy(&queue->not\_empty);

    pthread\_cond\_destroy(&queue->not\_full);

}

// Example usage

void\* producer(void\* arg) {

    CircularQueue\* queue = (CircularQueue\*)arg;

    for (int i = 1; i <= 10; ++i) {

        enqueue(queue, i);

        printf("Produced: %d\n", i);

    }

    pthread\_exit(NULL);

}

void\* consumer(void\* arg) {

    CircularQueue\* queue = (CircularQueue\*)arg;

    for (int i = 0; i < 10; ++i) {

        int item = dequeue(queue);

        printf("Consumed: %d\n", item);

    }

    pthread\_exit(NULL);

}

int main() {

    CircularQueue queue;

    initializeQueue(&queue, QUEUE\_SIZE);

    pthread\_t producerThread, consumerThread;

    pthread\_create(&producerThread, NULL, producer, &queue);

    pthread\_create(&consumerThread, NULL, consumer, &queue);

    pthread\_join(producerThread, NULL);

    pthread\_join(consumerThread, NULL);

    cleanupQueue(&queue);

    return 0;

}

## String to Integer(atoi)

int myAtoi(char \* s){

        int i = 0;

        int j = strlen(s)-1;

        int result = 0;

        int sign = 1;

        if (isalpha(s[i])) {

            //printf("\n1");

            return 0;

        }

        while (s[i] == ' ') {

                //printf("\n2");

                i++;

        }

        if (s[i] == '+') {

                //printf("\n3");

                sign = 1;

                i++;

        } else if (s[i] == '-'){

                //printf("\n4");

                sign = -1;

                i++;

        } else {

            sign = 1;

        }

        while (i <= j) {

            //printf("\n5");

            if (isdigit(s[i])) {

                //printf("\n6");

                if ((result > INT\_MAX/10) ||

                    (result == INT\_MAX/10) && (s[i]-'0') > INT\_MAX%10) {

                     printf("\na");

            return (sign == -1)? INT\_MIN:INT\_MAX;

                }

                result = 10\*result + (s[i]-'0');

                //printf("\nres : %d",result);

                i++;

            } else {

                //printf("\n7");

                i++;

                break;

            }

        }

        //printf("\n8");

        result = result\*sign;

        return result;

}

## Add Two Numbers

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     struct ListNode \*next;

 \* };

 \*/

 // Function to create a new node with a given value

struct ListNode\* newNode(int val) {

    struct ListNode\* node = (struct ListNode\*)malloc(sizeof(struct ListNode));

    node->val = val;

    node->next = NULL;

    return node;

}

struct ListNode\* addTwoNumbers(struct ListNode\* l1, struct ListNode\* l2) {

    struct ListNode\* dummyHead = (struct ListNode\*)malloc(sizeof(struct ListNode));

    struct ListNode\* current = dummyHead;

    int carry = 0;

    while (l1 != NULL || l2 != NULL) {

        int x = (l1 != NULL) ? l1->val : 0;

        int y = (l2 != NULL) ? l2->val : 0;

        int sum = x + y + carry;

        carry = sum / 10;

        current->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

        current->next->val = sum % 10;

        current->next->next = NULL;

        current = current->next;

        if (l1 != NULL) l1 = l1->next;

        if (l2 != NULL) l2 = l2->next;

    }

    if (carry > 0) {

        current->next = (struct ListNode\*)malloc(sizeof(struct ListNode));

        current->next->val = carry;

        current->next->next = NULL;

    }

    struct ListNode\* result = dummyHead->next;

    free(dummyHead);

    return result;

}

## Next Permutation

#include <stdio.h>

// Function to swap two elements in an array

void swap(int\* nums, int i, int j) {

    int temp = nums[i];

    nums[i] = nums[j];

    nums[j] = temp;

}

// Function to reverse a portion of an array

void reverse(int\* nums, int start, int end) {

    int i = start, j = end;

    while (i < j) {

        swap(nums, i, j);

        i++;

        j--;

    }

}

// Function to find the next permutation

void nextPermutation(int\* nums, int numsSize) {

    //First find the lesser element in the array starting from last

    int i = numsSize - 2;

    while (i >= 0 && nums[i + 1] <= nums[i]) {

        i--;

    }

    //Then find the element which is greater than that element

    if (i >= 0) {

        int j = numsSize - 1;

        while (nums[j] <= nums[i]) {

            j--;

        }

        //Swap those numbers

        swap(nums, i, j);

    }

    //Reverse the numbers from that index to the found index (ascending order)

    reverse(nums, i + 1, numsSize - 1);

}

public class Solution {

public void nextPermutation(int[] nums) {

int i = nums.length - 2;

while (i >= 0 && nums[i + 1] <= nums[i]) {

i--;

}

if (i >= 0) {

int j = nums.length - 1;

while (nums[j] <= nums[i]) {

j--;

}

swap(nums, i, j);

}

reverse(nums, i + 1);

}

private void reverse(int[] nums, int start) {

int i = start, j = nums.length - 1;

while (i < j) {

swap(nums, i, j);

i++;

j--;

}

}

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

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

**General Permutation**

#include <stdio.h>

// Function to swap two elements in an array

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to generate permutations

void generatePermutations(int arr[], int size, int index) {

if (index == size - 1) {

// Print the current permutation

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

printf("%d ", arr[i]);

}

printf("\n");

return;

}

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

// Swap the current element with the element at index

swap(&arr[index], &arr[i]);

// Recursively generate permutations for the remaining elements

generatePermutations(arr, size, index + 1);

// Undo the swap to backtrack

swap(&arr[index], &arr[i]);

}

}

int main() {

// Example usage

int arr[] = {1, 2, 3};

int size = sizeof(arr) / sizeof(arr[0]);

printf("Permutations:\n");

generatePermutations(arr, size, 0);

return 0;

}

## First Missing Positive

int firstMissingPositive(int\* nums, int numsSize) {

    // Create a boolean array to mark the presence of positive numbers

    bool present[numsSize + 1];

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

        present[i] = false;

    }

    // Mark positive numbers in the array

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

        if (nums[i] > 0 && nums[i] <= numsSize) {

            present[nums[i]] = true;

        }

    }

    // Find the first missing positive number

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

        if (!present[i]) {

            return i;

        }

    }

    // If all positive numbers are present, return the next positive number

    return numsSize + 1;

}

## Permutations

/\*\*

 \* Return an array of arrays of size \*returnSize.

 \* The sizes of the arrays are returned as \*returnColumnSizes array.

 \* Note: Both returned array and \*columnSizes array must be malloced, assume caller calls free().

 \*/

// Function to swap two elements in an array

void swap(int\* a, int\* b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

// Function to generate permutations

void generatePermutations(int\* nums, int numsSize, int index, int\*\* result, int\* returnSize, int\* returnColumnSizes) {

    if (index == numsSize - 1) {

        // Allocate memory for the current permutation

        result[\*returnSize] = (int\*)malloc(numsSize \* sizeof(int));

        // Copy the current permutation to the result array

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

            result[\*returnSize][i] = nums[i];

        }

        // Update returnColumnSizes with the size of the current permutation

        returnColumnSizes[\*returnSize] = numsSize;

        // Increment returnSize to keep track of the number of permutations

        (\*returnSize)++;

        return;

    }

    for (int i = index; i < numsSize; i++) {

        // Swap the current element with the element at index

        swap(&nums[index], &nums[i]);

        // Recursively generate permutations for the remaining elements

        generatePermutations(nums, numsSize, index + 1, result, returnSize, returnColumnSizes);

        // Undo the swap to backtrack

        swap(&nums[index], &nums[i]);

    }

}

int\*\* permute(int\* nums, int numsSize, int\* returnSize, int\*\* returnColumnSizes) {

    // Calculate the number of permutations (n!)

    int totalPermutations = 1;

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

        totalPermutations \*= i;

    }

    // Allocate memory for the result array

    int\*\* result = (int\*\*)malloc(totalPermutations \* sizeof(int\*));

    // Allocate memory for returnColumnSizes array

    \*returnColumnSizes = (int\*)malloc(totalPermutations \* sizeof(int));

    // Initialize returnSize

    \*returnSize = 0;

    // Generate permutations

    generatePermutations(nums, numsSize, 0, result, returnSize, \*returnColumnSizes);

    return result;

}

## Remove Duplicates From Sorted List

Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list****sorted****as well*.

**Example 1:**

A diagram of a flowchart

Description automatically generated

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

**Output:** [1,2]

**Example 2:**

A diagram of a diagram

Description automatically generated

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

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

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     struct ListNode \*next;

 \* };

 \*/

// Function to delete duplicates from a sorted linked list

struct ListNode\* deleteDuplicates(struct ListNode\* head) {

    struct ListNode\* current = head;

    // Traverse the list

    while (current != NULL && current->next != NULL) {

        // Check if the current and next nodes have the same value

        if (current->val == current->next->val) {

            // Duplicate found, remove the next node

            struct ListNode\* duplicate = current->next;

            current->next = current->next->next;

            free(duplicate);

        } else {

            // Move to the next node

            current = current->next;

        }

    }

    return head;

}

// Function to print the linked list

void printList(struct ListNode\* head) {

    struct ListNode\* current = head;

    while (current != NULL) {

        printf("%d -> ", current->val);

        current = current->next;

    }

    printf("NULL\n");

}

// Function to create a new node with the given value

struct ListNode\* newNode(int val) {

    struct ListNode\* node = (struct ListNode\*)malloc(sizeof(struct ListNode));

    node->val = val;

    node->next = NULL;

    return node;

}

## Binary Tree InOrder Traversal

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

// Function to perform inorder traversal of a binary tree

void inorderTraversalHelper(struct TreeNode\* root, int\* result, int\* index) {

    if (root == NULL) {

        return;

    }

    // Traverse the left subtree

    inorderTraversalHelper(root->left, result, index);

    // Visit the current node

    result[(\*index)++] = root->val;

    // Traverse the right subtree

    inorderTraversalHelper(root->right, result, index);

}

// Function to get the size of a binary tree

int getSize(struct TreeNode\* root) {

    if (root == NULL) {

        return 0;

    }

    return 1 + getSize(root->left) + getSize(root->right);

}

// Function to perform inorder traversal and return the result as an array

int\* inorderTraversal(struct TreeNode\* root, int\* returnSize) {

    // Get the size of the binary tree

    \*returnSize = getSize(root);

    // Allocate memory for the result array

    int\* result = (int\*)malloc((\*returnSize) \* sizeof(int));

    // Index to keep track of the position in the result array

    int index = 0;

    // Perform inorder traversal

    inorderTraversalHelper(root, result, &index);

    return result;

}

## Binary Tree ZigZag Level Order Traversal

class Solution {

    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

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

        if (root == null) return levels;

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

        queue.add(root);

        int level = 0;

        while ( !queue.isEmpty() ) {

            // start the current level

            levels.add(new ArrayList<Integer>());

            // number of elements in the current level

            int level\_length = queue.size();

            for(int i = 0; i < level\_length; ++i) {

                TreeNode node = queue.remove();

                // fulfill the current level

                levels.get(level).add(node.val);

                // add child nodes of the current level

                // in the queue for the next level

                if (node.left != null) queue.add(node.left);

                if (node.right != null) queue.add(node.right);

            }

            // go to next level

            level++;

        }

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

            if (i % 2 == 1) {

                Collections.reverse(levels.get(i));

            }

        }

        return levels;

    }

}

## Flatten Binary Tree To Linked List

Given the root of a binary tree, flatten the tree into a "linked list":

* The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.
* The "linked list" should be in the same order as a [**pre-order traversal**](https://en.wikipedia.org/wiki/Tree_traversal#Pre-order,_NLR) of the binary tree.

**Example 1:**

A diagram of a network

Description automatically generated

**Input:** root = [1,2,5,3,4,null,6]

**Output:** [1,null,2,null,3,null,4,null,5,null,6]

**Example 2:**

**Input:** root = []

**Output:** []

**Example 3:**

**Input:** root = [0]

**Output:** [0]

void flatten(struct TreeNode\* root) {

    if (root == NULL) {

        return;

    }

    struct TreeNode\* current = root;

    while (current != NULL) {

        // If the current node has a left child

        if (current->left != NULL) {

            // Find the rightmost node in the left subtree

            struct TreeNode\* rightmost = current->left;

            while (rightmost->right != NULL) {

                rightmost = rightmost->right;

            }

            // Move the right subtree to the rightmost node's right child

            rightmost->right = current->right;

            // Move the left subtree to the right child

            current->right = current->left;

            current->left = NULL;

        }

        // Move to the next node in the flattened tree

        current = current->right;

    }

}

// Function to create a new binary tree node with the given value

struct TreeNode\* newNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

// Function to print the flattened linked list

void printFlattenedList(struct TreeNode\* root) {

    struct TreeNode\* current = root;

    while (current != NULL) {

        printf("%d -> ", current->val);

        current = current->right;

    }

    printf("NULL\n");

}

## Binary Tree Post Order Traversal

// Function to get the size of a binary tree

int getSize(struct TreeNode\* root) {

    if (root == NULL) {

        return 0;

    }

    return 1 + getSize(root->left) + getSize(root->right);

}

// Function to perform postorder traversal of a binary tree

void postorderTraversalHelper(struct TreeNode\* root, int\* result, int\* index) {

    if (root == NULL) {

        return;

    }

    // Traverse the left subtree

    postorderTraversalHelper(root->left, result, index);

    // Traverse the right subtree

    postorderTraversalHelper(root->right, result, index);

    // Visit the current node

    result[(\*index)++] = root->val;

}

// Function to perform postorder traversal and return the result as an array

int\* postorderTraversal(struct TreeNode\* root, int\* returnSize) {

    // Get the size of the binary tree

    \*returnSize = getSize(root);

    // Allocate memory for the result array

    int\* result = (int\*)malloc((\*returnSize) \* sizeof(int));

    // Index to keep track of the position in the result array

    int index = 0;

    // Perform postorder traversal

    postorderTraversalHelper(root, result, &index);

    return result;

}

## Binary Tree Pre Order Traversal

int getSize(struct TreeNode\* root) {

    if (root == NULL) {

        return 0;

    }

    return 1 + getSize(root->left) + getSize(root->right);

}

// Function to perform preorder traversal of a binary tree

void preorderTraversalHelper(struct TreeNode\* root, int\* result, int\* index) {

    if (root == NULL) {

        return;

    }

    // Visit the current node

    result[(\*index)++] = root->val;

    // Traverse the left subtree

    preorderTraversalHelper(root->left, result, index);

    // Traverse the right subtree

    preorderTraversalHelper(root->right, result, index);

}

// Function to perform preorder traversal and return the result as an array

int\* preorderTraversal(struct TreeNode\* root, int\* returnSize) {

    // Get the size of the binary tree

    \*returnSize = getSize(root);

    // Allocate memory for the result array

    int\* result = (int\*)malloc((\*returnSize) \* sizeof(int));

    // Index to keep track of the position in the result array

    int index = 0;

    // Perform preorder traversal

    preorderTraversalHelper(root, result, &index);

    return result;

}

// Function to create a new binary tree node with the given value

struct TreeNode\* newNode(int val) {

    struct TreeNode\* node = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

    node->val = val;

    node->left = NULL;

    node->right = NULL;

    return node;

}

## Excel Sheet Column Number

int titleToNumber(char\* columnTitle) {

    int result = 0;

    int length = strlen(columnTitle);

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

        result += pow(26, (length - 1 - i)) \* (columnTitle[i] - 'A' + 1);

    }

    return result;

}

class Solution {

public int titleToNumber(String s) {

int result = 0;

int n = s.length();

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

result = result \* 26;

// In Java, subtracting characters is subtracting ASCII values of characters

result += (s.charAt(i) - 'A' + 1);

}

return result;

}

}

## Ugly Number

bool isUgly(int n) {

       if (n <= 0) {

        return false;

    }

    while (n % 2 == 0) {

        n /= 2;

    }

    while (n % 3 == 0) {

        n /= 3;

    }

    while (n % 5 == 0) {

        n /= 5;

    }

    return n == 1;

}

## Game Of Life

#define DIE 2

#define LIVE 3

int countLive(int i, int j, int\*\* board, int rows, int cols) {

    int count = 0;

    int dirs[8][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}, {1, 1}, {1, -1}, {-1, 1}, {-1, -1}};

    for (int k = 0; k < 8; k++) {

        int x = i + dirs[k][0];

        int y = j + dirs[k][1];

        if (x >= 0 && y >= 0 && x < rows && y < cols) {

            if (board[x][y] == 1 || board[x][y] == DIE)

                count++;

        }

    }

    return count;

}

void gameOfLife(int\*\* board, int boardSize, int\* boardColSize) {

    if (!board || boardSize == 0 || !boardColSize || \*boardColSize == 0)

        return;

    int rows = boardSize;

    int cols = \*boardColSize;

    for (int i = 0; i < rows; i++) {

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

            int around = countLive(i, j, board, rows, cols);

            if (board[i][j] == 0 && around == 3)

                board[i][j] = LIVE;

            else if (board[i][j] == 1) {

                if (around == 2 || around == 3)

                    continue;

                if (around < 2 || around > 3)

                    board[i][j] = DIE;

            }

        }

    }

    for (int i = 0; i < rows; i++) {

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

            if (board[i][j] == DIE)

                board[i][j] = 0;

            if (board[i][j] == LIVE)

                board[i][j] = 1;

        }

    }

}

## Minimum Number Of Arrows To Burst Balloons

// Comparison function for qsort

int compareIntervals(const void\* a, const void\* b) {

    int\* intervalA = \*(int\*\*)a;

    int\* intervalB = \*(int\*\*)b;

    if (intervalA[1] == intervalB[1]) {

        return 0;

    } else if (intervalA[1] < intervalB[1]) {

        return -1;

    } else {

        return 1;

    }

}

int findMinArrowShots(int\*\* points, int pointsSize, int\* pointsColSize) {

    if (pointsSize == 0) {

        return 0;

    }

    // Sort the intervals based on their end points

    qsort(points, pointsSize, sizeof(int\*), compareIntervals);

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

        printf("\n%d %d",points[i][0], points[i][1]);

    }

    int arrows = 1;

    int xStart, xEnd, firstEnd = points[0][1];

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

        xStart = points[i][0];

        xEnd = points[i][1];

        // If the current balloon starts after the end of another one,

        // one needs one more arrow

        if (firstEnd < xStart) {

            arrows++;

            firstEnd = xEnd;

        }

    }

    return arrows;

}

## Exclusive Time Of Functions

public class Solution {

    public int[] exclusiveTime(int n, List<String> logs) {

        // separate time to several intervals, add interval to their function

        int[] res = new int[n];

        //store id, not timestamp

        Stack<Integer> idStack = new Stack<>();

        //store timestamp: prevLogTime here is to maintain previous functions time

        // and if it is start function we just add it to previous if it is an end function

        // then add plus one to mark the end of the top of stack function

        //(It's used to record last log time)

        int prevLogTime = 0;

        // pre means the start of the interval

        for (String s : logs) {

            String[] log = s.split(":");

            if (log[1].equals("start")) {

                if (!idStack.isEmpty()) res[idStack.peek()] += Integer.parseInt(log[2]) - prevLogTime;

                // log[2] is the start of next interval, doesn't belong to current interval.

                idStack.push(Integer.parseInt(log[0]));

                prevLogTime = Integer.parseInt(log[2]);

            } else {

                res[idStack.pop()] += Integer.parseInt(log[2]) - prevLogTime + 1;

                // log[2] is end of current interval, belong to current interval. That's why we have +1 here

                prevLogTime = Integer.parseInt(log[2]) + 1;

                // prevLogTime means the start of next interval, so we need to +1

            }

        }

        return res;

    }

}

## Top K Frequent Words

class Solution {

    public List<String> topKFrequent(String[] words, int k) {

        HashMap<String,Integer> hm = new HashMap<String,Integer>();

        for (String s:words) {

            hm.put(s, hm.getOrDefault(s,0)+1);

        }

    Queue<Map.Entry<String, Integer>> pq = new PriorityQueue<>(

        (a, b) -> a.getValue().equals(b.getValue()) ? a.getKey().compareTo(b.   getKey ()) : b.getValue() - a.getValue()

);

        for(HashMap.Entry<String,Integer> entry: hm.entrySet()) {

            pq.add(entry);

        }

        System.out.println(pq.toString());

        List<String> l = new ArrayList<String>();

        int i = 0;

        while (i < k) {

            l.add(pq.poll().getKey());

            System.out.println("Now"+pq.toString());

            i++;

        }

        return l;

    }

}

## Reverse String In Place

#include <stdio.h>

#include <string.h>

void reverseString(char \*str) {

    int length = strlen(str);

    int i = 0, j = length - 1;

    char temp;

    // Swap characters from the beginning and end of the string using a while loop

    while (i < j) {

        // Swap str[i] and str[j]

        temp = str[i];

        str[i] = str[j];

        str[j] = temp;

        // Move to the next pair of characters

        i++;

        j--;

    }

}

int main() {

    char str[] = "Hello, World!";

    printf("Original string: %s\n", str);

    reverseString(str);

    printf("Reversed string: %s\n", str);

    return 0;

}

## Bitwise AND Range of Numbers

#include <stdio.h>

// Original function

int rangeBitwiseAnd(int left, int right) {

    int shift = 0;

    // find the common 1-bits

    while (left < right) {

        left >>= 1;

        right >>= 1;

        ++shift;

    }

    return left << shift;

}

// Alternative function

int rangeBitwiseAndAlternative(int left, int right) {

    while (left < right) {

        // turn off rightmost 1-bit

        right = right & (right - 1);

    }

    return left & right;

}

int main() {

    int left = 5;

    int right = 7;

    printf("Original function result: %d\n", rangeBitwiseAnd(left, right));

    printf("Alternative function result: %d\n", rangeBitwiseAndAlternative(left, right));

    return 0;

}

## Walking Robot Simulation

#include <stdlib.h>

#include <math.h>

// Define a structure to represent a point

struct Point {

    int x;

    int y;

};

// Function to calculate the squared Euclidean distance

int calculateDistanceSquared(struct Point p) {

    return p.x \* p.x + p.y \* p.y;

}

int robotSim(int\* commands, int commandsSize, int\*\* obstacles, int obstaclesSize, int\* obstaclesColSize) {

    // Define the four possible directions: North, East, South, West

    int directions[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    // Initialize the current position and direction of the robot

    struct Point currentPosition = {0, 0};

    int currentDirection = 0; // 0 represents North

    // Create a set to store obstacle positions for quick lookup

    int\*\* obstacleSet = (int\*\*)malloc(obstaclesSize \* sizeof(int\*));

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

        obstacleSet[i] = (int\*)malloc(2 \* sizeof(int));

        obstacleSet[i][0] = obstacles[i][0];

        obstacleSet[i][1] = obstacles[i][1];

    }

    // Initialize the result (maximum squared distance)

    int result = 0;

    // Iterate through the commands

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

        if (commands[i] == -1) {

            // Turn right 90 degrees

            currentDirection = (currentDirection + 1) % 4;

        } else if (commands[i] == -2) {

            // Turn left 90 degrees

            currentDirection = (currentDirection + 3) % 4;

        } else {

            // Move forward k units

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

                struct Point nextPosition = {currentPosition.x + directions[currentDirection][0],

                                             currentPosition.y + directions[currentDirection][1]};

                // Check for obstacles

                int obstacleDetected = 0;

                for (int k = 0; k < obstaclesSize; k++) {

                    if (nextPosition.x == obstacleSet[k][0] && nextPosition.y == obstacleSet[k][1]) {

                        obstacleDetected = 1;

                        break;

                    }

                }

                // Update position if no obstacle is detected

                if (!obstacleDetected) {

                    currentPosition = nextPosition;

                    result = fmax(result, calculateDistanceSquared(currentPosition));

                } else {

                    // Break out of the loop if an obstacle is detected

                    break;

                }

            }

        }

    }

    // Free memory allocated for the obstacle set

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

        free(obstacleSet[i]);

    }

    free(obstacleSet);

    return result;

}

## Cousins In Binary Tree

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     struct TreeNode \*left;

 \*     struct TreeNode \*right;

 \* };

 \*/

;

// Structure to represent a queue node for level-order traversal

struct QueueNode {

    struct TreeNode\* node;

    int depth;

    int parent;

};

// Queue functions for level-order traversal

struct Queue {

    int front, rear;

    int capacity;

    struct QueueNode\* array;

};

struct Queue\* createQueue(int capacity) {

    struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

    queue->capacity = capacity;

    queue->front = queue->rear = -1;

    queue->array = (struct QueueNode\*)malloc(capacity \* sizeof(struct QueueNode));

    return queue;

}

void enqueue(struct Queue\* queue, struct TreeNode\* node, int depth, int parent) {

    struct QueueNode qNode = {node, depth, parent};

    if (queue->front == -1) {

        queue->front = queue->rear = 0;

    } else {

        queue->rear = (queue->rear + 1) % queue->capacity;

    }

    queue->array[queue->rear] = qNode;

}

struct QueueNode dequeue(struct Queue\* queue) {

    struct QueueNode qNode = queue->array[queue->front];

    if (queue->front == queue->rear) {

        queue->front = queue->rear = -1;

    } else {

        queue->front = (queue->front + 1) % queue->capacity;

    }

    return qNode;

}

bool isCousins(struct TreeNode\* root, int x, int y) {

    struct Queue\* queue = createQueue(1000); // Assuming a maximum of 1000 nodes in the tree

    enqueue(queue, root, 0, -1); // Root has depth 0 and no parent

    int parentX = -1, parentY = -1;

    int depthX = -1, depthY = -1;

    while (queue->front != -1) {

        int levelSize = queue->rear - queue->front + 1;

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

            struct QueueNode qNode = dequeue(queue);

            struct TreeNode\* node = qNode.node;

            int depth = qNode.depth;

            int parent = qNode.parent;

            if (node->val == x) {

                parentX = parent;

                depthX = depth;

            } else if (node->val == y) {

                parentY = parent;

                depthY = depth;

            }

            if (node->left) {

                enqueue(queue, node->left, depth + 1, node->val);

            }

            if (node->right) {

                enqueue(queue, node->right, depth + 1, node->val);

            }

        }

        // Check if both nodes are found at the same level

        if ((parentX != -1 && parentY != -1) && (depthX == depthY) && (parentX != parentY)) {

            return true;

        }

        // Reset variables for the next level

        parentX = parentY = -1;

        depthX = depthY = -1;

    }

    return false;

}

## Maximum Equal Frequency

class Solution {

    public int maxEqualFreq(int[] nums) {

        int maxLen = 0;

        int n = nums.length;

        HashMap<Integer, Integer> freqMap = new HashMap<>();

        HashMap<Integer, Integer> countMap = new HashMap<>();

        for (int i = 0; i < n; i++) {

            int num = nums[i];

            // Update the frequency count for the current number

            freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);

            // Update the countMap based on the frequency count

            int freq = freqMap.get(num);

            int count = countMap.getOrDefault(freq, 0);

            countMap.put(freq, count + 1);

            // Check if removing one element makes all frequencies equal

            if (count \* freq == i || (count - 1) \* (freq - 1) == i) {

                maxLen = i + 1;

            }

        }

        return maxLen;

    }

}

## Packet Parser Multithreading

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

// Define packet structures (Ethernet, IP, TCP, Application)

typedef struct {

// Ethernet header fields

// ...

// IP header fields

// ...

} EthernetPacket;

typedef struct {

// TCP header fields

// ...

} TCPPacket;

typedef struct {

// Application layer payload

// ...

} ApplicationPacket;

// Packet structure containing all headers

typedef struct {

EthernetPacket ethernet;

TCPPacket tcp;

ApplicationPacket app;

} FullPacket;

// Function to process a packet

void processPacket(FullPacket\* packet) {

// Process packet headers and payload

// ...

// Example: Print packet details

printf("Processed packet\n");

}

// Function to simulate packet reception

FullPacket generateRandomPacket() {

// Simulate packet creation with random data

FullPacket packet;

// ...

return packet;

}

// Thread function for packet processing

void\* processThread(void\* arg) {

for (;;) {

FullPacket packet = generateRandomPacket();

processPacket(&packet);

}

return NULL;

}

int main() {

// Create threads for packet processing

pthread\_t threads[2];

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

if (pthread\_create(&threads[i], NULL, processThread, NULL) != 0) {

fprintf(stderr, "Error creating thread\n");

exit(EXIT\_FAILURE);

}

}

// Wait for threads to finish (this is a simple example, in a real scenario, threads may run indefinitely)

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

pthread\_join(threads[i], NULL);

}

return 0;

}

#include <stdio.h>

#include <stdint.h>

#include <stdlib.h>

#include <pthread.h>

// Define Ethernet, IPv4, TCP, and application header structures as before

struct EthernetHeader {

uint8\_t destMAC[6];

uint8\_t srcMAC[6];

uint16\_t etherType;

};

struct IPv4Header {

uint8\_t versionIHL;

uint8\_t dscpECN;

uint16\_t totalLength;

uint16\_t identification;

uint16\_t flagsFragmentOffset;

uint8\_t timeToLive;

uint8\_t protocol;

uint16\_t headerChecksum;

uint32\_t srcIP;

uint32\_t destIP;

};

struct TCPHeader {

uint16\_t srcPort;

uint16\_t destPort;

uint32\_t sequenceNumber;

uint32\_t ackNumber;

uint8\_t dataOffsetReserved;

uint8\_t flags;

uint16\_t windowSize;

uint16\_t checksum;

uint16\_t urgentPointer;

};

// Define a structure to hold frame data

struct FrameData {

uint8\_t\* frame;

uint16\_t frameLength;

};

void\* processFrame(void\* arg);

int main() {

// Example Ethernet frames (add more frames as needed)

uint8\_t frame1[] = { /\* Frame 1 data \*/ };

uint8\_t frame2[] = { /\* Frame 2 data \*/ };

uint8\_t frame[] = {

// Ethernet header

0x00, 0x11, 0x22, 0x33, 0x44, 0x55, // Source MAC

0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, // Destination MAC

0x08, 0x00, // EtherType (IPv4)

// IPv4 header

0x45, 0x00, 0x00, 0x3C, 0x12, 0x34, 0x40, 0x00, 0x40, 0x06, 0x00, 0x00,

0xC0, 0xA8, 0x01, 0x01, // Source IP: 192.168.1.1

0xC0, 0xA8, 0x01, 0x02, // Destination IP: 192.168.1.2

// TCP header

0x00, 0x50, 0x00, 0x37, 0xB8, 0xA0, 0x00, 0x00, 0x00, 0x00, 0xA0, 0x02,

0xFA, 0xF0, 0x00, 0x00, 0x02, 0x04, 0x05, 0xB4, 0x01, 0x03, 0x03, 0x06,

// Application data (HTTP request)

0x47, 0x45, 0x54, 0x20, 0x2F, 0x20, 0x48, 0x54, 0x54, 0x50, 0x2F, 0x31,

0x2E, 0x31, 0x0D, 0x0A, 0x48, 0x6F, 0x73, 0x74, 0x3A, 0x20, 0x77, 0x77,

// ... (HTTP request continues)

};

pthread\_t thread1, thread2;

// Create a FrameData structure for each frame

struct FrameData data1 = {frame1, sizeof(frame1)};

struct FrameData data2 = {frame2, sizeof(frame2)};

// ...

// Create threads to process frames concurrently

pthread\_create(&thread1, NULL, processFrame, (void\*)&data1);

pthread\_create(&thread2, NULL, processFrame, (void\*)&data2);

// ...

// Wait for threads to finish

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

// ...

return 0;

}

void\* processFrame(void\* arg) {

struct FrameData\* frameData = (struct FrameData\*)arg;

processEthernetFrame(frameData->frame, frameData->frameLength);

pthread\_exit(NULL);

}

void processEthernetFrame(const uint8\_t\* frame, uint16\_t frameLength) {

if (frameLength < sizeof(struct EthernetHeader)) {

printf("Invalid Ethernet frame\n");

return;

}

struct EthernetHeader\* ethHeader = (struct EthernetHeader\*)frame;

printf("Ethernet Frame - Source MAC: ");

printMACAddress(ethHeader->srcMAC);

printf(", Destination MAC: ");

printMACAddress(ethHeader->destMAC);

printf(", EtherType: %04x\n", ntohs(ethHeader->etherType));

// Assuming Ethernet frame contains IPv4 packet

if (ntohs(ethHeader->etherType) == 0x0800) {

processIPv4Packet(frame + sizeof(struct EthernetHeader), frameLength - sizeof(struct EthernetHeader));

} else {

printf("Unsupported EtherType: %04x\n", ntohs(ethHeader->etherType));

}

}

void processIPv4Packet(const uint8\_t\* packet, uint16\_t packetLength) {

if (packetLength < sizeof(struct IPv4Header)) {

printf("Invalid IPv4 packet\n");

return;

}

struct IPv4Header\* ipv4Header = (struct IPv4Header\*)packet;

printf("IPv4 Packet - Source IP: %s, Destination IP: %s\n",

inet\_ntoa(\*(struct in\_addr\*)&ipv4Header->srcIP), inet\_ntoa(\*(struct in\_addr\*)&ipv4Header->destIP));

// Assuming IPv4 packet contains TCP header

processTCPHeader(packet + (ipv4Header->versionIHL & 0x0F) \* 4, packetLength - (ipv4Header->versionIHL & 0x0F) \* 4);

}

void processTCPHeader(const uint8\_t\* tcpHeader, uint16\_t tcpHeaderLength) {

if (tcpHeaderLength < sizeof(struct TCPHeader)) {

printf("Invalid TCP header\n");

return;

}

struct TCPHeader\* tcpHeaderStruct = (struct TCPHeader\*)tcpHeader;

printf("TCP Header - Source Port: %u, Destination Port: %u\n",

ntohs(tcpHeaderStruct->srcPort), ntohs(tcpHeaderStruct->destPort));

// Assuming TCP header contains application data

processApplicationData(tcpHeader + (tcpHeaderStruct->dataOffsetReserved >> 4) \* 4, tcpHeaderLength - (tcpHeaderStruct->dataOffsetReserved >> 4) \* 4);

}

void processApplicationData(const uint8\_t\* data, uint16\_t dataLength) {

// Add code to process application-specific data

printf("Application Data (%u bytes):\n", dataLength);

for (uint16\_t i = 0; i < dataLength; ++i) {

printf("%c", data[i]);

}

printf("\n");

}

## Thread Pool

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define THREAD\_POOL\_SIZE 4

// Structure to hold task information

typedef struct {

void (\*task)(void\*); // Function pointer for the task

void\* arg; // Argument for the task

} Task;

// Structure to represent the thread pool

typedef struct {

pthread\_t threads[THREAD\_POOL\_SIZE]; // Array of threads

Task taskQueue[THREAD\_POOL\_SIZE \* 2]; // Task queue (adjust size as needed)

int queueSize; // Current size of the task queue

int front; // Front of the queue

int rear; // Rear of the queue

pthread\_mutex\_t mutex; // Mutex for synchronization

pthread\_cond\_t condition; // Condition variable for synchronization

} ThreadPool;

// Function to initialize the thread pool

void initializeThreadPool(ThreadPool\* pool) {

pool->queueSize = 0;

pool->front = 0;

pool->rear = 0;

pthread\_mutex\_init(&pool->mutex, NULL);

pthread\_cond\_init(&pool->condition, NULL);

// Create threads in the pool

for (int i = 0; i < THREAD\_POOL\_SIZE; ++i) {

pthread\_create(&pool->threads[i], NULL, (void\* (\*)(void\*))NULL, NULL);

}

}

// Function to add a task to the thread pool

void addTask(ThreadPool\* pool, void (\*task)(void\*), void\* arg) {

pthread\_mutex\_lock(&pool->mutex);

// Wait if the task queue is full

while (pool->queueSize == sizeof(pool->taskQueue) / sizeof(pool->taskQueue[0])) {

pthread\_cond\_wait(&pool->condition, &pool->mutex);

}

// Add the task to the queue

pool->taskQueue[pool->rear].task = task;

pool->taskQueue[pool->rear].arg = arg;

pool->rear = (pool->rear + 1) % (sizeof(pool->taskQueue) / sizeof(pool->taskQueue[0]));

pool->queueSize++;

// Signal a waiting thread that a new task is available

pthread\_cond\_signal(&pool->condition);

pthread\_mutex\_unlock(&pool->mutex);

}

// Function executed by worker threads

void\* workerThread(void\* arg) {

ThreadPool\* pool = (ThreadPool\*)arg;

for (;;) {

pthread\_mutex\_lock(&pool->mutex);

// Wait if the task queue is empty

while (pool->queueSize == 0) {

pthread\_cond\_wait(&pool->condition, &pool->mutex);

}

// Retrieve a task from the queue

void (\*task)(void\*) = pool->taskQueue[pool->front].task;

void\* taskArg = pool->taskQueue[pool->front].arg;

pool->front = (pool->front + 1) % (sizeof(pool->taskQueue) / sizeof(pool->taskQueue[0]));

pool->queueSize--;

// Signal that there is space in the task queue

pthread\_cond\_signal(&pool->condition);

pthread\_mutex\_unlock(&pool->mutex);

// Execute the task

task(taskArg);

}

return NULL;

}

// Function to destroy the thread pool

void destroyThreadPool(ThreadPool\* pool) {

// Cancel all threads in the pool

for (int i = 0; i < THREAD\_POOL\_SIZE; ++i) {

pthread\_cancel(pool->threads[i]);

}

// Destroy the mutex and condition variable

pthread\_mutex\_destroy(&pool->mutex);

pthread\_cond\_destroy(&pool->condition);

}

// Sample task to be executed by the thread pool

void sampleTask(void\* arg) {

int taskNumber = \*(int\*)arg;

printf("Task %d executed by thread %lu\n", taskNumber, pthread\_self());

free(arg); // Free the memory allocated for the argument

}

int main() {

ThreadPool pool;

initializeThreadPool(&pool);

// Submit tasks to the thread pool

for (int i = 0; i < 10; ++i) {

int\* taskNumber = malloc(sizeof(int)); // Allocate memory for the task argument

\*taskNumber = i;

addTask(&pool, sampleTask, taskNumber);

}

// Sleep to allow the tasks to be executed

sleep(5);

// Destroy the thread pool

destroyThreadPool(&pool);

return 0;

}

## Memory Pool

#include <stdio.h>

#include <stdlib.h>

#define PACKET\_SIZE 1500 // Adjust the size according to your needs

#define POOL\_SIZE 1000 // Adjust the number of packets in the pool

// Structure representing a packet

typedef struct {

char data[PACKET\_SIZE];

// Add other packet-related fields as needed

} Packet;

// Structure representing a memory pool

typedef struct {

Packet\* packets[POOL\_SIZE];

int nextIndex; // Index to track the next available packet

} MemoryPool;

// Initialize the memory pool

void initializeMemoryPool(MemoryPool\* pool) {

for (int i = 0; i < POOL\_SIZE; ++i) {

pool->packets[i] = (Packet\*)malloc(sizeof(Packet));

// Initialize other packet-related fields as needed

}

pool->nextIndex = 0;

}

// Get a packet from the memory pool

Packet\* allocatePacket(MemoryPool\* pool) {

if (pool->nextIndex < POOL\_SIZE) {

return pool->packets[pool->nextIndex++];

} else {

// Handle the case when the pool is exhausted (you may want to expand the pool or return NULL)

printf("Memory pool exhausted!\n");

return NULL;

}

}

// Release a packet back to the memory pool

void deallocatePacket(MemoryPool\* pool, Packet\* packet) {

// You may want to perform additional cleanup or re-initialization of the packet

pool->nextIndex--;

}

// Cleanup the memory pool

void cleanupMemoryPool(MemoryPool\* pool) {

for (int i = 0; i < POOL\_SIZE; ++i) {

free(pool->packets[i]);

}

pool->nextIndex = 0;

}

// Example of using the memory pool

int main() {

MemoryPool packetPool;

initializeMemoryPool(&packetPool);

// Allocate and use packets

Packet\* packet1 = allocatePacket(&packetPool);

// Process packet1...

Packet\* packet2 = allocatePacket(&packetPool);

// Process packet2...

// Release packets back to the pool when done

deallocatePacket(&packetPool, packet1);

deallocatePacket(&packetPool, packet2);

// Cleanup the memory pool when the application exits

cleanupMemoryPool(&packetPool);

return 0;

}

## DPDK

Run to Completion vs Pipeline

Dpdk setup -> compile drivers -> igb pmd - >bind driver to dpdk - >huge pages - >application run

Rte\_eal\_init (app to use dpdk framework)

Rte\_eth\_Dev\_configure – rx/tx queues , RSS

Rte\_Eth\_tx\_queue\_setup – ring descriptors

Rte\_eth\_rx\_queue\_setup

Rte\_eth\_dev\_Start – RX/TX packets

Rte\_eth\_rx\_burst/tx\_burst (port id.q no, array of structure to hold packets, number)

DPDK (Data Plane Development Kit) is a set of libraries and drivers for fast packet processing in user space. It is designed to optimize packet processing workloads and achieve high-performance data plane applications. DPDK allows developers to build efficient and scalable network applications by bypassing the traditional kernel networking stack and accessing network interfaces directly from user space.

Key features and components of DPDK include:

A close-up of a white background

Description automatically generated

**Poll Mode Drivers (PMDs):**

DPDK provides a set of Poll Mode Drivers (PMDs) for various network interface controllers (NICs).

PMDs allow applications to directly interact with network interfaces, eliminating the need for kernel involvement in the data plane.

**Memory Management:**

DPDK includes a memory management subsystem that provides a high-performance memory allocation and deallocation framework.

It uses a custom memory allocator called the DPDK Memory Manager, which helps reduce memory fragmentation and overhead.

**Poll Mode Operation:**

DPDK applications operate in a poll mode, continuously polling the NICs and processing packets without relying on interrupts.

Poll mode operation reduces the overhead of interrupt processing and context switching, resulting in lower latency.

**Packet Buffer Management:**

DPDK includes a memory pool manager to efficiently manage packet buffers.

The memory pool manager allocates fixed-sized memory blocks, or mbufs, to store packet data. This improves cache locality and reduces memory fragmentation.

**RTE (Run-Time Environment):**

DPDK provides a set of APIs and abstractions collectively known as the RTE (Run-Time Environment).

RTE includes functions for memory management, I/O operations, and other utilities that facilitate the development of high-performance applications.

Multi-Queue Support:

DPDK supports multi-queue operation, allowing applications to distribute packet processing across multiple CPU cores.

This helps achieve parallelism and scalability in packet processing workloads.

**Packet Classification and Filtering:**

DPDK provides mechanisms for packet classification, filtering, and flow steering.

Applications can define rules for classifying and directing packets based on criteria such as MAC addresses, IP addresses, and ports.

**Event Mode:**

DPDK includes an Event Mode framework that allows applications to handle events asynchronously.

The Event Mode enables event-driven architectures for handling various tasks, including packet processing.

Crypto and Compression Operations:

DPDK includes libraries for hardware-accelerated cryptographic and compression operations.

Applications can leverage these libraries for secure and efficient processing of encrypted and compressed data.

**Unified Packet Framework:**

DPDK provides a unified packet framework that abstracts packet data structures and operations, making it easier for developers to work with packets in a consistent manner.

Dynamic Device Configuration:

DPDK allows applications to dynamically configure and manage network devices, including adding or removing devices during runtime.

**DPDK EAL (Environment Abstraction Layer):**

The Environment Abstraction Layer provides an abstraction over system-specific details and allows DPDK applications to run on various operating systems.

DPDK is commonly used in the development of network functions, such as routers, firewalls, load balancers, and other data plane applications requiring low-latency, high-throughput packet processing. It is widely adopted in the telecommunications and networking industry for building efficient and scalable network infrastructure.

#include <stdio.h>

#include <stdint.h>

#include <inttypes.h>

#include <rte\_eal.h>

#include <rte\_ethdev.h>

#include <rte\_lcore.h>

#include <rte\_mbuf.h>

#define RX\_RING\_SIZE 1024

#define TX\_RING\_SIZE 1024

#define NUM\_MBUFS 8191

#define MBUF\_CACHE\_SIZE 250

#define BURST\_SIZE 32

static const struct rte\_eth\_conf port\_conf\_default = {

.rxmode = {

.max\_rx\_pkt\_len = RTE\_ETHER\_MAX\_LEN,

},

};

static struct rte\_eth\_dev\_tx\_buffer \*tx\_buffer[RTE\_MAX\_ETHPORTS];

static uint16\_t nb\_ports;

static int init\_port(uint16\_t port, struct rte\_mempool \*mbuf\_pool) {

struct rte\_eth\_conf port\_conf = port\_conf\_default;

const uint16\_t rx\_rings = 1, tx\_rings = 1;

int retval;

uint16\_t q;

if (port >= rte\_eth\_dev\_count\_avail()) {

printf("Error: Port %" PRIu16 " not available\n", port);

return -1;

}

retval = rte\_eth\_dev\_configure(port, rx\_rings, tx\_rings, &port\_conf);

if (retval != 0) {

printf("Error during port configuration: %s\n", rte\_strerror(retval));

return retval;

}

retval = rte\_eth\_dev\_adjust\_nb\_rx\_tx\_desc(port, &RX\_RING\_SIZE, &TX\_RING\_SIZE);

if (retval != 0) {

printf("Error during adjusting number of descriptors: %s\n", rte\_strerror(retval));

return retval;

}

for (q = 0; q < rx\_rings; q++) {

retval = rte\_eth\_rx\_queue\_setup(port, q, RX\_RING\_SIZE,

rte\_eth\_dev\_socket\_id(port), NULL, mbuf\_pool);

if (retval < 0) {

printf("Error during RX queue setup: %s\n", rte\_strerror(-retval));

return retval;

}

}

for (q = 0; q < tx\_rings; q++) {

retval = rte\_eth\_tx\_queue\_setup(port, q, TX\_RING\_SIZE,

rte\_eth\_dev\_socket\_id(port), NULL);

if (retval < 0) {

printf("Error during TX queue setup: %s\n", rte\_strerror(-retval));

return retval;

}

tx\_buffer[port] = rte\_zmalloc\_socket("tx\_buffer",

RTE\_ETH\_TX\_BUFFER\_SIZE(BURST\_SIZE),

0, rte\_eth\_dev\_socket\_id(port));

if (tx\_buffer[port] == NULL) {

printf("Error: cannot allocate buffer for tx on port %u\n", port);

return -1;

}

rte\_eth\_tx\_buffer\_init(tx\_buffer[port], BURST\_SIZE);

}

retval = rte\_eth\_dev\_start(port);

if (retval < 0) {

printf("Error during port start: %s\n", rte\_strerror(-retval));

return retval;

}

rte\_eth\_promiscuous\_enable(port);

return 0;

}

static void dpdk\_packet\_receive(uint16\_t port) {

struct rte\_mbuf \*bufs[BURST\_SIZE];

uint16\_t nb\_rx;

printf("Core %u receiving packets. [Ctrl+C to quit]\n", rte\_lcore\_id());

while (1) {

nb\_rx = rte\_eth\_rx\_burst(port, 0, bufs, BURST\_SIZE);

if (nb\_rx > 0) {

// Process received packets (can be done on a different core)

// For simplicity, just free the received packets

rte\_pktmbuf\_free\_bulk(bufs, nb\_rx);

}

}

}

static void dpdk\_packet\_process(uint16\_t port) {

struct rte\_mbuf \*bufs[BURST\_SIZE];

uint16\_t nb\_rx, nb\_tx;

uint16\_t dst\_port = (port + 1) % nb\_ports;

printf("Core %u processing packets. [Ctrl+C to quit]\n", rte\_lcore\_id());

while (1) {

nb\_rx = rte\_eth\_rx\_burst(port, 0, bufs, BURST\_SIZE);

if (nb\_rx > 0) {

// Process received packets

// For simplicity, just transmit the received packets to another port

nb\_tx = rte\_eth\_tx\_buffer(dst\_port, 0, tx\_buffer[dst\_port], bufs, nb\_rx);

if (nb\_tx < nb\_rx) {

rte\_pktmbuf\_free\_bulk(bufs + nb\_tx, nb\_rx - nb\_tx);

}

}

}

}

int main(int argc, char \*argv[]) {

struct rte\_mempool \*mbuf\_pool;

uint16\_t portid;

unsigned lcore\_id;

if (rte\_eal\_init(argc, argv) < 0) {

rte\_panic("Cannot init EAL\n");

return -1;

}

argc -= rte\_eal\_init(argc, argv);

argv += rte\_eal\_init(argc, argv);

nb\_ports = rte\_eth\_dev\_count\_avail();

if (nb\_ports == 0) {

rte\_panic("No Ethernet ports found\n");

return -1;

}

mbuf\_pool = rte\_pktmbuf\_pool\_create("MBUF\_POOL", NUM\_MBUFS \* nb\_ports,

MBUF\_CACHE\_SIZE, 0, RTE\_MBUF\_DEFAULT\_BUF\_SIZE,

rte\_socket\_id());

if (mbuf\_pool == NULL) {

rte\_panic("Cannot create mbuf pool\n");

return -1;

}

RTE\_ETH\_FOREACH\_DEV(portid) {

if (init\_port(portid, mbuf\_pool) != 0)

return -1;

}

RTE\_LCORE\_FOREACH\_SLAVE(lcore\_id) {

rte\_eal\_remote\_launch((lcore\_function\_t \*)dpdk\_packet\_process, NULL, lcore\_id);

}

// Run packet receiving on the master core

dpdk\_packet\_receive(0);

rte\_eal\_mp\_wait\_lcore();

return 0;

}

## Attack Packet Detection With PCAP

#include <stdio.h>

#include <pcap.h>

#include <string.h>

#define MAX\_PACKET\_SIZE 1500

void packet\_handler(unsigned char\* user, const struct pcap\_pkthdr\* pkthdr, const unsigned char\* packet) {

char\* attack\_string = "attack";

const int attack\_string\_length = strlen(attack\_string);

// Assuming Ethernet frames

int ethernet\_header\_length = 14;

// Assuming IPv4 packets

int ip\_header\_length = (packet[ethernet\_header\_length] & 0x0F) \* 4;

// Assuming TCP packets

int tcp\_header\_length = ((packet[ethernet\_header\_length + ip\_header\_length + 12] & 0xF0) >> 4) \* 4;

// Payload offset

int payload\_offset = ethernet\_header\_length + ip\_header\_length + tcp\_header\_length;

// Extract payload

const unsigned char\* payload = packet + payload\_offset;

// Check if the payload contains the attack string

if (strstr((char\*)payload, attack\_string) != NULL) {

printf("Attack detected!\n");

}

}

int main() {

char errbuf[PCAP\_ERRBUF\_SIZE];

pcap\_t\* handle;

// Open the network interface for packet capture

handle = pcap\_open\_live("your\_network\_interface", MAX\_PACKET\_SIZE, 1, 1000, errbuf);

if (handle == NULL) {

fprintf(stderr, "Couldn't open device: %s\n", errbuf);

return 2;

}

// Set a packet filter to capture only TCP traffic

struct bpf\_program fp;

char filter\_exp[] = "tcp";

if (pcap\_compile(handle, &fp, filter\_exp, 0, PCAP\_NETMASK\_UNKNOWN) == -1) {

fprintf(stderr, "Couldn't parse filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

return 2;

}

if (pcap\_setfilter(handle, &fp) == -1) {

fprintf(stderr, "Couldn't install filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

return 2;

}

// Start capturing packets

pcap\_loop(handle, 0, packet\_handler, NULL);

// Close the handle

pcap\_close(handle);

return 0;

}

## Attack Packet Context Generation - HTTP

#include <stdio.h>

#include <pcap.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_PACKET\_SIZE 1500

void process\_http\_packet(const unsigned char\* payload, int payload\_size) {

// Assuming HTTP method is present at the beginning of the payload

// Detecting the end of the HTTP method by finding the first space

int i = 0;

while (i < payload\_size && payload[i] != ' ') {

i++;

}

if (i < payload\_size) {

// Extracting the HTTP method

char http\_method[i + 1];

strncpy(http\_method, (char\*)payload, i);

http\_method[i] = '\0';

printf("HTTP Method: %s\n", http\_method);

}

}

void packet\_handler(unsigned char\* user, const struct pcap\_pkthdr\* pkthdr, const unsigned char\* packet) {

// Assuming Ethernet frames

int ethernet\_header\_length = 14;

// Assuming IPv4 packets

int ip\_header\_length = (packet[ethernet\_header\_length] & 0x0F) \* 4;

// Assuming TCP packets

int tcp\_header\_length = ((packet[ethernet\_header\_length + ip\_header\_length + 12] & 0xF0) >> 4) \* 4;

// Payload offset

int payload\_offset = ethernet\_header\_length + ip\_header\_length + tcp\_header\_length;

// Extract payload

const unsigned char\* payload = packet + payload\_offset;

// Assuming HTTP traffic starts with "HTTP/"

const char http\_identifier[] = "HTTP/";

const int http\_identifier\_length = strlen(http\_identifier);

// Check if the payload contains "HTTP/"

if (strncmp((char\*)payload, http\_identifier, http\_identifier\_length) == 0) {

// Process the HTTP packet

process\_http\_packet(payload, pkthdr->caplen - payload\_offset);

}

}

int main() {

char errbuf[PCAP\_ERRBUF\_SIZE];

pcap\_t\* handle;

// Open the network interface for packet capture

handle = pcap\_open\_live("your\_network\_interface", MAX\_PACKET\_SIZE, 1, 1000, errbuf);

if (handle == NULL) {

fprintf(stderr, "Couldn't open device: %s\n", errbuf);

return 2;

}

// Set a packet filter to capture only TCP traffic on port 80 (HTTP)

struct bpf\_program fp;

char filter\_exp[] = "tcp port 80";

if (pcap\_compile(handle, &fp, filter\_exp, 0, PCAP\_NETMASK\_UNKNOWN) == -1) {

fprintf(stderr, "Couldn't parse filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

return 2;

}

if (pcap\_setfilter(handle, &fp) == -1) {

fprintf(stderr, "Couldn't install filter %s: %s\n", filter\_exp, pcap\_geterr(handle));

return 2;

}

// Start capturing packets

pcap\_loop(handle, 0, packet\_handler, NULL);

// Close the handle

pcap\_close(handle);

return 0;

}

## Breadth First Search(BFS) Of Binary Tree

#include <stdio.h>

#include <stdlib.h>

// Structure for a tree node

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

// Function to create a new tree node

struct TreeNode\* createNode(int value) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to perform Breadth-First Search (BFS) on a binary tree

void breadthFirstSearch(struct TreeNode\* root) {

if (root == NULL) {

return;

}

// Use a queue for BFS

struct TreeNode\* queue[100];

int front = 0, rear = 0;

// Enqueue the root

queue[rear++] = root;

while (front < rear) {

// Dequeue a node and print its data

struct TreeNode\* current = queue[front++];

printf("%d ", current->data);

// Enqueue left child

if (current->left != NULL) {

queue[rear++] = current->left;

}

// Enqueue right child

if (current->right != NULL) {

queue[rear++] = current->right;

}

}

}

int main() {

// Constructing a sample binary tree

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->left = createNode(6);

root->right->right = createNode(7);

printf("Breadth-First Search (BFS) of the tree: ");

breadthFirstSearch(root);

return 0;

}

## Depth First Search(DFS) Of Binary Tree

#include <stdio.h>

#include <stdlib.h>

// Structure for a tree node

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

// Function to create a new tree node

struct TreeNode\* createNode(int value) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Pre-order DFS traversal

void preOrderDFS(struct TreeNode\* root) {

if (root == NULL) {

return;

}

// Print the data of the current node

printf("%d ", root->data);

// Recur on the left subtree

preOrderDFS(root->left);

// Recur on the right subtree

preOrderDFS(root->right);

}

// In-order DFS traversal

void inOrderDFS(struct TreeNode\* root) {

if (root == NULL) {

return;

}

// Recur on the left subtree

inOrderDFS(root->left);

// Print the data of the current node

printf("%d ", root->data);

// Recur on the right subtree

inOrderDFS(root->right);

}

// Post-order DFS traversal

void postOrderDFS(struct TreeNode\* root) {

if (root == NULL) {

return;

}

// Recur on the left subtree

postOrderDFS(root->left);

// Recur on the right subtree

postOrderDFS(root->right);

// Print the data of the current node

printf("%d ", root->data);

}

int main() {

// Constructing a sample binary tree

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->left = createNode(6);

root->right->right = createNode(7);

printf("Pre-order DFS of the tree: ");

preOrderDFS(root);

printf("\n");

printf("In-order DFS of the tree: ");

inOrderDFS(root);

printf("\n");

printf("Post-order DFS of the tree: ");

postOrderDFS(root);

printf("\n");

return 0;

}

## Breadth First Search(BFS) Of Graph

#include <stdio.h>

#include <stdlib.h>

// Structure for a node in the adjacency list

struct Node {

int vertex;

struct Node\* next;

};

// Structure for the adjacency list

struct AdjList {

struct Node\* head;

};

// Structure for the graph

struct Graph {

int numVertices;

struct AdjList\* array;

};

// Function to create a new node

struct Node\* createNode(int vertex) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

// Function to create a graph with a given number of vertices

struct Graph\* createGraph(int numVertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = numVertices;

// Create an array of adjacency lists

graph->array = (struct AdjList\*)malloc(numVertices \* sizeof(struct AdjList));

// Initialize each adjacency list as empty

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

graph->array[i].head = NULL;

}

return graph;

}

// Function to add an edge to an undirected graph

void addEdge(struct Graph\* graph, int src, int dest) {

// Add an edge from src to dest

struct Node\* newNode = createNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

// Since the graph is undirected, add an edge from dest to src as well

newNode = createNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// Function to perform Breadth-First Search (BFS) on the graph

void BFS(struct Graph\* graph, int startVertex) {

// Create a queue for BFS

int queue[100];

int front = 0, rear = 0;

int visited[100] = {0}; // Initialize visited array

// Enqueue the start vertex and mark it as visited

queue[rear++] = startVertex;

visited[startVertex] = 1;

while (front < rear) {

// Dequeue a vertex and print it

int currentVertex = queue[front++];

printf("%d ", currentVertex);

// Traverse the adjacency list of the dequeued vertex

struct Node\* temp = graph->array[currentVertex].head;

while (temp != NULL) {

int adjacentVertex = temp->vertex;

// If the adjacent vertex is not visited, enqueue it and mark it as visited

if (!visited[adjacentVertex]) {

queue[rear++] = adjacentVertex;

visited[adjacentVertex] = 1;

}

temp = temp->next;

}

}

}

int main() {

// Create a sample graph

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 4);

addEdge(graph, 3, 5);

addEdge(graph, 4, 5);

printf("Breadth-First Search (BFS) starting from vertex 0: ");

BFS(graph, 0);

return 0;

}

## Depth First Search(DFS) Of Graph

#include <stdio.h>

#include <stdlib.h>

// Structure for a node in the adjacency list

struct Node {

int vertex;

struct Node\* next;

};

// Structure for the adjacency list

struct AdjList {

struct Node\* head;

};

// Structure for the graph

struct Graph {

int numVertices;

struct AdjList\* array;

};

// Function to create a new node

struct Node\* createNode(int vertex) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

// Function to create a graph with a given number of vertices

struct Graph\* createGraph(int numVertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = numVertices;

// Create an array of adjacency lists

graph->array = (struct AdjList\*)malloc(numVertices \* sizeof(struct AdjList));

// Initialize each adjacency list as empty

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

graph->array[i].head = NULL;

}

return graph;

}

// Function to add an edge to an undirected graph

void addEdge(struct Graph\* graph, int src, int dest) {

// Add an edge from src to dest

struct Node\* newNode = createNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

// Since the graph is undirected, add an edge from dest to src as well

newNode = createNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// Recursive function for DFS traversal

void DFSUtil(struct Graph\* graph, int vertex, int visited[]) {

// Mark the current vertex as visited and print it

visited[vertex] = 1;

printf("%d ", vertex);

// Recur for all the vertices adjacent to this vertex

struct Node\* temp = graph->array[vertex].head;

while (temp != NULL) {

int adjacentVertex = temp->vertex;

if (!visited[adjacentVertex]) {

DFSUtil(graph, adjacentVertex, visited);

}

temp = temp->next;

}

}

// Function to perform Depth-First Search (DFS) on the graph

void DFS(struct Graph\* graph, int startVertex) {

// Create an array to track visited vertices

int visited[100] = {0}; // Assuming a maximum of 100 vertices

// Call the recursive utility function starting from the specified vertex

DFSUtil(graph, startVertex, visited);

}

int main() {

// Create a sample graph

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 4);

addEdge(graph, 3, 5);

addEdge(graph, 4, 5);

printf("Depth-First Search (DFS) starting from vertex 0: ");

DFS(graph, 0);

return 0;

}

## Shortest Path In Graph

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

// Structure for a node in the adjacency list

struct Node {

int vertex;

struct Node\* next;

};

// Structure for the adjacency list

struct AdjList {

struct Node\* head;

};

// Structure for the graph

struct Graph {

int numVertices;

struct AdjList\* array;

};

// Function to create a new node

struct Node\* createNode(int vertex) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

// Function to create a graph with a given number of vertices

struct Graph\* createGraph(int numVertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = numVertices;

// Create an array of adjacency lists

graph->array = (struct AdjList\*)malloc(numVertices \* sizeof(struct AdjList));

// Initialize each adjacency list as empty

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

graph->array[i].head = NULL;

}

return graph;

}

// Function to add an edge to an undirected graph

void addEdge(struct Graph\* graph, int src, int dest) {

// Add an edge from src to dest

struct Node\* newNode = createNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

// Since the graph is undirected, add an edge from dest to src as well

newNode = createNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// Function to find the shortest path using BFS

void shortestPath(struct Graph\* graph, int startVertex, int endVertex) {

int queue[MAX\_VERTICES];

int front = 0, rear = 0;

int visited[MAX\_VERTICES] = {0};

int parent[MAX\_VERTICES];

// Initialize parent array to -1

for (int i = 0; i < MAX\_VERTICES; ++i) {

parent[i] = -1;

}

// Enqueue the start vertex and mark it as visited

queue[rear++] = startVertex;

visited[startVertex] = 1;

while (front < rear) {

int currentVertex = queue[front++];

struct Node\* temp = graph->array[currentVertex].head;

while (temp != NULL) {

int adjacentVertex = temp->vertex;

if (!visited[adjacentVertex]) {

// Enqueue adjacent vertex, mark it as visited, and update parent

queue[rear++] = adjacentVertex;

visited[adjacentVertex] = 1;

parent[adjacentVertex] = currentVertex;

}

temp = temp->next;

}

}

// Print the shortest path from startVertex to endVertex

printf("Shortest path from %d to %d: ", startVertex, endVertex);

int currentVertex = endVertex;

while (currentVertex != -1) {

printf("%d ", currentVertex);

currentVertex = parent[currentVertex];

}

printf("\n");

}

int main() {

// Create a sample graph

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 4);

addEdge(graph, 3, 5);

addEdge(graph, 4, 5);

// Find the shortest path from vertex 0 to vertex 5

shortestPath(graph, 0, 5);

return 0;

}

## Sliding Window Maximum(Maximum element in a window of size k in linear time)

#include <stdio.h>

#include <stdlib.h>

int max(int a, int b) {

    return (a > b) ? a : b;

}

int\* maxSlidingWindow(int\* nums, int numsSize, int k, int\* returnSize) {

    if (nums == NULL || k <= 0) {

        \*returnSize = 0;

        return NULL;

    }

    int n = numsSize;

    int\* leftMax = (int\*)malloc(sizeof(int) \* n);

    int\* rightMax = (int\*)malloc(sizeof(int) \* n);

    leftMax[0] = nums[0];

    rightMax[n - 1] = nums[n - 1];

    for (int i = 1; i < n; i++) {

        // Start over every window and update it with maximum elements

        if ((i + 1) % k == 0) {

            leftMax[i] = nums[i];

            printf("\nReset leftmax[%d] = %d",i,leftMax[i]);

        } else {

            leftMax[i] = max(nums[i], leftMax[i - 1]);

            printf("\nleftmax[%d] = %d",i,leftMax[i]);

        }

        int j = n - i - 1;

        printf("\nj = %d",j);

        if ((j + 1) % k == 0) {

            rightMax[j] = nums[j];

            printf("\nReset righttmax[%d] = %d",j,rightMax[j]);

        } else {

            rightMax[j] = max(nums[j], rightMax[j + 1]);

            printf("\nrighttmax[%d] = %d",j,rightMax[j]);

        }

    }

    int\* ans = (int\*)malloc(sizeof(int) \* (n - k + 1));

    \*returnSize = n - k + 1;

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

        // Since leftMax has the maximum to the rightmost of the window,

        // and rightMax has the maximum element of the window to the leftMax

        // as per their traversals above.

        printf("\nInitial leftmax = %d rightmax = %d",leftMax[i],rightMax[i]);

        ans[i] = max(leftMax[i + k - 1], rightMax[i]);

        printf("\nleftmax = %d rightmax = %d",leftMax[i + k - 1],rightMax[i]);

    }

    free(leftMax);

    free(rightMax);

    return ans;

}

int main() {

    int nums[] = {1, 3, -1, -3, 5, 3, 6, 7}; // Sample array

    int k = 3; // Window size

    int numsSize = sizeof(nums) / sizeof(nums[0]); // Size of the array

    int returnSize; // Size of the returned array

    int\* result = maxSlidingWindow(nums, numsSize, k, &returnSize);

    // Print the maximum sliding window

    printf("Maximum sliding window: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    // Free dynamically allocated memory

    free(result);

    return 0;

}

## Throughput Load

#define NUM\_THREADS 10

// Define the workload function

void\* processRequest(void\* threadId) {

long tid = (long)threadId;

// Simulate some processing time

usleep(100000); // 100ms

printf("Thread %ld: Request processed\n", tid);

pthread\_exit(NULL);

}

int main() {

pthread\_t threads[NUM\_THREADS];

int rc;

long t;

printf("Simulation of Throughput Load\n");

// Create threads to simulate concurrent connections

for (t = 0; t < NUM\_THREADS; t++) {

printf("Creating thread %ld\n", t);

rc = pthread\_create(&threads[t], NULL, processRequest, (void\*)t);

if (rc) {

printf("Error creating thread %ld; return code from pthread\_create() is %d\n", t, rc);

exit(-1);

}

}

// Wait for all threads to finish

for (t = 0; t < NUM\_THREADS; t++) {

pthread\_join(threads[t], NULL);

}

printf("All threads completed.\n");

return 0;

}

## Middle Element Of Linked List

#include <stdio.h>

#include <stdlib.h>

// Definition of a simple singly linked list node

struct Node {

int data;

struct Node\* next;

};

// Function to insert a new node at the end of the linked list

void insertNode(struct Node\*\* head, int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

\*head = newNode;

return;

}

struct Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

// Function to find the middle element of the linked list

struct Node\* findMiddle(struct Node\* head) {

if (head == NULL || head->next == NULL) {

return head; // Empty list or single node

}

struct Node\* slow = head;

struct Node\* fast = head;

while (fast != NULL && fast->next != NULL) {

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

// Function to print the elements of the linked list

void printList(struct Node\* head) {

while (head != NULL) {

printf("%d -> ", head->data);

head = head->next;

}

printf("NULL\n");

}

int main() {

// Creating a sample linked list

struct Node\* head = NULL;

insertNode(&head, 1);

insertNode(&head, 2);

insertNode(&head, 3);

insertNode(&head, 4);

insertNode(&head, 5);

// Print the original linked list

printf("Original Linked List: ");

printList(head);

// Find the middle element

struct Node\* middle = findMiddle(head);

// Print the middle element

printf("Middle Element: %d\n", middle->data);

return 0;

}

## Build Process Tree

#include <stdio.h>

#include <stdlib.h>

struct ProcessNode {

char process\_name[20];

int pid;

int parent\_pid;

struct ProcessNode\* children[100];

int num\_children;

};

struct ProcessNode\* createProcessNode(const char\* process\_name, int pid, int parent\_pid) {

struct ProcessNode\* newNode = (struct ProcessNode\*)malloc(sizeof(struct ProcessNode));

snprintf(newNode->process\_name, sizeof(newNode->process\_name), "%s", process\_name);

newNode->pid = pid;

newNode->parent\_pid = parent\_pid;

newNode->num\_children = 0;

return newNode;

}

int findIndexByPid(struct ProcessNode\* process\_dict[], int num\_processes, int pid) {

for (int i = 0; i < num\_processes; i++) {

if (process\_dict[i]->pid == pid) {

return i;

}

}

return -1; // Process not found

}

void buildProcessTree(struct ProcessNode\* process\_dict[], int num\_processes, struct ProcessNode\*\* root) {

\*root = NULL;

int pidToIndex[num\_processes];

for (int i = 0; i < num\_processes; i++) {

pidToIndex[process\_dict[i]->pid] = i;

}

for (int i = 0; i < num\_processes; i++) {

if (process\_dict[i]->parent\_pid == -1) {

\*root = process\_dict[i];

} else {

int parentIndex = findIndexByPid(process\_dict, num\_processes, process\_dict[i]->parent\_pid);

if (parentIndex != -1) {

process\_dict[parentIndex]->children[process\_dict[parentIndex]->num\_children++] = process\_dict[i];

}

}

}

}

void printProcessTree(struct ProcessNode\* root, int depth) {

if (root == NULL) {

return;

}

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

printf("----");

}

printf("%s (PID: %d)\n", root->process\_name, root->pid);

for (int i = 0; i < root->num\_children; i++) {

printProcessTree(root->children[i], depth + 1);

}

}

int main() {

struct ProcessNode\* processes[9];

processes[0] = createProcessNode("a.exe", 420, 428);

processes[1] = createProcessNode("c.exe", 428, -1);

processes[2] = createProcessNode("d.exe", 551, 420);

processes[3] = createProcessNode("e.exe", 552, 428);

processes[4] = createProcessNode("f.exe", 553, -1);

processes[5] = createProcessNode("g.exe", 4, 553);

processes[6] = createProcessNode("b.exe", 7, 4);

processes[7] = createProcessNode("h.exe", 11, 7);

processes[8] = createProcessNode("j.exe", 666, 428);

struct ProcessNode\* root;

buildProcessTree(processes, 9, &root);

printProcessTree(root, 0);

return 0;

}Given the array, build a tree as follows:

[

{"process\_name":"a.exe", "pid":420, "parent\_pid":428},

{"process\_name":"c.exe", "pid":428, "parent\_pid":None},

{"process\_name":"d.exe", "pid":551, "parent\_pid":420},

{"process\_name":"e.exe", "pid":552, "parent\_pid":428},

{"process\_name":"f.exe", "pid":553, "parent\_pid":None},

{"process\_name":"g.exe", "pid": 4, "parent\_pid":553},

{"process\_name":"b.exe", "pid": 7, "parent\_pid": 4},

{"process\_name":"h.exe", "pid": 11, "parent\_pid": 7},

{"process\_name":"j.exe", "pid":666, "parent\_pid":428}

]

The program should output the following:

c.exe

---- a.exe

-------- d.exe

---- e.exe

---- j.exe

f.exe

---- g.exe

-------- b.exe

------------ h.exe

## Construct Tree From Data

#include <stdio.h>

#include <stdlib.h>

// Structure for a tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new tree node

struct TreeNode\* createNode(int value) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to construct a tree from an array of values

struct TreeNode\* constructTree(int values[], int index, int n) {

if (index < n) {

struct TreeNode\* root = createNode(values[index]);

root->left = constructTree(values, 2 \* index + 1, n); // Left child

root->right = constructTree(values, 2 \* index + 2, n); // Right child

return root;

} else {

return NULL;

}

}

// Function to print the tree using inorder traversal

void printTree(struct TreeNode\* root) {

if (root != NULL) {

printTree(root->left);

printf("%d ", root->data);

printTree(root->right);

}

}

int main() {

// Example array of values to construct a binary tree

int values[] = {1, 2, 3, 4, 5, 6, 7};

// Calculate the number of elements in the array

int n = sizeof(values) / sizeof(values[0]);

// Construct the binary tree

struct TreeNode\* root = constructTree(values, 0, n);

// Print the tree using inorder traversal

printf("Inorder Traversal of the Tree: ");

printTree(root);

return 0;

}

## Peak Element

#include <stdio.h>

// Function to find a peak element using binary search

int findPeakElement(int arr[], int low, int high, int n) {

while (low <= high) {

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

// Check if the mid element is a peak

if ((mid == 0 || arr[mid] >= arr[mid - 1]) && (mid == n - 1 || arr[mid] >= arr[mid + 1])) {

return mid;

}

// If the element to the right of mid is greater, then there is a peak element on the right

if (mid < n - 1 && arr[mid] < arr[mid + 1]) {

low = mid + 1;

} else {

// If the element to the left of mid is greater, then there is a peak element on the left

high = mid - 1;

}

}

return -1; // No peak element found

}

int main() {

int arr[] = {1, 3, 20, 4, 1, 0};

int n = sizeof(arr) / sizeof(arr[0]);

int peakIndex = findPeakElement(arr, 0, n - 1, n);

if (peakIndex != -1) {

printf("Peak element is %d at index %d\n", arr[peakIndex], peakIndex);

} else {

printf("No peak element found\n");

}

return 0;

}

## T2 subtree of T1

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

// Definition for a binary tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to check if two trees are identical

bool areIdentical(struct TreeNode\* root1, struct TreeNode\* root2) {

if (root1 == NULL && root2 == NULL) {

return true;

}

if (root1 == NULL || root2 == NULL) {

return false;

}

return (root1->data == root2->data &&

areIdentical(root1->left, root2->left) &&

areIdentical(root1->right, root2->right));

}

// Function to check if Tree T2 is a subtree of Tree T1

bool isSubtree(struct TreeNode\* T1, struct TreeNode\* T2) {

if (T2 == NULL) {

return true; // An empty tree is always a subtree

}

if (T1 == NULL) {

return false; // T2 cannot be a subtree if T1 is empty

}

// Check if the current subtree of T1 is identical to T2

if (areIdentical(T1, T2)) {

return true;

}

// Recursively check in the left and right subtrees of T1

return isSubtree(T1->left, T2) || isSubtree(T1->right, T2);

}

// Function to create a new tree node

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

int main() {

// Construct Tree T1

struct TreeNode\* T1 = createNode(3);

T1->left = createNode(4);

T1->right = createNode(5);

T1->left->left = createNode(1);

T1->left->right = createNode(2);

// Construct Tree T2

struct TreeNode\* T2 = createNode(4);

T2->left = createNode(1);

T2->right = createNode(2);

// Check if T2 is a subtree of T1

if (isSubtree(T1, T2)) {

printf("Tree T2 is a subtree of Tree T1.\n");

} else {

printf("Tree T2 is not a subtree of Tree T1.\n");

}

// Free the allocated memory

free(T1);

free(T2);

return 0;

}

## Minimum Swaps Required To Convert Min Heap to BST

#include <stdio.h>

#include <stdlib.h>

// Structure for a tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to count the number of swaps required to convert a min-heap to a BST

int countSwaps(int arr[], int n) {

// Create an array to store the positions of elements in the sorted order

int positions[n];

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

positions[i] = i;

}

// Sort the positions array based on the values in the arr array

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

for (int j = i + 1; j < n; j++) {

if (arr[i] > arr[j]) {

// Swap the values in the arr array

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

// Swap the positions in the positions array

temp = positions[i];

positions[i] = positions[j];

positions[j] = temp;

}

}

}

// Count the number of swaps needed to convert positions array to a sorted array

int swaps = 0;

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

while (positions[i] != i) {

// Swap the positions until the element reaches its correct position

int temp = positions[i];

positions[i] = positions[positions[i]];

positions[temp] = temp;

swaps++;

}

}

return swaps;

}

int main() {

int arr[] = {3, 2, 1, 5, 4};

int n = sizeof(arr) / sizeof(arr[0]);

int swaps = countSwaps(arr, n);

printf("Minimum number of swaps required: %d\n", swaps);

return 0;

}

## Check If Two Files Are Identical

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <openssl/sha.h>

#define MAX\_FILE\_SIZE 1024

void calculate\_sha256(const char \*filename, unsigned char hash[SHA256\_DIGEST\_LENGTH]) {

FILE \*file = fopen(filename, "rb");

if (file == NULL) {

perror("Error opening file");

exit(EXIT\_FAILURE);

}

SHA256\_CTX sha256;

SHA256\_Init(&sha256);

size\_t bytes;

unsigned char buffer[MAX\_FILE\_SIZE];

while ((bytes = fread(buffer, 1, sizeof(buffer), file)) != 0) {

SHA256\_Update(&sha256, buffer, bytes);

}

SHA256\_Final(hash, &sha256);

fclose(file);

}

int compare\_files(const char \*file1, const char \*file2) {

unsigned char hash1[SHA256\_DIGEST\_LENGTH];

unsigned char hash2[SHA256\_DIGEST\_LENGTH];

calculate\_sha256(file1, hash1);

calculate\_sha256(file2, hash2);

return memcmp(hash1, hash2, SHA256\_DIGEST\_LENGTH) == 0;

}

int main(int argc, char \*argv[]) {

if (argc < 3) {

fprintf(stderr, "Usage: %s file1 file2\n", argv[0]);

return EXIT\_FAILURE;

}

const char \*file1 = argv[1];

const char \*file2 = argv[2];

if (compare\_files(file1, file2)) {

printf("The content of %s and %s is the same.\n", file1, file2);

} else {

printf("The content of %s and %s is different.\n", file1, file2);

}

return EXIT\_SUCCESS;

}

## Git Commit History

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_COMMIT\_ID\_LEN 41 // 40 characters for the SHA-1 hash + 1 for the null terminator

typedef struct {

char commit\_id[MAX\_COMMIT\_ID\_LEN];

unsigned long review\_timestamp;

unsigned long merge\_timestamp;

} GitCommit;

// Function to parse commit information from a Git log

void parse\_git\_log(const char \*log\_content, GitCommit \*commits, int \*num\_commits) {

char \*token = strtok((char \*)log\_content, "\n");

int index = 0;

while (token != NULL) {

// Assuming the log format is "commit\_id review\_timestamp merge\_timestamp"

sscanf(token, "%s %lu %lu", commits[index].commit\_id,

&commits[index].review\_timestamp, &commits[index].merge\_timestamp);

token = strtok(NULL, "\n");

index++;

}

\*num\_commits = index;

}

int main() {

// Example Git log content

const char \*git\_log\_content = "commit\_id\_1 1633336800 1633336900\n"

"commit\_id\_2 1633337000 1633337100\n"

"commit\_id\_3 1633337200 1633337300\n";

// Parse Git log and store commit information

GitCommit commits[100]; // Adjust the array size based on your needs

int num\_commits = 0;

parse\_git\_log(git\_log\_content, commits, &num\_commits);

// Print commit information

for (int i = 0; i < num\_commits; i++) {

printf("Commit ID: %s\n", commits[i].commit\_id);

printf("Review Timestamp: %lu\n", commits[i].review\_timestamp);

printf("Merge Timestamp: %lu\n", commits[i].merge\_timestamp);

printf("\n");

}

return 0;

}

## Recursive Links In Web page

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <libxml/HTMLparser.h>

#define MAX\_URL\_LENGTH 2048

void parseLinks(const char \*url);

void parseChildLinks(xmlNode \*node) {

    xmlNode \*cur\_node = NULL;

    for (cur\_node = node; cur\_node; cur\_node = cur\_node->next) {

        if (cur\_node->type == XML\_ELEMENT\_NODE && strcmp((char \*)cur\_node->name, "a") == 0) {

            xmlChar \*link = xmlGetProp(cur\_node, (const xmlChar \*)"href");

            if (link != NULL) {

                printf("Link: %s\n", link);

                xmlFree(link);

            }

        }

        if (cur\_node->type == XML\_ELEMENT\_NODE && strcmp((char \*)cur\_node->name, "a") != 0) {

            parseChildLinks(cur\_node->children);

        }

    }

}

void parseLinks(const char \*url) {

    xmlDoc \*doc = NULL;

    htmlDocPtr html\_doc = NULL;

    htmlNode \*root\_element = NULL;

    html\_doc = htmlReadFile(url, NULL, HTML\_PARSE\_NOBLANKS | HTML\_PARSE\_NOERROR | HTML\_PARSE\_NOWARNING);

    if (html\_doc == NULL) {

        fprintf(stderr, "Failed to parse document: %s\n", url);

        return;

    }

    root\_element = xmlDocGetRootElement((xmlDoc \*)html\_doc);

    parseChildLinks(root\_element->children);

    xmlFreeDoc(html\_doc);

}

int main(int argc, char \*argv[]) {

    if (argc != 2) {

        fprintf(stderr, "Usage: %s <URL>\n", argv[0]);

        return 1;

    }

    parseLinks(argv[1]);

    return 0;

}

-----

#include <stdio.h>

#include <stdlib.h>

#include <curl/curl.h>

#include <string.h>

#include <json-c/json.h>

// Buffer to store the response from the Wikipedia API

struct MemoryStruct {

char \*memory;

size\_t size;

};

// Callback function to write received data to the buffer

static size\_t WriteMemoryCallback(void \*contents, size\_t size, size\_t nmemb, void \*userp) {

size\_t realsize = size \* nmemb;

struct MemoryStruct \*mem = (struct MemoryStruct \*)userp;

mem->memory = realloc(mem->memory, mem->size + realsize + 1);

if (mem->memory == NULL) {

// Out of memory

fprintf(stderr, "Not enough memory (realloc returned NULL)\n");

return 0;

}

memcpy(&(mem->memory[mem->size]), contents, realsize);

mem->size += realsize;

mem->memory[mem->size] = 0;

return realsize;

}

// Function to fetch links from a Wikipedia page using Wikipedia API

void fetchWikipediaLinks(const char \*page) {

CURL \*curl;

CURLcode res;

struct MemoryStruct chunk;

chunk.memory = malloc(1);

chunk.size = 0;

// Initialize libcurl

curl\_global\_init(CURL\_GLOBAL\_DEFAULT);

curl = curl\_easy\_init();

if (curl) {

char url[256];

snprintf(url, sizeof(url), "https://en.wikipedia.org/w/api.php?action=parse&page=%s&prop=links&format=json", page);

// Set the URL for the request

curl\_easy\_setopt(curl, CURLOPT\_URL, url);

// Set the callback function to write response data to the buffer

curl\_easy\_setopt(curl, CURLOPT\_WRITEFUNCTION, WriteMemoryCallback);

curl\_easy\_setopt(curl, CURLOPT\_WRITEDATA, (void \*)&chunk);

// Perform the request

res = curl\_easy\_perform(curl);

// Check for errors

if (res != CURLE\_OK) {

fprintf(stderr, "curl\_easy\_perform() failed: %s\n", curl\_easy\_strerror(res));

} else {

// Parse JSON response

struct json\_object \*root, \*pages, \*links;

root = json\_tokener\_parse(chunk.memory);

if (!json\_object\_object\_get\_ex(root, "parse", &pages)) {

fprintf(stderr, "Error parsing JSON response\n");

} else if (!json\_object\_object\_get\_ex(pages, "links", &links)) {

fprintf(stderr, "Error parsing links from JSON response\n");

} else {

// Iterate through links and print them

printf("Links on %s:\n", page);

for (int i = 0; i < json\_object\_array\_length(links); i++) {

struct json\_object \*linkObj = json\_object\_array\_get\_idx(links, i);

const char \*link;

json\_object\_object\_get\_ex(linkObj, "\*",

&link); // The actual link is stored under the "\*" key

printf("%d. %s\n", i + 1, link);

}

}

}

// Cleanup

curl\_easy\_cleanup(curl);

curl\_global\_cleanup();

}

free(chunk.memory);

}

int main() {

const char \*startPage = "Computer\_Science"; // Replace with your desired starting Wikipedia page

// Fetch links recursively

fetchWikipediaLinks(startPage);

return 0;

}

## Left Most Nodes

#include <stdio.h>

#include <stdlib.h>

// Structure for a tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new tree node

struct TreeNode\* createNode(int value) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to find the leftmost nodes in a binary tree

void leftmostNodes(struct TreeNode\* root) {

if (root == NULL) {

printf("Tree is empty\n");

return;

}

struct TreeNode\* current = root;

while (current != NULL) {

printf("%d ", current->data);

current = current->left;

}

printf("\n");

}

int main() {

// Constructing a sample binary tree

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

root->left->left = createNode(4);

root->left->right = createNode(5);

root->right->left = createNode(6);

root->right->right = createNode(7);

printf("Leftmost nodes in the binary tree: ");

leftmostNodes(root);

return 0;

}

## Recursive Print Linked List

#include <stdio.h>

#include <stdlib.h>

// Structure for a linked list node

struct Node {

int data;

struct Node\* next;

};

// Function to create a new linked list node

struct Node\* createNode(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = NULL;

return newNode;

}

// Function to recursively parse and print the linked list

void recursiveParse(struct Node\* current) {

if (current == NULL) {

return;

}

// Print the current node's data

printf("%d ", current->data);

// Move to the next node recursively

recursiveParse(current->next);

}

int main() {

// Constructing a sample linked list

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = createNode(4);

printf("Linked List: ");

recursiveParse(head);

return 0;

}

## Minimum Distance Between Two Elements

#include <stdio.h>

#include <stdlib.h>

// Function to find minimum distance between two elements in an array

int minDistance(int arr[], int n, int x, int y) {

int minDist = n; // Initialize the result to a large value

int prev = -1;

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

if (arr[i] == x || arr[i] == y) {

if (prev != -1 && arr[i] != arr[prev]) {

minDist = (i - prev) < minDist ? (i - prev) : minDist;

}

prev = i;

}

}

return minDist;

}

int main() {

int arr[] = {4, 6, 7, 8, 6, 2, 2, 6, 4, 7};

int n = sizeof(arr) / sizeof(arr[0]);

int x = 4;

int y = 7;

int distance = minDistance(arr, n, x, y);

if (distance == n)

printf("One or both elements not present in the array\n");

else

printf("Minimum distance between %d and %d is: %d\n", x, y, distance);

return 0;

}

## Unique Elements

#include <stdio.h>

#include <stdbool.h>

// Function to check if an element exists in the hash set

bool contains(int \*hashSet, int size, int value) {

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

if (hashSet[i] == value) {

return true;

}

}

return false;

}

// Function to find unique elements in the array

int\* findUniqueElements(int \*arr, int size, int \*resultSize) {

// Assuming maximum array size is 100 (you can adjust this as needed)

int hashSet[100];

int uniqueCount = 0;

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

if (!contains(hashSet, uniqueCount, arr[i])) {

hashSet[uniqueCount] = arr[i];

uniqueCount++;

}

}

// Allocate memory for the result array

int \*resultArray = (int\*)malloc(uniqueCount \* sizeof(int));

// Copy unique elements to the result array

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

resultArray[i] = hashSet[i];

}

// Set the size of the result array

\*resultSize = uniqueCount;

return resultArray;

}

int main() {

int inputArray[] = {1, 2, 3, 2, 4, 5, 6, 4, 7, 8, 9, 1};

int size = sizeof(inputArray) / sizeof(inputArray[0]);

int resultSize;

int \*resultArray = findUniqueElements(inputArray, size, &resultSize);

printf("Unique elements in the array: ");

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

printf("%d ", resultArray[i]);

}

// Remember to free the dynamically allocated memory

free(resultArray);

return 0;

}

## Square A Sorted Array

#include <stdio.h>

#include <stdlib.h>

// Function prototypes

int\* sortedSquares(int\* nums, int numsSize, int\* returnSize);

// Function to square a sorted array and return a sorted array

int\* sortedSquares(int\* nums, int numsSize, int\* returnSize) {

    int\* result = (int\*)malloc(numsSize \* sizeof(int));

    \*returnSize = numsSize;

    int left = 0; // Pointer for the left end of the array

    int right = numsSize - 1; // Pointer for the right end of the array

    int index = numsSize - 1; // Index for the result array

    // Square the numbers and fill the result array from right to left

    while (left <= right) {

        if (abs(nums[left]) > abs(nums[right])) {

            result[index--] = nums[left] \* nums[left];

            left++;

        } else {

            result[index--] = nums[right] \* nums[right];

            right--;

        }

    }

    return result;

}

int main() {

    int nums[] = {-4, -2, 0, 1, 3, 5}; // Sample sorted array

    int numsSize = sizeof(nums) / sizeof(nums[0]); // Size of the array

    int returnSize; // Size of the returned array

    int\* result = sortedSquares(nums, numsSize, &returnSize);

    // Print the sorted squares array

    printf("Sorted Squares Array: ");

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

        printf("%d ", result[i]);

    }

    printf("\n");

    // Free dynamically allocated memory

    free(result);

    return 0;

}

OR

#include <stdio.h>

#include <stdlib.h>

// Function to square each element of a sorted array and return the sorted result

int\* squareSortedArray(int\* nums, int size) {

    int\* result = (int\*)malloc(size \* sizeof(int));

    // Squaring each element

    for (int i = 0; i < size; i++) {

        result[i] = nums[i] \* nums[i];

    }

    // Sorting the squared array using insertion sort

    for (int i = 1; i < size; i++) {

        int key = result[i];

        int j = i - 1;

        while (j >= 0 && result[j] > key) {

            result[j + 1] = result[j];

            j = j - 1;

        }

        result[j + 1] = key;

    }

    return result;

}

// Function to print an array

void printArray(int\* arr, int size) {

    for (int i = 0; i < size; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main() {

    // Example sorted array

    int nums[] = {-4, -2, 0, 2, 5};

    int size = sizeof(nums) / sizeof(nums[0]);

    // Square the sorted array and return the result

    int\* squaredArray = squareSortedArray(nums, size);

    // Print the squared and sorted array

    printf("Squared and Sorted Array: ");

    printArray(squaredArray, size);

    // Remember to free the dynamically allocated memory

    free(squaredArray);

    return 0;

}

## NTP Logger

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/time.h>

#include <syslog.h>

#include <ntp.h>

#define JAN\_1970 2208988800UL

#define DRIFT\_THRESHOLD 1000 // Adjust this threshold as needed

#define SYNC\_INTERVAL\_SECONDS 300 // Adjust synchronization interval as needed

// Function to log messages to syslog

void log\_message(const char \*message, int priority) {

openlog("NTPSyncExample", LOG\_PID | LOG\_NDELAY, LOG\_USER);

syslog(priority, "%s", message);

closelog();

}

// Function to handle clock drift

void handle\_clock\_drift(struct ntptimeval \*ntp\_time) {

// Calculate the difference between local time and NTP time

struct timeval local\_time;

gettimeofday(&local\_time, NULL);

// Calculate the time drift in milliseconds

long drift = (ntp\_time->time.tv\_sec - local\_time.tv\_sec) \* 1000 +

(ntp\_time->time.tv\_usec - local\_time.tv\_usec) / 1000;

// Log the clock drift

char message[100];

snprintf(message, sizeof(message), "Clock drift: %ld milliseconds", drift);

log\_message(message, LOG\_INFO);

// Adjust the local clock if the drift exceeds a certain threshold

if (labs(drift) > DRIFT\_THRESHOLD) {

struct timeval adjust\_time;

adjust\_time.tv\_sec = ntp\_time->time.tv\_sec;

adjust\_time.tv\_usec = ntp\_time->time.tv\_usec;

settimeofday(&adjust\_time, NULL);

// Log the clock adjustment

log\_message("Clock adjusted", LOG\_INFO);

}

}

int main() {

while (1) {

// Initialize NTP data structures

struct ntptimeval ntp\_time;

memset(&ntp\_time, 0, sizeof(ntp\_time));

// Initialize the NTP server address

char ntp\_server[] = "pool.ntp.org";

// Perform the NTP query

int result = ntp\_gettime(ntp\_server, &ntp\_time);

if (result == 0) {

// Convert NTP time to UNIX timestamp

time\_t unix\_time = ntp\_time.time.tv\_sec - JAN\_1970;

// Display the result

printf("NTP Time: %s", ctime(&unix\_time));

// Log the successful NTP synchronization

log\_message("NTP synchronization successful", LOG\_INFO);

// Handle clock drift

handle\_clock\_drift(&ntp\_time);

} else {

// Log the NTP query failure

log\_message("NTP query failed", LOG\_ERR);

fprintf(stderr, "NTP query failed\n");

}

// Sleep for a specific interval before the next synchronization

sleep(SYNC\_INTERVAL\_SECONDS);

}

return 0;

}

## Job Scheduler

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Structure to represent a job

typedef struct {

int job\_id;

time\_t scheduled\_time;

void (\*job\_function)();

} Job;

// Priority Queue Node

typedef struct {

Job \*job;

struct PriorityQueueNode \*next;

} PriorityQueueNode;

// Priority Queue

typedef struct {

PriorityQueueNode \*front;

} PriorityQueue;

// Function to create a new job

Job\* createJob(int job\_id, time\_t scheduled\_time, void (\*job\_function)()) {

Job \*newJob = (Job\*)malloc(sizeof(Job));

newJob->job\_id = job\_id;

newJob->scheduled\_time = scheduled\_time;

newJob->job\_function = job\_function;

return newJob;

}

// Function to create a new PriorityQueueNode

PriorityQueueNode\* createPriorityQueueNode(Job \*job) {

PriorityQueueNode \*node = (PriorityQueueNode\*)malloc(sizeof(PriorityQueueNode));

node->job = job;

node->next = NULL;

return node;

}

// Function to initialize a priority queue

PriorityQueue\* initializePriorityQueue() {

PriorityQueue \*queue = (PriorityQueue\*)malloc(sizeof(PriorityQueue));

queue->front = NULL;

return queue;

}

// Function to insert a job into the priority queue

void insertJob(PriorityQueue \*queue, Job \*job) {

PriorityQueueNode \*newNode = createPriorityQueueNode(job);

// If the queue is empty or the job's scheduled time is earlier than the front job, insert at the front

if (queue->front == NULL || job->scheduled\_time < queue->front->job->scheduled\_time) {

newNode->next = queue->front;

queue->front = newNode;

return;

}

// Traverse the queue to find the correct position for insertion

PriorityQueueNode \*current = queue->front;

while (current->next != NULL && current->next->job->scheduled\_time <= job->scheduled\_time) {

current = current->next;

}

newNode->next = current->next;

current->next = newNode;

}

// Function to execute the scheduled jobs

void executeJobs(PriorityQueue \*queue) {

while (queue->front != NULL) {

time\_t current\_time = time(NULL);

// Check if the front job is ready to be executed

if (current\_time >= queue->front->job->scheduled\_time) {

// Execute the job

printf("Executing Job %d at %s", queue->front->job->job\_id, ctime(&current\_time));

queue->front->job->job\_function();

// Remove the executed job from the queue

PriorityQueueNode \*temp = queue->front;

queue->front = queue->front->next;

free(temp);

} else {

// No jobs are ready to be executed, break out of the loop

break;

}

}

}

// Sample job functions

void jobFunction1() {

printf("Job Function 1\n");

}

void jobFunction2() {

printf("Job Function 2\n");

}

int main() {

// Initialize the priority queue

PriorityQueue \*jobQueue = initializePriorityQueue();

// Create sample jobs

Job \*job1 = createJob(1, time(NULL) + 5, jobFunction1);

Job \*job2 = createJob(2, time(NULL) + 10, jobFunction2);

// Insert jobs into the priority queue

insertJob(jobQueue, job1);

insertJob(jobQueue, job2);

// Simulate job execution

executeJobs(jobQueue);

// Free allocated memory

free(job1);

free(job2);

free(jobQueue);

return 0;

}

## Course Schedule II

/\*\*

 \* Note: The returned array must be malloced, assume caller calls free().

 \*/

struct Node {

    int val;

    struct Node\* next;

};

struct Graph {

    int numCourses;

    struct Node\*\* adjList;

    int\* inDegrees;

};

// Function to initialize a graph

struct Graph\* initializeGraph(int numCourses) {

    struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

    graph->numCourses = numCourses;

    graph->adjList = (struct Node\*\*)malloc(numCourses \* sizeof(struct Node\*));

    graph->inDegrees = (int\*)calloc(numCourses, sizeof(int));

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

        graph->adjList[i] = NULL;

    }

    return graph;

}

// Function to add a directed edge to the graph

void addEdge(struct Graph\* graph, int src, int dest) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->val = dest;

    newNode->next = graph->adjList[src];

    graph->adjList[src] = newNode;

    // Increment in-degree of the destination node

    graph->inDegrees[dest]++;

}

// Function to perform topological sort using Kahn's algorithm

int\* findOrder(int numCourses, int\*\* prerequisites, int prerequisitesSize, int\* prerequisitesColSize, int\* returnSize) {

    // Initialize the graph

    struct Graph\* graph = initializeGraph(numCourses);

    // Build the graph and compute in-degrees

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

        addEdge(graph, prerequisites[i][1], prerequisites[i][0]);

    }

    // Create a queue for topological sort

    int\* queue = (int\*)malloc(numCourses \* sizeof(int));

    int front = 0, rear = 0;

    // Enqueue nodes with in-degree zero

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

        if (graph->inDegrees[i] == 0) {

            queue[rear++] = i;

        }

    }

    // Initialize the result array

    int\* result = (int\*)malloc(numCourses \* sizeof(int));

    int index = 0;

    // Perform topological sort

    while (front < rear) {

        int current = queue[front++];

        result[index++] = current;

        // Update in-degrees of adjacent nodes

        struct Node\* temp = graph->adjList[current];

        while (temp != NULL) {

            graph->inDegrees[temp->val]--;

            if (graph->inDegrees[temp->val] == 0) {

                queue[rear++] = temp->val;

            }

            temp = temp->next;

        }

    }

    // Check if topological sort is possible

    if (index != numCourses) {

        \*returnSize = 0;

        free(result);

        result = NULL;

    } else {

        \*returnSize = numCourses;

    }

    // Free memory

    free(graph->inDegrees);

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

        struct Node\* temp = graph->adjList[i];

        while (temp != NULL) {

            struct Node\* nextNode = temp->next;

            free(temp);

            temp = nextNode;

        }

    }

    free(graph->adjList);

    free(graph);

    free(queue);

    return result;

}

## Cycle/Loop In Linked List

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Definition for singly-linked list

struct ListNode {

    int val;

    struct ListNode \*next;

};

// Function to detect cycle in a linked list

bool hasCycle(struct ListNode \*head) {

    if (head == NULL) return false;

    struct ListNode \*slow = head;

    struct ListNode \*fast = head;

    while (fast != NULL && fast->next != NULL) {

        slow = slow->next;

        fast = fast->next->next;

        if (slow == fast) return true;

    }

    return false;

}

int main() {

    // Creating linked list nodes

    struct ListNode\* head = (struct ListNode\*)malloc(sizeof(struct ListNode));

    struct ListNode\* second = (struct ListNode\*)malloc(sizeof(struct ListNode));

    struct ListNode\* third = (struct ListNode\*)malloc(sizeof(struct ListNode));

    struct ListNode\* fourth = (struct ListNode\*)malloc(sizeof(struct ListNode));

    // Assigning values and creating a cycle

    head->val = 1;

    head->next = second;

    second->val = 2;

    second->next = third;

    third->val = 3;

    third->next = fourth;

    fourth->val = 4;

    fourth->next = second; // Creating a cycle

    // Checking if there is a cycle

    bool result = hasCycle(head);

    if (result)

        printf("The linked list contains a cycle.\n");

    else

        printf("The linked list does not contain a cycle.\n");

    // Freeing allocated memory

    free(head);

    free(second);

    free(third);

    free(fourth);

    return 0;

}

## Version Sorting

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Comparison function for qsort to perform version sorting

int compareVersions(const void\* a, const void\* b) {

// Assuming versions are in the format "major.minor"

// Extract major and minor version components

// Convert version strings to integers

int majorA, minorA, majorB, minorB;

sscanf(\*(const char\*\*)a, "%d.%d", &majorA, &minorA);

sscanf(\*(const char\*\*)b, "%d.%d", &majorB, &minorB);

// Compare major versions first

if (majorA != majorB) {

return majorA - majorB;

}

// If major versions are equal, compare minor versions

return minorA - minorB;

}

int main() {

// Example array of version strings

const char\* versions[] = {"1.2", "2.0", "1.10", "2.1"};

// Calculate the number of versions

size\_t numVersions = sizeof(versions) / sizeof(versions[0]);

// Perform version sorting using qsort

qsort(versions, numVersions, sizeof(versions[0]), compareVersions);

// Print the sorted versions

printf("Sorted Versions:\n");

for (size\_t i = 0; i < numVersions; ++i) {

printf("%s\n", versions[i]);

}

return 0;

}

## Reverse Linked List Using Stack

#include <stdio.h>

#include <stdlib.h>

// Node structure for a singly linked list

struct Node {

int data;

struct Node\* next;

};

// Function to push a node onto the stack

void push(struct Node\*\* top, int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = \*top;

\*top = newNode;

}

// Function to pop a node from the stack

int pop(struct Node\*\* top) {

if (\*top == NULL) {

// Stack is empty

return -1;

}

struct Node\* temp = \*top;

int data = temp->data;

\*top = temp->next;

free(temp);

return data;

}

// Function to reverse a linked list using a stack

void reverseLinkedList(struct Node\*\* head) {

if (\*head == NULL || (\*head)->next == NULL) {

// Empty or single-node list, no need to reverse

return;

}

struct Node\* stackTop = NULL;

struct Node\* current = \*head;

// Push each element onto the stack

while (current != NULL) {

push(&stackTop, current->data);

current = current->next;

}

// Pop elements from the stack to reconstruct the reversed linked list

current = \*head;

while (current != NULL) {

current->data = pop(&stackTop);

current = current->next;

}

}

// Function to print a linked list

void printList(struct Node\* head) {

while (head != NULL) {

printf("%d -> ", head->data);

head = head->next;

}

printf("NULL\n");

}

int main() {

// Create a sample linked list

struct Node\* head = NULL;

for (int i = 1; i <= 5; i++) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = i;

newNode->next = head;

head = newNode;

}

// Print the original linked list

printf("Original Linked List: ");

printList(head);

// Reverse the linked list using a stack

reverseLinkedList(&head);

// Print the reversed linked list

printf("Reversed Linked List: ");

printList(head);

// Free the memory allocated for the linked list nodes

while (head != NULL) {

struct Node\* temp = head;

head = head->next;

free(temp);

}

return 0;

}

## Hexadecimal to decimal

#include <stdio.h>

// Function to convert hexadecimal to decimal without using strtol

unsigned long hexToDecimal(const char \*hexNumber) {

unsigned long decimalNumber = 0;

int i = 0;

// Skip the "0x" prefix if present

if (hexNumber[0] == '0' && (hexNumber[1] == 'x' || hexNumber[1] == 'X')) {

i = 2;

}

// Iterate through the characters of the hexadecimal string

while (hexNumber[i] != '\0') {

decimalNumber \*= 16;

// Convert character to corresponding decimal value

if (hexNumber[i] >= '0' && hexNumber[i] <= '9') {

decimalNumber += hexNumber[i] - '0';

} else if (hexNumber[i] >= 'A' && hexNumber[i] <= 'F') {

decimalNumber += hexNumber[i] - 'A' + 10;

} else if (hexNumber[i] >= 'a' && hexNumber[i] <= 'f') {

decimalNumber += hexNumber[i] - 'a' + 10;

} else {

// Invalid character in the hexadecimal string

printf("Invalid hexadecimal character: %c\n", hexNumber[i]);

return 0;

}

i++;

}

return decimalNumber;

}

int main() {

// Hexadecimal number as a string

const char \*hexNumber = "1A";

// Using custom function for hexadecimal to decimal conversion

unsigned long decimalNumber = hexToDecimal(hexNumber);

// Printing the result

printf("Hexadecimal: %s\n", hexNumber);

printf("Decimal: %lu\n", decimalNumber);

return 0;

}

## Decimal To Hexadecimal

#include <stdio.h>

// Function to convert decimal to hexadecimal without using printf

void decimalToHexadecimal(unsigned long decimalNumber) {

char hexadecimal[20]; // Assuming a reasonable maximum length for the result

int i = 0;

// Iterate until the decimal number becomes 0

while (decimalNumber > 0) {

int remainder = decimalNumber % 16;

// Convert remainder to hexadecimal character

if (remainder < 10) {

hexadecimal[i] = remainder + '0';

} else {

hexadecimal[i] = remainder - 10 + 'A';

}

decimalNumber /= 16;

i++;

}

// Print the hexadecimal representation in reverse order

printf("Hexadecimal: ");

for (int j = i - 1; j >= 0; j--) {

printf("%c", hexadecimal[j]);

}

printf("\n");

}

int main() {

// Decimal number

unsigned long decimalNumber = 26;

// Using custom function for decimal to hexadecimal conversion

decimalToHexadecimal(decimalNumber);

return 0;

}

## Absolute Path From Relative Path

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int main() {

// Relative path

const char \*relativePath = "myfolder/myfile.txt";

// Buffer to store the absolute path

char absolutePath[PATH\_MAX];

// Using realpath to find the absolute path

char \*result = realpath(relativePath, absolutePath);

// Check if realpath was successful

if (result != NULL) {

printf("Relative Path: %s\n", relativePath);

printf("Absolute Path: %s\n", absolutePath);

} else {

perror("realpath");

return EXIT\_FAILURE;

}

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <limits.h>

int main() {

// Relative path

const char \*relativePath = "myfolder/myfile.txt";

// Buffer to store the absolute path

char absolutePath[PATH\_MAX];

// Get the current working directory

if (getcwd(absolutePath, sizeof(absolutePath)) != NULL) {

// Concatenate the relative path to the current working directory

snprintf(absolutePath, sizeof(absolutePath), "%s/%s", absolutePath, relativePath);

printf("Relative Path: %s\n", relativePath);

printf("Absolute Path: %s\n", absolutePath);

} else {

perror("getcwd");

return EXIT\_FAILURE;

}

return 0;

}

## LockLess programming

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <stdatomic.h>

// Define a global atomic variable for the counter

\_Atomic int counter = 0;

// Function to increment the counter atomically

void\* increment\_counter(void\* arg) {

for (int i = 0; i < 1000000; i++) {

// Atomically increment the counter

atomic\_fetch\_add(&counter, 1);

}

return NULL;

}

int main() {

// Create two threads to increment the counter concurrently

pthread\_t thread1, thread2;

pthread\_create(&thread1, NULL, increment\_counter, NULL);

pthread\_create(&thread2, NULL, increment\_counter, NULL);

// Wait for the threads to finish

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

// Print the final value of the counter

printf("Final counter value: %d\n", counter);

return 0;

}

## IOCTL

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <net/if.h>

#include <sys/ioctl.h>

#include <linux/mii.h>

#include <linux/sockios.h>

#include<errno.h>

char \*name = NULL;

int phy = 0;

int control\_register\_value = 0;

int choice = 0;

int cr[10] = {0x0040,0x0080,0x0100,0x0200,0x0400,

0x0800,0x1000,0x2000,0x4000,0x8000};

char control\_register[10][20] = {

"BMCR\_SPEED1000", "BMCR\_CTST", "BMCR\_FULLDPLX",

"BMCR\_ANRESTART", "BMCR\_ISOLATE", "BMCR\_PDOWN",

"BMCR\_ANENABLE","BMCR\_SPEED100", "BMCR\_LOOPBACK", "BMCR\_RESET"

};

void read\_mii\_register(int fd, char \*name) {

struct ifreq ifr;

memset(&ifr, 0, sizeof(ifr));

strcpy(ifr.ifr\_name, name);

if (ioctl(fd, SIOCGMIIPHY, &ifr) < 0) {

printf("\nPHY FAILED!");

return;

}

struct mii\_ioctl\_data\* mii = (struct mii\_ioctl\_data\*)(&ifr.ifr\_data);

/\*Read status register\*/

mii->reg\_num = MII\_BMSR;

mii->val\_in = 0;

mii->val\_out = 0;

if (ioctl(fd, SIOCGMIIREG, &ifr) != -1){

if ((mii->val\_out != 0xffff) || (mii->val\_out != 0x0)) {

printf("\nMII Transceiver Found for %s and phy\_id is %d!",name,mii->phy\_id);

}

else {

printf("\nNo MII Transceiver Found for %s and phy\_id is %d!",name,mii->phy\_id);

return;

}

phy = mii->phy\_id;

printf("\n========================= BASIC MODE STATUS REGISTER MII\_BMSR : 0x%x =========================\n",mii->val\_out);

printf("BMSR\_ERCAP(External Register Capability) = %d \n",(mii->val\_out & BMSR\_ERCAP) ? 1 : 0);

printf("BMSR\_JCD(Jabber Detected) = %d \n",(mii->val\_out & BMSR\_JCD) ? 1 : 0);

printf("BMSR\_LSTATUS(Link Status) = %d \n",(mii->val\_out & BMSR\_LSTATUS) ? 1 : 0);

printf("BMSR\_ANEGCAPABLE(Auto-negotiation) = %d \n",(mii->val\_out & BMSR\_ANEGCAPABLE) ? 1 : 0);

printf("BMSR\_RFAULT(Remote Fault) = %d \n",(mii->val\_out & BMSR\_RFAULT) ? 1 : 0);

printf("BMSR\_ANEGCOMPLETE(Auto-negotiation Complete) = %d \n",(mii->val\_out & BMSR\_ANEGCOMPLETE) ? 1 : 0);

printf("BMSR\_ESTATEN(Extended Status) = %d \n",(mii->val\_out & BMSR\_ESTATEN) ? 1 : 0);

printf("BMSR\_100HALF2(100Base-T2 Half Duplex) = %d \n",(mii->val\_out & BMSR\_100HALF2) ? 1 : 0);

printf("BMSR\_100FULL2(100Base-T2 Full Duplex) = %d \n",(mii->val\_out & BMSR\_100FULL2) ? 1 : 0);

printf("BMSR\_10HALF(10Mbps Half Duplex) = %d \n",(mii->val\_out & BMSR\_10HALF) ? 1 : 0);

printf("BMSR\_10FULL(10Mbps Full Duplex) = %d \n",(mii->val\_out & BMSR\_10FULL) ? 1 : 0);

printf("BMSR\_100HALF(100Mbps Half Duplex) = %d \n",(mii->val\_out & BMSR\_100HALF) ? 1 : 0);

printf("BMSR\_100BASE4(100Mbps 4K Packets) = %d \n",(mii->val\_out & BMSR\_100BASE4) ? 1 : 0);

}else{

printf(" MII\_BMSR register read returns -1\n");

return;

}

/\*Read control register\*/

mii->reg\_num = MII\_BMCR;

mii->val\_in = 0;

mii->val\_out = 0;

if (ioctl(fd, SIOCGMIIREG, &ifr) != -1){

printf("\n========================= BASIC MODE CONTROL REGISTER MII\_BMCR : 0x%x =========================\n",mii->val\_out);

printf("BMCR\_SPEED1000(MSB of Link Speed) = %d \n",(mii->val\_out & BMCR\_SPEED1000) ? 1 : 0);

printf("BMCR\_CTST(Collission test) = %d \n",(mii->val\_out & BMCR\_CTST) ? 1 : 0);

printf("BMCR\_FULLDPLX(Full Duplex) = %d \n",(mii->val\_out & BMCR\_FULLDPLX) ? 1 : 0);

printf("BMCR\_ANRESTART(Auto-negotiation restart) = %d \n",(mii->val\_out & BMCR\_ANRESTART) ? 1 : 0);

printf("BMCR\_ISOLATE(Disconnect DP83840 Physical Layer from MII) = %d \n",(mii->val\_out & BMCR\_ISOLATE) ? 1 : 0);

printf("BMCR\_PDOWN(Power down DP83840) = %d \n",(mii->val\_out & BMCR\_PDOWN) ? 1 : 0);

printf("BMCR\_ANENABLE(Enable Auto-negotiation) = %d \n",(mii->val\_out & BMCR\_ANENABLE) ? 1 : 0);

printf("BMCR\_SPEED100(Select 100Mbps) = %d \n",(mii->val\_out & BMCR\_SPEED100) ? 1 : 0);

printf("BMCR\_LOOPBACK(TXD Loopback bits) = %d \n",(mii->val\_out & BMCR\_LOOPBACK) ? 1 : 0);

printf("BMCR\_RESET(Reset DP83840) = %d \n",(mii->val\_out & BMCR\_RESET) ? 1 : 0);

control\_register\_value = mii->val\_out;

}else{

printf(" MII\_BMCR register read returns -1\n");

}

return;

}

void write\_mii\_register(int fd, char \*name, int choice) {

printf("\nPreferred to write register %s at location : 0x%x\n",control\_register[choice], cr[choice]);

struct ifreq ifr;

memset(&ifr, 0, sizeof(ifr));

strcpy(ifr.ifr\_name, name);

if (ioctl(fd, SIOCGMIIPHY, &ifr) < 0) {

printf("\nPHY FAILED!");

return;

}

struct mii\_ioctl\_data\* mii = (struct mii\_ioctl\_data\*)(&ifr.ifr\_data);

mii->phy\_id = phy;

mii->reg\_num = MII\_BMCR;

if (strcmp(name, "eth6") == 0) {

mii->val\_in = (control\_register\_value ^ cr[choice]);

}

else {

mii->val\_in = (control\_register\_value | cr[choice]);

}

if (ioctl(fd, SIOCSMIIREG, &ifr) < 0) {

fprintf(stderr,"\nwrite failure %s",strerror(errno));

}

else {

fprintf(stderr,"\nwrite success %s",strerror(errno));

}

}

int main(int argc, char \*argv[]){

if (argc<3) {

printf("\nUsage ./phy\_register\_util [r/w] [dev\_name]\n");

return 0;

}

const int fd = socket(AF\_INET, SOCK\_DGRAM, 0);

if (fd < 0) {

printf("\nSocket error!");

return -1;

}

name = argv[2];

if (strcmp(argv[1],"r") == 0) {

read\_mii\_register(fd,name);

}

else if (strcmp(argv[1],"w") == 0) {

printf("\nAvailable registers to write:");

printf("\n0: BMCR\_SPEED1000(MSB of Link Speed)\n1: BMCR\_CTST(Collission test)\

\n2: BMCR\_FULLDPLX(Full Duplex)\n3: BMCR\_ANRESTART(Auto-negotiation restart)\

\n4: BMCR\_ISOLATE(Disconnect DP83840 Physical Layer from MII)\

\n5: BMCR\_PDOWN(Power down DP83840)\n6: BMCR\_ANENABLE(Enable Auto-negotiation)\

\n7: BMCR\_SPEED100(Select 100Mbps)\n8: BMCR\_LOOPBACK(TXD Loopback bits)\

\n9: BMCR\_RESET(Reset DP83840)\n");

printf("\nSelect one from above:");

scanf("%d",&choice);

write\_mii\_register(fd,name,choice);

}

else {

printf("\nUsage ./phy\_register\_util [r/w] [dev\_name]\n");

return 0;

}

close(fd);

}

## I2C

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ioctl.h>

#include <linux/i2c-dev.h>

#define I2C\_DEVICE "/dev/i2c-1" // Change this to match your I2C device file

// Function to initialize the I2C bus and device

int initI2C(int addr) {

int file;

// Open the I2C device file

if ((file = open(I2C\_DEVICE, O\_RDWR)) < 0) {

perror("Failed to open the I2C device");

return -1;

}

// Set the I2C slave address

if (ioctl(file, I2C\_SLAVE, addr) < 0) {

perror("Failed to set I2C slave address");

close(file);

return -1;

}

return file;

}

// Function to write data to the I2C device

int writeI2C(int file, unsigned char \*data, int len) {

if (write(file, data, len) != len) {

perror("Failed to write to the I2C device");

return -1;

}

return 0;

}

// Function to read data from the I2C device

int readI2C(int file, unsigned char \*data, int len) {

if (read(file, data, len) != len) {

perror("Failed to read from the I2C device");

return -1;

}

return 0;

}

// Function to close the I2C bus

void closeI2C(int file) {

close(file);

}

int main() {

int file;

unsigned char data[2];

// Initialize the I2C bus and device

if ((file = initI2C(0x68)) < 0) { // Replace 0x68 with the actual device address

exit(EXIT\_FAILURE);

}

// Example: Write to a register on the device

data[0] = 0x00; // Register address

data[1] = 0x55; // Data to be written

if (writeI2C(file, data, 2) < 0) {

closeI2C(file);

exit(EXIT\_FAILURE);

}

// Example: Read from a register on the device

data[0] = 0x00; // Register address to read from

if (writeI2C(file, data, 1) < 0) {

closeI2C(file);

exit(EXIT\_FAILURE);

}

if (readI2C(file, data, 1) < 0) {

closeI2C(file);

exit(EXIT\_FAILURE);

}

printf("Read data: 0x%02X\n", data[0]);

// Close the I2C bus

closeI2C(file);

return 0;

}

## Large File Word Frequency

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_WORD\_LENGTH 100

#define HASH\_TABLE\_SIZE 10000

// Node structure for the hash table

typedef struct Node {

char\* word;

int count;

struct Node\* next;

} Node;

// Function to create a new node

Node\* createNode(char\* word) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

if (newNode == NULL) {

fprintf(stderr, "Memory allocation failed\n");

exit(1);

}

newNode->word = strdup(word);

newNode->count = 1;

newNode->next = NULL;

return newNode;

}

// Function to calculate the hash value of a word

unsigned int hash(char\* word) {

unsigned int hashValue = 0;

while (\*word) {

hashValue = (hashValue << 5) + \*word++;

}

return hashValue % HASH\_TABLE\_SIZE;

}

// Function to insert a word into the hash table

void insert(Node\*\* hashTable, char\* word) {

unsigned int index = hash(word);

Node\* current = hashTable[index];

while (current != NULL) {

if (strcmp(current->word, word) == 0) {

current->count++;

return;

}

current = current->next;

}

// Word not found, insert new node at the beginning of the list

Node\* newNode = createNode(word);

newNode->next = hashTable[index];

hashTable[index] = newNode;

}

// Function to find the word with the highest frequency

char\* findMostFrequentWord(Node\*\* hashTable) {

int maxCount = 0;

char\* mostFrequentWord = NULL;

for (int i = 0; i < HASH\_TABLE\_SIZE; i++) {

Node\* current = hashTable[i];

while (current != NULL) {

if (current->count > maxCount) {

maxCount = current->count;

mostFrequentWord = current->word;

}

current = current->next;

}

}

return mostFrequentWord;

}

int main() {

// Initialize hash table

Node\* hashTable[HASH\_TABLE\_SIZE] = {NULL};

// Open the file

FILE\* file = fopen("large\_file.txt", "r");

if (file == NULL) {

fprintf(stderr, "Failed to open file\n");

exit(1);

}

// Read the file word by word

char word[MAX\_WORD\_LENGTH];

while (fscanf(file, "%s", word) != EOF) {

// Insert each word into the hash table

insert(hashTable, word);

}

fclose(file);

// Find the word with the highest frequency of occurrence

char\* mostFrequentWord = findMostFrequentWord(hashTable);

// Print the result

if (mostFrequentWord != NULL) {

printf("Most frequent word: %s\n", mostFrequentWord);

} else {

printf("No words found in the file\n");

}

// Free memory allocated for the hash table

for (int i = 0; i < HASH\_TABLE\_SIZE; i++) {

Node\* current = hashTable[i];

while (current != NULL) {

Node\* temp = current;

current = current->next;

free(temp->word);

free(temp);

}

}

return 0;

}

## Compressed String

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* compressString(const char\* str) {

int length = strlen(str);

if (length == 0) return NULL; // Handle empty string

char\* result = (char\*)malloc(sizeof(char) \* (2 \* length + 1)); // Maximum length of compressed string

if (result == NULL) {

fprintf(stderr, "Memory allocation failed\n");

exit(1);

}

char current\_char = str[0];

int count = 1;

int index = 0;

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

if (str[i] == current\_char) {

count++;

} else {

// Write compressed character and count to the result string

index += sprintf(&result[index], "%c%d", current\_char, count);

current\_char = str[i];

count = 1;

}

}

return result;

}

int main() {

const char\* str = "aaabbccccddaaeeeee";

char\* compressed = compressString(str);

if (compressed != NULL) {

printf("Compressed string: %s\n", compressed);

free(compressed); // Free dynamically allocated memory

} else {

printf("Failed to compress string\n");

}

return 0;

}

## Remove Node from Linked List

#include <stdio.h>

#include <stdlib.h>

// Definition for singly-linked list node

struct ListNode {

    int val;

    struct ListNode \*next;

};

// Function to create a new node

struct ListNode\* createNode(int val) {

    struct ListNode\* newNode = (struct ListNode\*)malloc(sizeof(struct ListNode));

    if (newNode != NULL) {

        newNode->val = val;

        newNode->next = NULL;

    }

    return newNode;

}

// Function to insert a new node at the end of the linked list

void insertNode(struct ListNode\*\* head, int val) {

    struct ListNode\* newNode = createNode(val);

    if (\*head == NULL) {

        \*head = newNode;

    } else {

        struct ListNode\* temp = \*head;

        while (temp->next != NULL) {

            temp = temp->next;

        }

        temp->next = newNode;

    }

}

// Function to remove a node from the linked list

void removeNode(struct ListNode\*\* head, int val) {

    struct ListNode\* curr = \*head;

    struct ListNode\* prev = NULL;

    // Traverse the list to find the node to be removed

    while (curr != NULL && curr->val != val) {

        prev = curr;

        curr = curr->next;

    }

    // If the node is found

    if (curr != NULL) {

        // If the node to be removed is the head node

        if (prev == NULL) {

            \*head = curr->next;

        } else {

            prev->next = curr->next;

        }

        free(curr);

    }

}

// Function to print the linked list

void printList(struct ListNode\* head) {

    struct ListNode\* curr = head;

    while (curr != NULL) {

        printf("%d ", curr->val);

        curr = curr->next;

    }

    printf("\n");

}

int main() {

    struct ListNode\* head = NULL;

    // Insert some nodes

    insertNode(&head, 1);

    insertNode(&head, 2);

    insertNode(&head, 3);

    insertNode(&head, 4);

    insertNode(&head, 5);

    printf("Original list: ");

    printList(head);

    // Remove node with value 3

    removeNode(&head, 3);

    printf("List after removing node with value 3: ");

    printList(head);

    // Remove node with value 1 (head node)

    removeNode(&head, 1);

    printf("List after removing node with value 1: ");

    printList(head);

    // Remove node with value 5 (last node)

    removeNode(&head, 5);

    printf("List after removing node with value 5: ");

    printList(head);

    // Free memory allocated for the list nodes

    struct ListNode\* temp;

    while (head != NULL) {

        temp = head;

        head = head->next;

        free(temp);

    }

    return 0;

}

## DOS attack detection from file

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#define MAX\_ATTACKS 100

#define THRESHOLD 10 // Adjust the threshold as needed

// Define a structure to represent network traffic data

struct NetworkData {

    char timestamp[20];

    char sourceIP[20];

    int sourcePort;

    char destinationIP[20];

    int destinationPort;

};

// Define a structure to represent potential DoS attacks

struct DoSAttack {

    char timestamp[20];

    char sourceIP[20];

    int sourcePort;

    int count;

};

// Function to check if two network data entries are identical

bool isSameData(struct NetworkData data1, struct NetworkData data2) {

    return strcmp(data1.sourceIP, data2.sourceIP) == 0 &&

           data1.sourcePort == data2.sourcePort &&

           strcmp(data1.destinationIP, data2.destinationIP) == 0 &&

           data1.destinationPort == data2.destinationPort;

}

// Function to read network data from file and detect potential DoS attacks

void detectDoS(FILE\* file) {

    struct NetworkData data;

    struct DoSAttack attacks[MAX\_ATTACKS];

    int attackCount = 0;

    // Read each line from the file

    while (fscanf(file, "%s %s %d %s %d", data.timestamp, data.sourceIP, &data.sourcePort, data.destinationIP, &data.destinationPort) != EOF) {

        // Check if the current data matches any previous entry

        bool found = false;

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

            if (isSameData(data, attacks[i])) {

                attacks[i].count++;

                found = true;

                break;

            }

        }

        // If no match found, consider it as a new attack

        if (!found) {

            strcpy(attacks[attackCount].timestamp, data.timestamp);

            strcpy(attacks[attackCount].sourceIP, data.sourceIP);

            attacks[attackCount].sourcePort = data.sourcePort;

            attacks[attackCount].count = 1;

            attackCount++;

        }

    }

    // Print detected potential DoS attacks

    printf("Potential DoS Attacks:\n");

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

        if (attacks[i].count >= THRESHOLD) {

            printf("Timestamp: %s, Source IP: %s, Source Port: %d, Count: %d\n", attacks[i].timestamp, attacks[i].sourceIP, attacks[i].sourcePort, attacks[i].count);

        }

    }

}

int main() {

    FILE\* file = fopen("network\_data.txt", "r");

    if (file == NULL) {

        perror("Error opening file");

        return EXIT\_FAILURE;

    }

    detectDoS(file);

    fclose(file);

    return 0;

}

## TCAM

#include <stdio.h>

#include <stdlib.h>

// Define TCAM entry structure

typedef struct {

int key;

int mask;

char\* value;

} TCAM\_Entry;

// Define TCAM structure

typedef struct {

int size;

TCAM\_Entry\* entries;

} TCAM;

// Initialize TCAM

TCAM\* initialize\_tcam(int size) {

TCAM\* tcam = (TCAM\*)malloc(sizeof(TCAM));

tcam->size = size;

tcam->entries = (TCAM\_Entry\*)malloc(size \* sizeof(TCAM\_Entry));

return tcam;

}

// Insert entry into TCAM

void insert\_entry(TCAM\* tcam, int key, int mask, char\* value) {

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

if (tcam->entries[i].value == NULL) {

tcam->entries[i].key = key;

tcam->entries[i].mask = mask;

tcam->entries[i].value = value;

return;

}

}

printf("TCAM is full. Cannot insert entry.\n");

}

// Lookup value in TCAM based on key and mask

char\* lookup\_value(TCAM\* tcam, int key) {

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

if ((key & tcam->entries[i].mask) == tcam->entries[i].key) {

return tcam->entries[i].value;

}

}

return NULL; // No matching entry found

}

// Main function

int main() {

TCAM\* tcam = initialize\_tcam(4);

insert\_entry(tcam, 0b1010, 0b1111, "Value 1");

insert\_entry(tcam, 0b1100, 0b1100, "Value 2");

char\* result1 = lookup\_value(tcam, 0b1011);

char\* result2 = lookup\_value(tcam, 0b1101);

char\* result3 = lookup\_value(tcam, 0b1111);

printf("Lookup result 1: %s\n", result1 ? result1 : "Not found");

printf("Lookup result 2: %s\n", result2 ? result2 : "Not found");

printf("Lookup result 3: %s\n", result3 ? result3 : "Not found");

free(tcam->entries);

free(tcam);

return 0;

}

## Spin Locks

Spinlocks are a synchronization primitive used in concurrent programming to protect shared resources from simultaneous access by multiple threads or processes. Unlike traditional mutex locks, which put the calling thread to sleep when the lock is held by another thread, spinlocks continuously "spin" or loop until the lock becomes available.

When a thread attempts to acquire a spinlock that is already held by another thread, it will keep checking the lock repeatedly in a tight loop, without yielding the processor. This means that the thread consumes CPU resources while waiting for the lock to become available.

Spinlocks are typically used in situations where the expected wait time for the lock to become available is very short, and the overhead of putting the thread to sleep and waking it up again would be excessive. They are commonly used in kernel-level programming and in situations where lock contention is rare or expected to be brief.

However, spinlocks are not suitable for scenarios where the lock is expected to be held for a long time, as spinning consumes CPU resources and can lead to resource wastage and degraded performance. In such cases, other synchronization primitives like mutex locks or semaphore should be used instead.

#include <stdio.h>

#include <pthread.h>

#include <stdatomic.h>

#define NUM\_THREADS 4

#define NUM\_INCREMENTS 1000000

// Define spinlock structure

typedef struct {

    atomic\_flag lock;

} spinlock\_t;

// Initialize the spinlock

void spinlock\_init(spinlock\_t \*lock) {

    atomic\_flag\_clear(&lock->lock);

}

// Acquire the spinlock

void spinlock\_lock(spinlock\_t \*lock) {

    // Spin until we successfully set the lock flag

    while (atomic\_flag\_test\_and\_set(&lock->lock)) {

        // Another thread holds the lock, so keep spinning

    }

}

// Release the spinlock

void spinlock\_unlock(spinlock\_t \*lock) {

    atomic\_flag\_clear(&lock->lock);

}

// Shared counter variable

volatile int counter = 0;

spinlock\_t counter\_lock;

// Thread function to increment the counter

void\* increment\_counter(void\* arg) {

    for (int i = 0; i < NUM\_INCREMENTS; i++) {

        spinlock\_lock(&counter\_lock);

        counter++;

        spinlock\_unlock(&counter\_lock);

    }

    return NULL;

}

int main() {

    // Initialize the spinlock

    spinlock\_init(&counter\_lock);

    // Create an array of threads

    pthread\_t threads[NUM\_THREADS];

    // Create threads

    for (int i = 0; i < NUM\_THREADS; i++) {

        pthread\_create(&threads[i], NULL, increment\_counter, NULL);

    }

    // Wait for threads to finish

    for (int i = 0; i < NUM\_THREADS; i++) {

        pthread\_join(threads[i], NULL);

    }

    // Print the final value of the counter

    printf("Final counter value: %d\n", counter);

    return 0;

}

**Memory Barrier and Compare and Swap(CAS):**

#include <stdio.h>

#include <pthread.h>

#include <stdatomic.h>

#define NUM\_THREADS 4

#define NUM\_INCREMENTS 1000000

// Define a shared variable

int counter = 0;

// Memory barrier

#define MEMBAR() \_\_asm\_\_ \_\_volatile\_\_("": : :"memory")

// CAS operation

#define CAS(ptr, oldval, newval) \

    \_\_sync\_bool\_compare\_and\_swap(ptr, oldval, newval)

// Thread function to increment the counter

void\* increment\_counter(void\* arg) {

    for (int i = 0; i < NUM\_INCREMENTS; i++) {

        int oldval, newval;

        do {

            // Load the current value of the counter

            oldval = counter;

            // Increment the value

            newval = oldval + 1;

            // Memory barrier

            MEMBAR();

        } while (!CAS(&counter, oldval, newval)); // CAS operation

    }

    return NULL;

}

int main() {

    // Create an array of threads

    pthread\_t threads[NUM\_THREADS];

    // Create threads

    for (int i = 0; i < NUM\_THREADS; i++) {

        pthread\_create(&threads[i], NULL, increment\_counter, NULL);

    }

    // Wait for threads to finish

    for (int i = 0; i < NUM\_THREADS; i++) {

        pthread\_join(threads[i], NULL);

    }

    // Print the final value of the counter

    printf("Final counter value: %d\n", counter);

    return 0;

}

**Barriers:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NUM\_THREADS 5

pthread\_barrier\_t barrier;

void\* thread\_function(void\* arg) {

    int thread\_id = \*((int\*)arg);

    printf("Thread %d waiting at the barrier\n", thread\_id);

    pthread\_barrier\_wait(&barrier);

    printf("Thread %d passed the barrier\n", thread\_id);

    return NULL;

}

int main() {

    pthread\_t threads[NUM\_THREADS];

    int thread\_ids[NUM\_THREADS];

    // Initialize the barrier with the number of threads

    pthread\_barrier\_init(&barrier, NULL, NUM\_THREADS);

    // Create threads

    for (int i = 0; i < NUM\_THREADS; ++i) {

        thread\_ids[i] = i;

        if (pthread\_create(&threads[i], NULL, thread\_function, &thread\_ids[i]) != 0) {

            perror("pthread\_create");

            exit(EXIT\_FAILURE);

        }

    }

    // Join threads

    for (int i = 0; i < NUM\_THREADS; ++i) {

        if (pthread\_join(threads[i], NULL) != 0) {

            perror("pthread\_join");

            exit(EXIT\_FAILURE);

        }

    }

    // Destroy the barrier

    pthread\_barrier\_destroy(&barrier);

    return 0;

}

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

// Define a spin lock structure

typedef struct {

int flag;

} spinlock\_t;

// Initialize the spin lock

void spinlock\_init(spinlock\_t \*lock) {

lock->flag = 0;

}

// Acquire the spin lock

void spinlock\_lock(spinlock\_t \*lock) {

while (\_\_sync\_lock\_test\_and\_set(&lock->flag, 1)) {}

}

// Release the spin lock

void spinlock\_unlock(spinlock\_t \*lock) {

\_\_sync\_lock\_release(&lock->flag);

}

// Function to be executed by multiple threads

void \*thread\_function(void \*arg) {

spinlock\_t \*lock = (spinlock\_t \*)arg;

// Acquire the spin lock

spinlock\_lock(lock);

// Critical section

printf("Thread %ld acquired the lock\n", pthread\_self());

// Do some work...

// Release the spin lock

spinlock\_unlock(lock);

return NULL;

}

int main() {

// Initialize the spin lock

spinlock\_t lock;

spinlock\_init(&lock);

// Create multiple threads

pthread\_t threads[5];

for (int i = 0; i < 5; i++) {

pthread\_create(&threads[i], NULL, thread\_function, &lock);

}

// Join the threads

for (int i = 0; i < 5; i++) {

pthread\_join(threads[i], NULL);

}

return 0;

}

## Synchronization Primitives

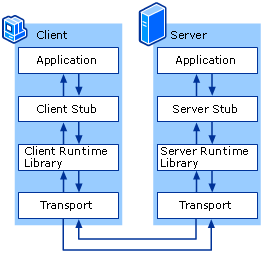
Synchronization primitives in multithreading are mechanisms used to coordinate the execution of multiple threads to ensure correct and consistent behavior of shared resources. They help prevent race conditions, data races, and other concurrency issues that can occur when multiple threads access shared data concurrently. Some common synchronization primitives include:

1. Mutexes (Mutual Exclusion): Mutexes are locks that allow only one thread to access a shared resource at a time. They provide exclusive access to critical sections of code, ensuring that only one thread can execute the protected code block at any given time.
2. Semaphores: Semaphores are a more generalized synchronization primitive that can be used to control access to a shared resource by multiple threads. Semaphores maintain a count that can be incremented or decremented atomically by threads. They can be used to limit the number of threads allowed to access a resource simultaneously or to signal the availability of resources.
3. Condition Variables: Condition variables are used to coordinate the execution of threads based on certain conditions. They allow threads to wait for a particular condition to become true before proceeding with their execution. Condition variables are typically used in conjunction with mutexes to protect shared data while waiting for a condition to be satisfied.
4. Barriers: Barriers are synchronization primitives that allow a group of threads to wait for each other to reach a certain point in their execution before proceeding further. Barriers are often used in parallel algorithms where multiple threads need to synchronize their progress at specific stages of computation.
5. Atomic Operations: Atomic operations provide a way to perform operations on shared variables atomically, without the need for explicit locking mechanisms. Common atomic operations include compare-and-swap (CAS), fetch-and-add, and load-linked/store-conditional (LL/SC) operations. These operations ensure that modifications to shared variables are performed atomically, without the risk of data corruption due to concurrent access by multiple threads.

These synchronization primitives, along with others, form the building blocks for implementing thread-safe and efficient concurrent algorithms in multithreaded programming environments. The choice of synchronization primitive depends on the specific requirements of the application and the level of concurrency control needed to ensure correct behavior.

## RPC (Remote Procedure Call)

**Message passing in RPC**

* The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
* The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
* The client’s local operating system sends the message from the client machine to the server machine.
* The local operating system on the server machine passes the incoming packets to the server stub.
* The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshalling.
* Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

**RPC Application Development**

Here i’m using UBUNTU OS and C as the programming language in my simple example.

Steps involved in developing a RPC application developing:

* Specify the protocol for client server communication
* Develop the client program
* Develop the server program

##### Step1  : Specify the protocol for client server communication

An interface description language (IDL) to let various platforms call the RPC. The IDL files can then be used to generate code to interface between the client and servers. So, in IDL file, we define the program structure like below. Save this IDL file with .x extension.

calculate.x

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **struct** inputs{  **float** num1;  **float** num2;  **char** operator;  };    program CALCULATE\_PROG{   version CALCULATE\_VER{  **float** ADD(inputs)=1;  **float** SUB(inputs)=2;  **float** MUL(inputs)=3;  **float** DIV(inputs)=4;     }=1;  }=0x2fffffff; |

**inputs** — Name of the data structure. Through this data structure the parameters are send to the server for computations.

**CALCULATE\_PROG** — Name of the program

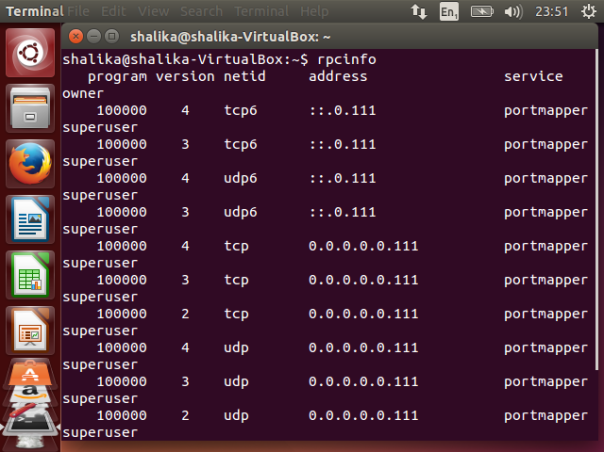
**CALCULATE\_VER** — Name of the program version

**ADD(inputs)** — This is a remote method which is calling locally and the parameters are passed to that remote method through the inputs structure which contains 2 operands(2 numbers) and an operator.

A remote procedure is uniquely identified by the triple: (**program number, version number, procedure number)**. Therefore, you need to give any numbers you like to the program, to the version and to the procedures as in the above sample IDL file.

Then compile your IDL file using rpcgen protocol compiler. The protocol compiler reads the definition of the IDL file and automatically generates client and server stubs. First you need to check whether the rpcbind has been installed in your machine. Type this command to check it and if the execution of the command gives you a long list as below in the image, then you machine has already installed rpcbind package and you can work on with the rpc commands.

$ rpcinfo



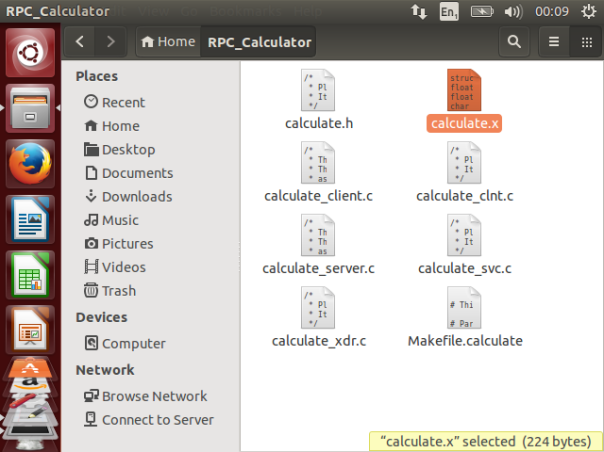
Otherwise, you have to install the rpcbind package by following below command and check whether it has been actually installed by typing the $ rpcinfo command again.

$ sudo apt-get install rpcbind

Now your PC is ready to execute rpc commands. The next step is to compile your IDL file using **rpcgen** command.

$ rpcgen -a -C calculate.x

This command generated 7 additional files as displayed in the below picture. Keep in mind that, you can edit only the ‘calculate\_client.c’ and ‘calculate\_server.c’ files among from the created files according to your need. Don’t do changes to other files.



* **calculate\_client.c** —-> client program (editable file)
* **calculate\_server.c** —> server program (editable file)
* **calculate\_cln.c** —> client stub
* **calculate\_svc.c** —> server stub
* **calculate\_xdr.c** —> XDR(External Data Representation) filters
* **calculate.h** —> header file needed for any XDR filters
* **Makefile.calculate** —> compile all the source files by using this file

##### Step 2 – Develop the client program

Here i’m using C language as i said earlier, so, i edited the code in the **calculate\_client.c** file like the way  it accepts the client’s inputs from the keyboard and outputs the remote procedures (add,sub,mul,div) invoking results to the client .

calculate\_client.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110 | #include "calculate.h"      **float** calculate\_prog\_1(**char** \*host,**float** n1,**float** n2,**char** opr,CLIENT \*clnt)  {    **float** \*result\_1;   inputs add\_1\_arg;  **float** \*result\_2;   inputs sub\_1\_arg;  **float** \*result\_3;   inputs mul\_1\_arg;  **float** \*result\_4;   inputs div\_1\_arg;    **if**(opr=='+'){     add\_1\_arg.num1=n1;   add\_1\_arg.num2=n2;   add\_1\_arg.operator=opr;     result\_1 = add\_1(&add\_1\_arg, clnt);  **if** (result\_1 == (**float** \*) NULL) {   clnt\_perror (clnt, "call failed");   }  **return** \*result\_1;  }    **else** **if**(opr=='-'){     sub\_1\_arg.num1=n1;   sub\_1\_arg.num2=n2;   sub\_1\_arg.operator=opr;     result\_2 = sub\_1(&sub\_1\_arg, clnt);  **if** (result\_2 == (**float** \*) NULL) {   clnt\_perror (clnt, "call failed");   }  **return** \*result\_2;  }    **else** **if**(opr=='\*'){     mul\_1\_arg.num1=n1;   mul\_1\_arg.num2=n2;   mul\_1\_arg.operator=opr;     result\_3 = mul\_1(&mul\_1\_arg, clnt);  **if** (result\_3 == (**float** \*) NULL) {   clnt\_perror (clnt, "call failed");   }  **return** \*result\_3;  }    **else** **if**(opr=='/'){     div\_1\_arg.num1=n1;   div\_1\_arg.num2=n2;   div\_1\_arg.operator=opr;    **if**(n2 == 0){  **printf**("Division by zero is not valid.\n");  **exit**(0);   }**else**{     result\_4 = div\_1(&div\_1\_arg, clnt);  **if** (result\_4 == (**float** \*) NULL) {   clnt\_perror (clnt, "call failed");   }  **return** \*result\_4;   }  }  }    **int** main (**int** argc, **char** \*argv[])  {  **char** \*host;  **float** a,b;  **char** op;   CLIENT \*clnt;    **if** (argc < 2) {  **printf** ("usage: %s server\_host\n", argv[0]);  **exit** (1);   }    **printf**("Welcome to Quick Cal!!!\n");  **printf**("+ for Addition\n- for Substraction\n\* for Multiplication\n/ for Division\n");  **printf**("Enter number 1 :\n");  **scanf**("%f",&a);  **printf**("Enter number 2 :\n");  **scanf**("%f",&b);  **printf**("Enter the Operator :\n");  **scanf**("%s",&op);     host = argv[1];     clnt = clnt\_create (host, CALCULATE\_PROG, CALCULATE\_VER, "udp");    **if** (clnt == NULL) {   clnt\_pcreateerror (host);  **exit** (1);   }    **printf**("The Answer = %f\n",calculate\_prog\_1 (host,a,b,op,clnt));     clnt\_destroy (clnt);    **exit** (0);  } |

##### Step 3 – Develop the server program

Similarly you can edit the **calculate\_server.c** file while adding your own codes for the previously defined remote methods(add,sub,mul,div).

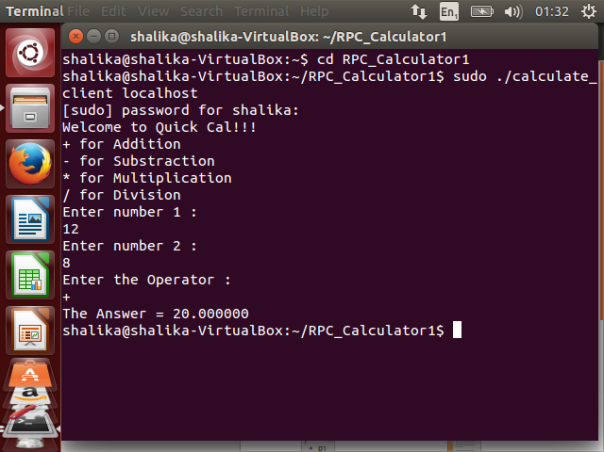
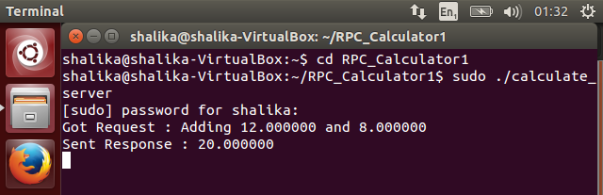
calculate\_server.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | #include "calculate.h"    **float** \* add\_1\_svc(inputs \*argp, **struct** svc\_req \*rqstp)  {  **static** **float** result;     result = argp->num1+argp->num2;  **printf**("Got Request : Adding %f and %f\n",argp->num1,argp->num2);  **printf**("Sent Response : %f\n",result);    **return** &result;  }    **float** \* sub\_1\_svc(inputs \*argp, **struct** svc\_req \*rqstp)  {  **static** **float** result;     result = argp->num1-argp->num2;  **printf**("Got Request : substituting %f from %f\n",argp->num2,argp->num1);  **printf**("Sent Response : %f\n",result);    **return** &result;  }    **float** \* mul\_1\_svc(inputs \*argp, **struct** svc\_req \*rqstp)  {  **static** **float** result;     result = argp->num1\*argp->num2;  **printf**("Got Request : Multiplying %f by %f\n",argp->num1,argp->num2);  **printf**("Sent Response : %f\n",result);    **return** &result;  }    **float** \* div\_1\_svc(inputs \*argp, **struct** svc\_req \*rqstp)  {  **static** **float** result;     result = argp->num1/argp->num2;  **printf**("Got Request : Dividing %f by %f\n",argp->num1,argp->num2);  **printf**("Sent Response : %f\n",result);    **return** &result;  } |

After editing the client and the server code, you need to compile the files. As usual, it’s mandatory to compile the files if u made any changes. Otherwise those changes won’t apply at the execution time. Use this command for compilation of files.

$ make -f Makefile.calculate

Take a new terminal. Then run the executable file of the client(calculate\_client) which was created at the compile time.  
type —-> $ sudo ./calculate\_client localhost  
  
Now the client program is ready to accept keyboard inputs from the client. So, you can enter 2 numbers and an operator which gives the output to the client by invoking the remote procedures at the server side which the client has requested.

**Client Terminal**  
 **Server Terminal**  


* Caller stub, which has to…
  + Pack the argument
  + Transmit argument
  + Receive the result
  + Unpack the result
* Callee stub, which has to…
  + Receive the argument
  + Unpack argument
  + Call the function
  + Pack the result
  + Transmit the result

Our example is a pretty simple one, since we’re just packing and sending a single int as the argument and receiving a single byte as a result. For the caller library, we can pack the data, create a socket, connect to a host (let’s assume localhost for now), send the data, wait to receive, unpack, and then return. Here’s what the header file looks like for the caller library:

// client/is\_prime\_rpc\_client.h

#ifndef IS\_PRIME\_RPC\_CLIENT\_H

#define IS\_PRIME\_RPC\_CLIENT\_H

#include <stdbool.h>

bool is\_prime\_rpc(int number);

#endif

The astute (or just conscious) reader will notice that the interface is actually the exact same as when it was just the library, and this is the point! The caller doesn’t have to worry about anything other than the business logic it’s trying to send (but see caveats below). The implementation, on the other hand, is a little more complex:

// client/is\_prime\_rpc\_client.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#define SERVERPORT "5005" // The port the server will be listening on.

#define SERVER "localhost" // Assume localhost for now

#include "is\_prime\_rpc\_client.h"

// Packs an int. We need to convert it from host order to network order.

int pack(int input) {

return htons(input);

}

// Gets the IPv4 or IPv6 sockaddr.

void \*get\_in\_addr(struct sockaddr \*sa) {

if (sa->sa\_family == AF\_INET) {

return &(((struct sockaddr\_in\*)sa)->sin\_addr);

} else {

return &(((struct sockaddr\_in6\*)sa)->sin6\_addr);

}

}

// Gets a socket to connect with.

int get\_socket() {

int sockfd;

struct addrinfo hints, \*server\_info, \*p;

int number\_of\_bytes;

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM; // We want to use TCP to ensure it gets there

int return\_value = getaddrinfo(SERVER, SERVERPORT, &hints, &server\_info);

if (return\_value != 0) {

fprintf(stderr, "getaddrinfo: %s\n", gai\_strerror(return\_value));

exit(1);

}

// We end up with a linked-list of addresses, and we want to connect to the

// first one we can

for (p = server\_info; p != NULL; p = p->ai\_next) {

// Try to make a socket with this one.

if ((sockfd = socket(p->ai\_family, p->ai\_socktype, p->ai\_protocol)) == -1) {

// Something went wrong getting this socket, so we can try the next one.

perror("client: socket");

continue;

}

// Try to connect to that socket.

if (connect(sockfd, p->ai\_addr, p->ai\_addrlen) == -1) {

// If something went wrong connecting to this socket, we can close it and

// move on to the next one.

close(sockfd);

perror("client: connect");

continue;

}

// If we've made it this far, we have a valid socket and can stop iterating

// through.

break;

}

// If we haven't gotten a valid sockaddr here, that means we can't connect.

if (p == NULL) {

fprintf(stderr, "client: failed to connect\n");

exit(2);

}

// Otherwise, we're good.

return sockfd;

}

// Client side library for the is\_prime RPC.

bool is\_prime\_rpc(int number) {

// First, we need to pack the data, ensuring that it's sent across the

// network in the right format.

int packed\_number = pack(number);

// Now, we can grab a socket we can use to connect see how we can connect

int sockfd = get\_socket();

// Send just the packed number.

if (send(sockfd, &packed\_number, sizeof packed\_number, 0) == -1) {

perror("send");

close(sockfd);

exit(0);

}

// Now, wait to receive the answer.

int buf[1]; // Just receiving a single byte back that represents a boolean.

int bytes\_received = recv(sockfd, &buf, 1, 0);

if (bytes\_received == -1) {

perror("recv");

exit(1);

}

// Since we just have the one byte, we don't really need to do anything while

// unpacking it, since one byte in reverse order is still just a byte.

bool result = buf[0];

// All done! Close the socket and return the result.

close(sockfd);

return result;

}

As mentioned earlier, this client code needs to pack the argument, connect to the server, send the data, receive the data, unpack it, and return it. This is relatively simple for our example, since we just need to ensure the byte order of the number is in the network order.

Next, we need to run the callee library on the server. It will call the is\_prime library we wrote earlier, which now lives entirely on the server.

// server/is\_prime\_rpc\_server.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <sys/wait.h>

#include <signal.h>

#include "is\_prime.h"

#define SERVERPORT "5005" // The port the server will be listening on.

// Gets the IPv4 or IPv6 sockaddr.

void \*get\_in\_addr(struct sockaddr \*sa) {

if (sa->sa\_family == AF\_INET) {

return &(((struct sockaddr\_in\*)sa)->sin\_addr);

} else {

return &(((struct sockaddr\_in6\*)sa)->sin6\_addr);

}

}

// Unpacks an int. We need to convert it from network order to our host order.

int unpack(int packed\_input) {

return ntohs(packed\_input);

}

// Gets a socket to listen with.

int get\_and\_bind\_socket() {

int sockfd;

struct addrinfo hints, \*server\_info, \*p;

int number\_of\_bytes;

memset(&hints, 0, sizeof hints);

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM; // We want to use TCP to ensure it gets there

hints.ai\_flags = AI\_PASSIVE; // Just use the server's IP.

int return\_value = getaddrinfo(NULL, SERVERPORT, &hints, &server\_info);

if (return\_value != 0) {

fprintf(stderr, "getaddrinfo: %s\n", gai\_strerror(return\_value));

exit(1);

}

// We end up with a linked-list of addresses, and we want to connect to the

// first one we can

for (p = server\_info; p != NULL; p = p->ai\_next) {

// Try to make a socket with this one.

if ((sockfd = socket(p->ai\_family, p->ai\_socktype, p->ai\_protocol)) == -1) {

// Something went wrong getting this socket, so we can try the next one.

perror("server: socket");

continue;

}

// We want to be able to reuse this, so we can set the socket option.

int yes = 1;

if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &yes, sizeof(int)) == -1) {

perror("setsockopt");

exit(1);

}

// Try to bind that socket.

if (bind(sockfd, p->ai\_addr, p->ai\_addrlen) == -1) {

// If something went wrong binding this socket, we can close it and

// move on to the next one.

close(sockfd);

perror("server: bind");

continue;

}

// If we've made it this far, we have a valid socket and can stop iterating

// through.

break;

}

// If we haven't gotten a valid sockaddr here, that means we can't connect.

if (p == NULL) {

fprintf(stderr, "server: failed to bind\n");

exit(2);

}

// Otherwise, we're good.

return sockfd;

}

int main(void) {

int sockfd = get\_and\_bind\_socket();

// We want to listen forever on this socket

if (listen(sockfd, /\*backlog=\*/1) == -1) {

perror("listen");

exit(1);

}

printf("Server waiting for connections.\n");

struct sockaddr their\_addr; // Address information of the client

socklen\_t sin\_size;

int new\_fd;

while(1) {

sin\_size = sizeof their\_addr;

new\_fd = accept(sockfd, (struct sockaddr \*)&their\_addr, &sin\_size);

if (new\_fd == -1) {

perror("accept");

continue;

}

// Once we've accepted an incoming request, we can read from it into a buffer.

int buffer;

int bytes\_received = recv(new\_fd, &buffer, sizeof buffer, 0);

if (bytes\_received == -1) {

perror("recv");

continue;

}

// We need to unpack the received data.

int number = unpack(buffer);

printf("Received a request: is %d prime?\n", number);

// Now, we can finally call the is\_prime library!

bool number\_is\_prime = is\_prime(number);

printf("Sending response: %s\n", number\_is\_prime ? "true" : "false");

// Note that we don't have to pack a single byte.

// We can now send it back.

if (send(new\_fd, &number\_is\_prime, sizeof number\_is\_prime, 0) == -1) {

perror("send");

}

close(new\_fd);

}

}

Finally, we can update our main function that runs on the client to use the new RPC library call:

// client/basic\_math\_program\_distributed.c

#include <stdio.h>

#include <stdbool.h>

#include "is\_prime\_rpc\_client.h"

int main(void) {

// Prompt the user to enter a number.

printf("Please enter a number: ");

// Read the user's number. Assume they're entering a valid number.

int input\_number;

scanf("%d", &input\_number);

// Check if it's prime, but now via the RPC library

if (is\_prime\_rpc(input\_number)) {

printf("%d is prime\n", input\_number);

} else {

printf("%d is not prime\n", input\_number);

}

return 0;

}

## Ether Types

1. **IPv4**: 0x0800 (2048 in decimal)
2. **IPv6**: 0x86DD (34525 in decimal)
3. **Address Resolution Protocol (ARP)**: 0x0806 (2054 in decimal)
4. **Reverse Address Resolution Protocol (RARP)**: 0x8035 (32821 in decimal)
5. **AppleTalk**: 0x809B (32923 in decimal)
6. **AppleTalk ARP**: 0x80F3 (33011 in decimal)
7. **DECnet Phase IV**: 0x8137 (33079 in decimal)
8. **Xerox NS IDP**: 0x0600 (1536 in decimal)
9. **Ethernet Loopback Test**: 0x9000 (36864 in decimal)
10. **802.1Q VLAN Tagging**: 0x8100 (33024 in decimal)
11. **PPP (Point-to-Point Protocol)**: 0x880B (34827 in decimal)
12. **MPLS (Multiprotocol Label Switching)**: 0x8847 (34855 in decimal)
13. **PPPoE Discovery Stage**: 0x8863 (34915 in decimal)
14. **PPPoE Session Stage**: 0x8864 (34916 in decimal)
15. **Link Layer Discovery Protocol (LLDP)**: 0x88CC (35020 in decimal)
16. **Cisco Discovery Protocol (CDP)**: 0x2000 (8192 in decimal)
17. **IEEE 802.1X Authentication**: 0x888E (34958 in decimal)
18. **IEEE 802.1Qbg Edge Virtual Bridging (EVB)**: 0x8946 (35142 in decimal)
19. **IPv6 over IPv4 tunneling (6in4)**: 0x86DD (34525 in decimal)
20. **IPv4 over IPv6 tunneling (4in6)**: 0x0800 (2048 in decimal)
21. **Ethernet OAM Protocol (Ethernet in the First Mile)**: 0x8809 (34825 in decimal)
22. **Ethernet CFM (Connectivity Fault Management)**: 0x8902 (35010 in decimal)
23. **EtherCAT Protocol**: 0x88A4 (34980 in decimal)
24. **TRILL (Transparent Interconnection of Lots of Links)**: 0x22F3 (8947 in decimal)
25. **Fibre Channel over Ethernet (FCoE)**: 0x8906 (35014 in decimal)
26. **IEEE 802.11 Wireless LANs**: 0x88B4 (34996 in decimal)
27. **InfiniBand**: 0x8915 (35029 in decimal)
28. **FCoE Initialization Protocol (FIP)**: 0x8914 (35028 in decimal)
29. **HomePlug AV MME (Multimedia over Powerline Alliance Management Message)**: 0x887B (34939 in decimal)
30. **IEEE 802.3 Control**: 0x8808 (34824 in decimal)
31. **Provider Backbone Bridges (PBB)**: 0x88E7 (35047 in decimal)
32. **IEEE 802.1AE MAC Security (MACsec)**: 0x88E5 (35045 in decimal)
33. **IEEE 802.1AX Link Aggregation**: 0x8809 (34825 in decimal)
34. **IEEE 802.11 Management Frames**: 0x888E (34958 in decimal)
35. **IEEE 802.11 Data Frames**: 0x8863 (34915 in decimal)
36. **IEEE 802.11 Control Frames**: 0x8864 (34916 in decimal)
37. **IEEE 802.11 Beacon Frames**: 0x8865 (34917 in decimal)
38. **Ethernet Configuration Testing Protocol (ECP)**: 0x8808 (34824 in decimal)
39. **Ethernet Audio/Video Bridging (AVB) Transport Protocol**: 0x22EA (8938 in decimal)
40. **Ethernet Audio/Video Bridging (AVB) Protocol Type**: 0x22EB (8939 in decimal)
41. **IEEE 802.1Qay Provider Backbone Bridge Traffic Engineering (PBB-TE)**: 0x8902 (35010 in decimal)
42. **IEEE 802.1Qaz Data Center Bridging Exchange (DCBX)**: 0x890D (35021 in decimal)
43. **IEEE 802.1Qbb Priority-based Flow Control (PFC)**: 0x8808 (34824 in decimal)
44. **LLDP (Link Layer Discovery Protocol)**: 0x88CC (35020 in decimal)
45. **IEEE 802.1ag Connectivity Fault Management (CFM)**: 0x8902 (35010 in decimal)
46. **IEEE 802.1ah Provider Backbone Bridges (PBB)**: 0x88E7 (35047 in decimal)
47. **IEEE 802.1Qbg Edge Virtual Bridging (EVB)**: 0x8946 (35142 in decimal)
48. **MAC-over-PDH (Multiplexing Over Pseudo-Wire)**: 0x888F (34959 in decimal)
49. **MAC-over-SDH (Multiplexing Over Pseudo-Wire)**: 0x8890 (34960 in decimal)
50. **MAC-over-SONET/SDH (Multiplexing Over Pseudo-Wire)**: 0x8891 (34961 in decimal)

## Maximum Length Of A Concatenated String With Unique Characters