VISVESVARAYA TECHNOLOGICAL UNIVERSITY

BELAGAVI - 590 018, KARNATAKA

A Report on

"Analysis and Design of Algorithms AAT"

(BCS401)

in

COMPUTER SCIENCE AND ENGINEERING

By

Mr. KAUSHIK K S **USN: 1BY23CS102**

Mrs. Anusha KL
Assistant Professor
Department of CSE, BMSIT&M.

ಬಿ.ಎಂ.ಎಸ್. ತಂತ್ರಿಕ ಮತ್ತು ನಿರ್ವಹಣಾ ಮಹಾವಿದ್ಯಾಲಯ

BMS Institute of Technology and Management

(An Autonomous Institution Affiliated to VTU, Belagavi)

Avalahalli, Doddaballapur Main Road, Bengaluru – 560119

2024-2025

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

BELAGAVI – 590 018, KARNATAKA

ಬಿ.ಎಂ.ಎಸ್. ತಂತ್ರಿಕ ಮತ್ತು ನಿರ್ವಹಣಾ ಅಧ್ಯಯನ ಸಂಸ್ಥೆ

BMS Institute of Technology and Management

(An Autonomous Institution Affiliated to VTU, Belagavi)

Avalahalli, Doddaballapur Main Road, Bengaluru – 560119

CERTIFICATE OF COMPLETION

This is to Certify That

Mr. KAUSHIK K S (1BY23CS102)

Has Successfully Completed the Alternate Assessment Activity

Conducted as part of the course

"Analysis and Design of Algorithms (BCS401)"

This assessment was designed to provide a flexible approach to evaluate the students' understanding, and application of the concepts covered in the course. This certificate acknowledges the students' achievement and ability to apply theoretical knowledge in a practical or alternative setting, showcasing skills such as problem-solving, analytical thinking, and coding.

The report has been approved as it satisfies the academic requirements in respect of AAT work for the B.E Degree.

Marks Distribution

Programs (MAX 10 Marks)	
Report (MAX 05 Marks)	
Viva (MAX 05 Marks)	
Total (MAX 20 Marks)	

Signature of the Course Coordinator

PROGRAMS

ARRAYS/STRINGS

1.Find the Index of the First Occurrence in a String

Question:

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:

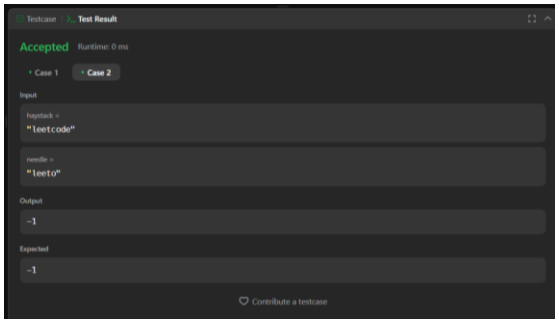
Input: haystack = "sadbutsad", needle = "sad"

Output: 0

Solution in C:

```
int strStr(char * haystack, char * needle){
    if (!*needle) return 0;
    for (int i = 0; haystack[i]; i++) {
        int j = 0;
        while (needle[j] && haystack[i + j] == needle[j]) j++;
        if (!needle[j]) return i;
    }
    return -1;
}
```

OUTPUT :



2.**Valid Anagram**

Question:

Given two strings s and t, return true if t is an anagram of s, and false otherwise.

Example 1:

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

Output: true

Solution in C:

```
#include <stdbool.h>

bool isAnagram(char * s, char * t){
    int count[26] = {0};
    for (int i = 0; s[i]; i++) count[s[i] - 'a']++;
    for (int i = 0; t[i]; i++) count[t[i] - 'a']--;
    for (int i = 0; i < 26; i++) {
        if (count[i] != 0) return false;
    }
    return true;
}
```

OUTPUT:

3. Sort an Array

Question:

Given an array of integers, sort the array in ascending order.

Example 1:

Input: nums = [5,2,3,1]

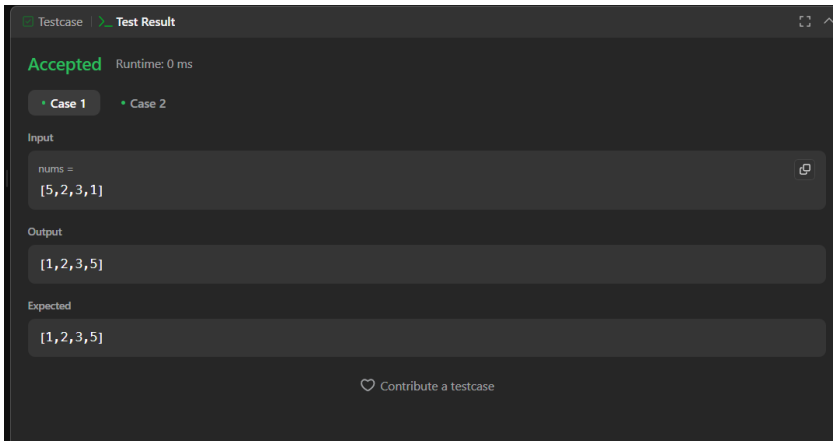
Output: [1,2,3,5]

Solution in C:

```
void swap(int* a, int* b) {
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
int partition(int* nums, int low, int high) {
   int pivot = nums[high];
   int i = low - 1;
   for (int j = low; j < high; j++) {
      if (nums[j] < pivot) {
         swap(&nums[++i], &nums[j]);
      }
   }
   swap(&nums[i + 1], &nums[high]);
   return i + 1;
}
void quickSort(int* nums, int low, int high) {
   if (low < high) {
      int pi = partition(nums, low, high);
```

```
        quickSort(nums, low, pi - 1);
        quickSort(nums, pi + 1, high);
    }
}
int* sortArray(int* nums, int numsSize, int* returnSize) {
    *returnSize = numsSize;
    quickSort(nums, 0, numsSize - 1);
    return nums;
}
```

OUTPUT :



4. Merge Two Sorted Lists

Question:

Merge two sorted linked lists and return it as a new sorted list. The new list should be made by splicing together the nodes of the first two lists.

Example 1:

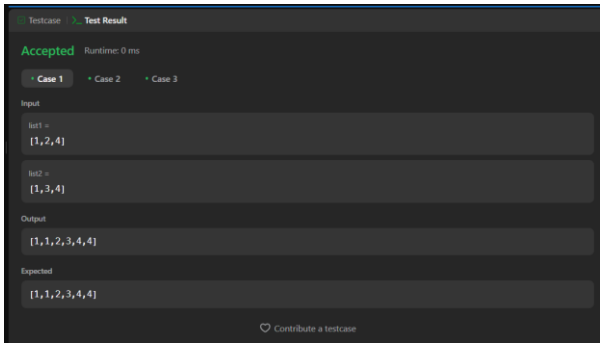
Input: l1 = [1,2,4], l2 = [1,3,4]

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

Solution in C:

```
struct ListNode* mergeTwoLists(struct ListNode* l1, struct ListNode* l2) {
    struct ListNode dummy;
    struct ListNode* tail = &dummy;
    dummy.next = NULL;
    while (l1 && l2) {
        if (l1->val < l2->val) {
            tail->next = l1;
            l1 = l1->next;
        }
        else {
            tail->next = l2;
            l2 = l2->next;
        }
        tail = tail->next;
    }
    tail->next = l1 ? l1 : l2;
    return dummy.next;
}
```

OUTPUT :



5. Word Search

Question:

Given a 2D board and a word, find if the word exists in the grid. The word can be constructed from letters of sequentially adjacent cells, horizontally or vertically.

Example:

Input: board = [

["A","B","C","E"],

["S","F","C","S"],

["A","D","E","E"]

], word = "ABCCED"

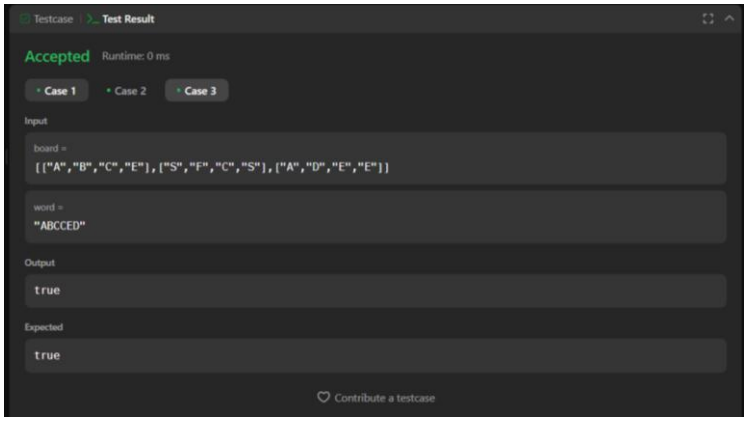
Output: true

Solution:

```
bool dfs(char** board, int boardSize, int* boardColSize, int i, int j, char* word, int idx) {
   if (idx == strlen(word)) return true;
   if (i < 0 || i >= boardSize || j < 0 || j >= boardColSize[0] || board[i][j] != word[idx]) return
false;
   char tmp = board[i][j];
   board[i][j] = '#';
   bool res = dfs(board, boardSize, boardColSize, i+1, j, word, idx+1) ||
           dfs(board, boardSize, boardColSize, i-1, j, word, idx+1) ||
           dfs(board, boardSize, boardColSize, i, j+1, word, idx+1) ||
           dfs(board, boardSize, boardColSize, i, j-1, word, idx+1);
   board[i][j] = tmp;
   return res;
}


bool exist(char** board, int boardSize, int* boardColSize, char* word){
```

```
    for (int i = 0; i < boardSize; i++) {
        for (int j = 0; j < boardColSize[0]; j++) {
            if (dfs(board, boardSize, boardColSize, i, j, word, 0)) return true;
        }
    }
    return false;
}
```

OUTPUT :



BINARY SEARCH

1.Binary Search

Question:

Given a sorted (in ascending order) integer array nums of n elements and a target value, write a function to search target in nums. If target exists, then return its index, otherwise return -1.

Example 1:

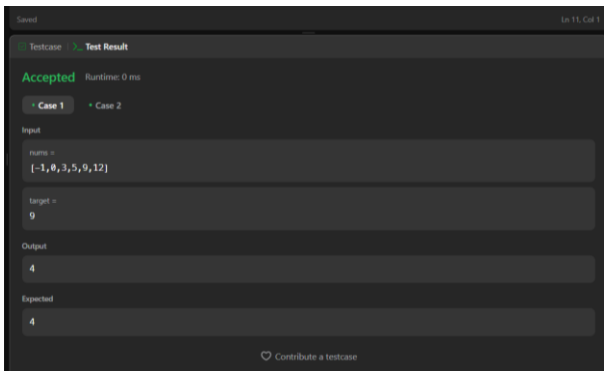
Input: nums = [-1,0,3,5,9,12], target = 9

Output: 4

Solution in C:

```
int search(int* nums, int numsSize, int target) {
   int left = 0, right = numsSize - 1;
   while (left <= right) {
      int mid = left + (right - left) / 2;
      if (nums[mid] == target) return mid;
      else if (nums[mid] < target) left = mid + 1;
      else right = mid - 1;
   }
   return -1;
}
```

OUTPUT :



2. Search Insert Position

Question:

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

Example 1:

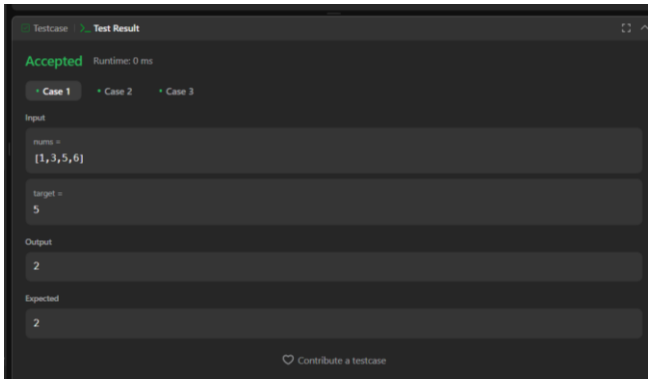
Input: nums = [1,3,5,6], target = 5

Output: 2

Solution in C:

```
int searchInsert(int* nums, int numsSize, int target) {
   int left = 0, right = numsSize - 1;
   while (left <= right) {
      int mid = left + (right - left) / 2;
      if (nums[mid] == target) return mid;
      else if (nums[mid] < target) left = mid + 1;
      else right = mid - 1;
   }
   return left;
}
```

OUTPUT :



3. Kth Largest Element in an Array

Question:

Find the kth largest element in an unsorted array. Note that it is the kth largest element in sorted order, not the kth distinct element.

Example 1:

Input: nums = [3,2,1,5,6,4], k = 2

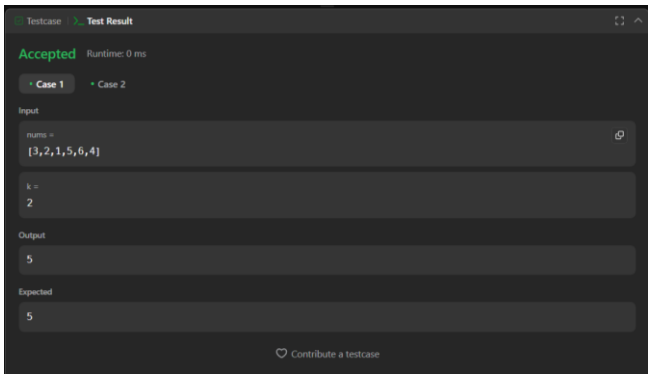
Output: 5

Solution in C:

```
void swap(int* a, int* b) {
   int tmp = *a;
   *a = *b;
   *b = tmp;
}

int partition(int* nums, int low, int high) {
   int pivot = nums[high];
   int i = low;
   for (int j = low; j < high; j++) {
      if (nums[j] > pivot) {
         swap(&nums[i], &nums[j]);
         i++;
      }
   }
   swap(&nums[i], &nums[high]);
```

```
    return i;
}

int quickSelect(int* nums, int low, int high, int k) {
    if (low == high) return nums[low];
    int pi = partition(nums, low, high);
    if (pi == k) return nums[pi];
    else if (pi < k) return quickSelect(nums, pi + 1, high, k);
    else return quickSelect(nums, low, pi - 1, k);
}

int findKthLargest(int* nums, int numsSize, int k){
    return quickSelect(nums, 0, numsSize - 1, k - 1);
}
```

OUTPUT :



Sliding Window

1. Minimum Size Subarray Sum

Question:

Given an array of positive integers nums and a positive integer target, return the minimal length of a contiguous subarray of which the sum is greater than or equal to target. If there is no such subarray, return 0.

Example 1:

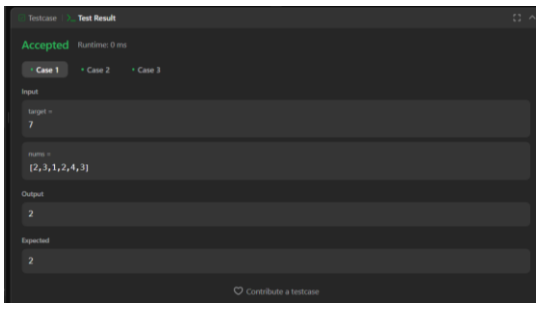
Input: target = 7, nums = [2,3,1,2,4,3]

Output: 2

Solution in C:

```
int minSubArrayLen(int target, int* nums, int numsSize){
   int left = 0, sum = 0, minLen = numsSize + 1;
   for (int right = 0; right < numsSize; right++) {
      sum += nums[right];
      while (sum >= target) {
         if (right - left + 1 < minLen) minLen = right - left + 1;
         sum -= nums[left++];
      }
   }
   return (minLen == numsSize + 1) ? 0 : minLen;
}
```

OUTPUT :



2. Longest Substring Without Repeating Characters

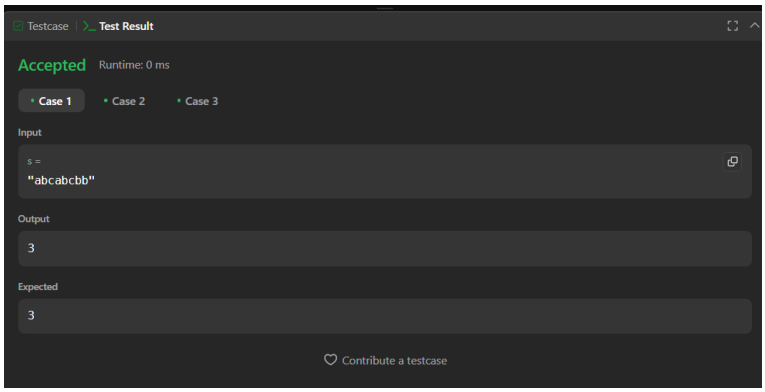
Question:
Given a string s, find the length of the longest substring without repeating characters.

Example 1:
Input: s = "abcabcbb"
Output: 3

Solution in C:

```
int lengthOfLongestSubstring(char * s){
    int map[128] = {0};
    int left = 0, right = 0, maxLen = 0;
    while (s[right]) {
        char r = s[right];
        map[r]++;
        while (map[r] > 1) {
            char l = s[left];
            map[l]--;
            left++;
        }
        if (right - left + 1 > maxLen)
            maxLen = right - left + 1;
        right++;
    }
    return maxLen;
}
```

OUTPUT :



3. Permutation in String

Question:
Given two strings s1 and s2, return true if s2 contains a permutation of s1, or false otherwise.
In other words, return true if one of s1's permutations is the substring of s2.

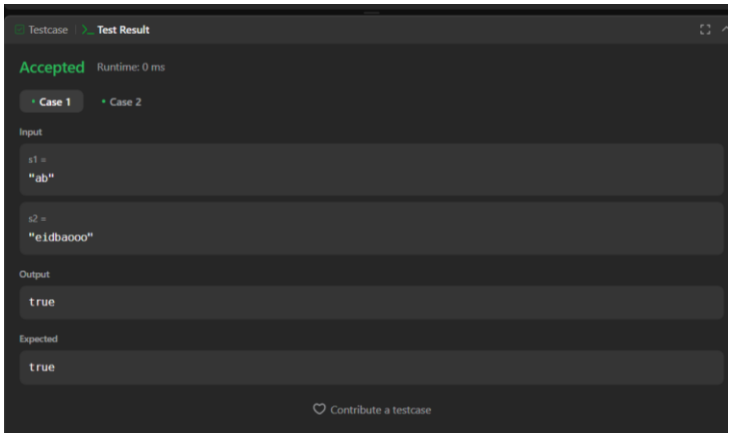
Example 1:
Input: s1 = "ab", s2 = "eidbaooo"
Output: true

Solution in C:

```
bool checkInclusion(char * s1, char * s2){

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

  if (len1 > len2) return false;


  int count1[26] = {0}, count2[26] = {0};

  for (int i = 0; i < len1; i++) {

    count1[s1[i] - 'a']++;

    count2[s2[i] - 'a']++;

  }


  for (int i = 0; i <= len2 - len1; i++) {

    if (memcmp(count1, count2, sizeof(count1)) == 0)
```

```
            return true;
        if (i + len1 < len2) {
            count2[s2[i] - 'a']--;
            count2[s2[i + len1] - 'a']++;
        }
    }
    return false;
}
```

OUTPUT :



Dynamic Programming (1D / Basic)

1. Climbing Stairs

Question:

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:

Input: 3

Output: 3

Explanation:

There are three ways to climb to the top:

1 step + 1 step + 1 step

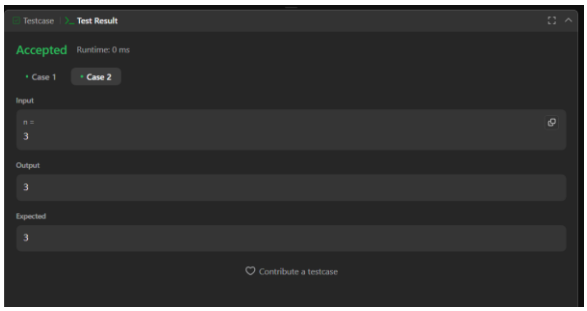
1 step + 2 steps

2 steps + 1 step

Solution:

```
int climbStairs(int n){
   if (n <= 2) return n;
   int first = 1, second = 2, third;
   for (int i = 3; i <= n; i++) {
      third = first + second;
      first = second;
      second = third;
   }
   return second;
}
```

OUTPUT:



2. Triangle

Question:

Given a triangle array, return the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

Example:

Input: [[2],[3,4],[6,5,7],[4,1,8,3]]

Output: 11

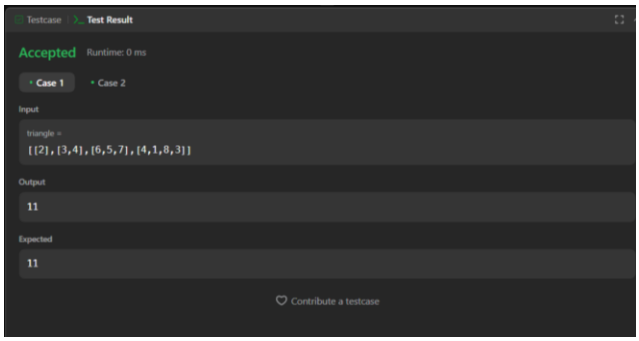
Explanation:

The minimum path is 2 + 3 + 5 + 1 = 11

Solution:

```
int minimumTotal(int** triangle, int triangleSize, int* triangleColSize){
   int dp[triangleSize + 1];
   for (int i = 0; i <= triangleSize; i++) dp[i] = 0;
   for (int i = triangleSize - 1; i >= 0; i--) {
      for (int j = 0; j <= i; j++) {
         dp[j] = (dp[j] < dp[j + 1] ? dp[j] : dp[j + 1]) + triangle[i][j];
      }
   }
   return dp[0];
}
```

OUTPUT :



3. 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

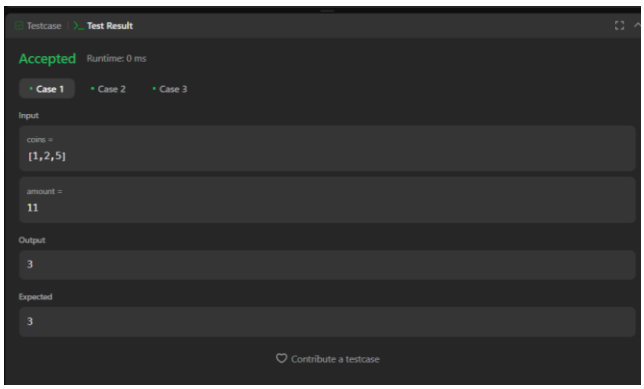
Explanation: 11 = 5 + 5 + 1

Solution :

```
int coinChange(int* coins, int coinsSize, int amount){
    int dp[amount + 1];
    for (int i = 0; i <= amount; i++)
        dp[i] = amount + 1; // set to "infinity"
    dp[0] = 0;

    for (int i = 1; i <= amount; i++) {
        for (int j = 0; j < coinsSize; j++) {
            if (coins[j] <= i) {
```

```
            if (dp[i - coins[j]] + 1 < dp[i])
                dp[i] = dp[i - coins[j]] + 1;
        }
      }
    }
    return (dp[amount] > amount) ? -1 : dp[amount];
}
```

OUTPUT :



Matrix / Multi-Dimensional DP

1. **Dungeon Game**

Question:

The demons had captured the princess and imprisoned her in the bottom-right corner of a dungeon. The dungeon is represented by a 2D grid with integers. Each cell represents health points lost (negative) or gained (positive). You start at the top-left corner and want to reach the princess with a minimum initial health so that you never drop to zero or below.

Example:

Input: dungeon = [[-2,-3,3],[-5,-10,1],[10,30,-5]]

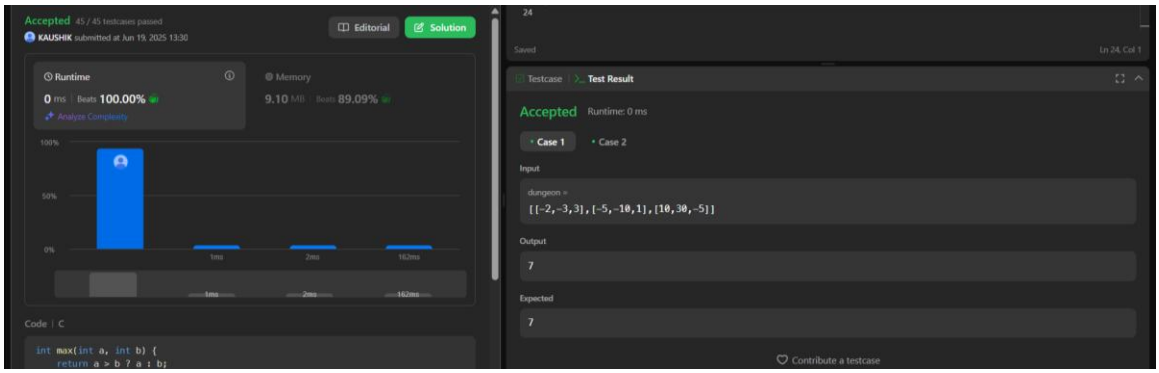
Output: 7

Solution:

```
int max(int a, int b) {
   return a > b ? a : b;
}
int calculateMinimumHP(int** dungeon, int dungeonSize, int* dungeonColSize){
   int m = dungeonSize, n = dungeonColSize[0];
   int dp[m+1][n+1];

   for (int i = 0; i <= m; i++)
      for (int j = 0; j <= n; j++)
         dp[i][j] = 1000000;

   dp[m][n-1] = 1;
   dp[m-1][n] = 1;

   for (int i = m - 1; i >= 0; i--) {
      for (int j = n - 1; j >= 0; j--) {
```

```
            int need = (dp[i+1][j] < dp[i][j+1]) ? dp[i+1][j] : dp[i][j+1];
            dp[i][j] = max(1, need - dungeon[i][j]);
        }
    }
    return dp[0][0];
}
```

OUTPUT :



2. Sudoku Solver

Question:

Write a program to solve a Sudoku puzzle by filling the empty cells. Empty cells are denoted by '.'.

Example:

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: The solved board.

Solution:

```
#include <stdbool.h>
#include <string.h>
bool isValid(char** board, int row, int col, char c) {
  for (int i = 0; i < 9; i++) {
    if (board[row][i] == c) return false;
    if (board[i][col] == c) return false;
    if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c) return false;
  }
  return true;
}

bool solve(char** board) {
  for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
      if (board[i][j] == '.') {
        for (char c = '1'; c <= '9'; c++) {
          if (isValid(board, i, j, c)) {
            board[i][j] = c;
            if (solve(board)) return true;
            board[i][j] = '.';
          }
```

```
            }
            return false;
          }
        }
      }
    return true;
}

void solveSudoku(char** board, int boardSize, int* boardColSize) {
    solve(board);
}
```

OUTPUT:

3. Longest Increasing Path in a Matrix

Question:
Given an m x n integers matrix, return the length of the longest increasing path in the matrix.
From each cell, you can either move in four directions: left, right, up, or down. You may **not** move diagonally or move outside the boundary (i.e., wrap-around is not allowed).

Example 1:
Input:

csharp

CopyEdit

matrix = [

[9,9,4],

[6,6,8],

[2,1,1]

]

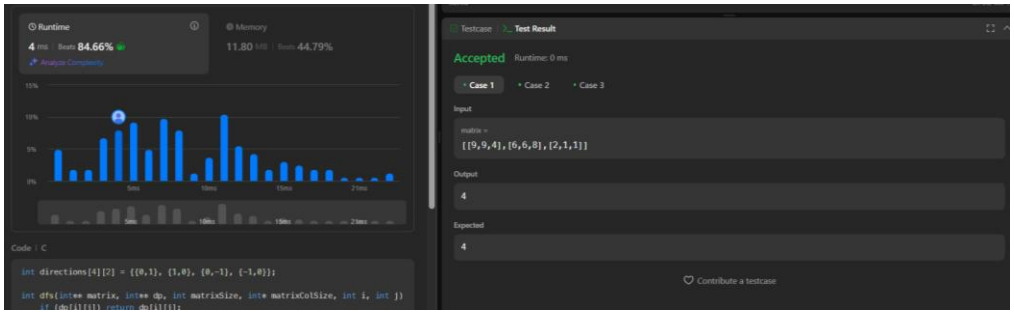
Output: 4
Explanation: The longest increasing path is [1, 2, 6, 9].

Solution :

```
int directions[4][2] = {{0,1}, {1,0}, {0,-1}, {-1,0}};

int dfs(int** matrix, int** dp, int matrixSize, int* matrixColSize, int i, int j) {

   if (dp[i][j]) return dp[i][j];


   int maxLen = 1;

   for (int d = 0; d < 4; d++) {

      int x = i + directions[d][0];

      int y = j + directions[d][1];

      if (x >= 0 && y >= 0 && x < matrixSize && y < matrixColSize[0] && matrix[x][y] >
matrix[i][j]) {

         int len = 1 + dfs(matrix, dp, matrixSize, matrixColSize, x, y);

         if (len > maxLen) maxLen = len;

      }

   }

   return dp[i][j] = maxLen;

}
```

```
int longestIncreasingPath(int** matrix, int matrixSize, int* matrixColSize){
    if (matrixSize == 0 || matrixColSize[0] == 0) return 0;

    int** dp = malloc(matrixSize * sizeof(int*));
    for (int i = 0; i < matrixSize; i++) {
        dp[i] = calloc(matrixColSize[0], sizeof(int));
    }

    int maxPath = 0;
    for (int i = 0; i < matrixSize; i++) {
        for (int j = 0; j < matrixColSize[0]; j++) {
            int len = dfs(matrix, dp, matrixSize, matrixColSize, i, j);
            if (len > maxPath) maxPath = len;
        }
    }

    for (int i = 0; i < matrixSize; i++) free(dp[i]);
    free(dp);
    return maxPath;
}
```

OUTPUT :



4. Unique Paths

Question:
There is a robot located at the top-left corner of an m x n grid. The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid.
How many possible unique paths are there?

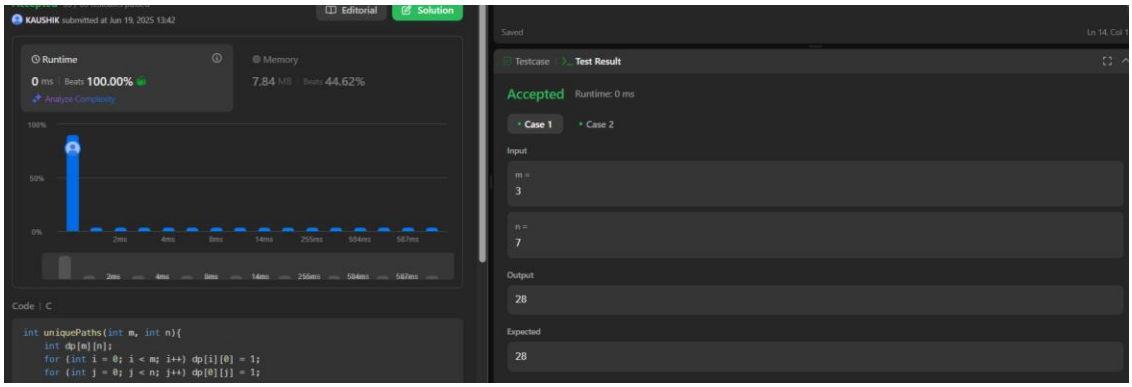
Example 1:
Input: m = 3, n = 7
Output: 28

Example 2:
Input: m = 3, n = 2
Output: 3
Explanation: From top-left to bottom-right, the robot can go right->down->down, or down->down->right, or down->right->down.

Solution :

```
int uniquePaths(int m, int n){

   int dp[m][n];

   for (int i = 0; i < m; i++) dp[i][0] = 1;

   for (int j = 0; j < n; j++) dp[0][j] = 1;


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

      for (int j = 1; j < n; j++) {

         dp[i][j] = dp[i-1][j] + dp[i][j-1];

      }

   }


   return dp[m-1][n-1];

}
```

OUTPUT :



Graph / Topological Sort

1. **Network Delay Time**

Question:

There are n network nodes, labeled 1 to n. Given times array where times[i] = (u, v, w) represents a directed edge from u to v with travel time w, return the time it takes for all nodes to receive a signal sent from node k. If impossible, return -1.

Example:

Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

Output: 2

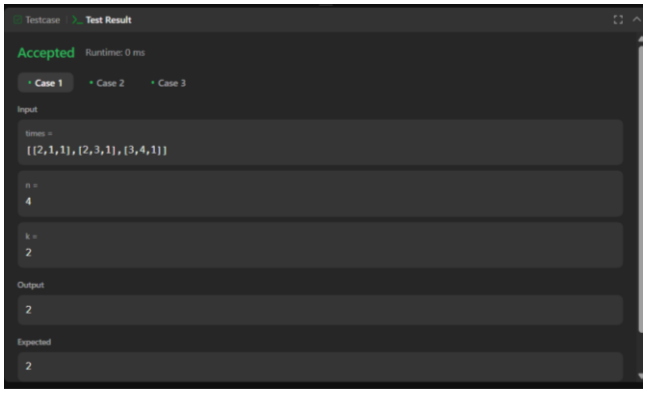
Solution:

```
#include <limits.h>

#include <stdbool.h>

int networkDelayTime(int** times, int timesSize, int* timesColSize, int n, int k){

   int dist[n+1];

   bool visited[n+1];

   for (int i = 1; i <= n; i++) {
```

```
    dist[i] = INT_MAX;
    visited[i] = false;
}
dist[k] = 0;

for (int i = 1; i <= n; i++) {
    int u = -1, minDist = INT_MAX;
    for (int j = 1; j <= n; j++) {
        if (!visited[j] && dist[j] < minDist) {
            minDist = dist[j];
            u = j;
        }
    }
    if (u == -1) break;
    visited[u] = true;
    for (int t = 0; t < timesSize; t++) {
        int src = times[t][0], dst = times[t][1], w = times[t][2];
        if (src == u && dist[u] != INT_MAX && dist[u] + w < dist[dst]) {
            dist[dst] = dist[u] + w;
        }
    }
}

int ans = 0;
for (int i = 1; i <= n; i++) {
    if (dist[i] == INT_MAX) return -1;
```

```
    if (dist[i] > ans) ans = dist[i];
  }
  return ans;
}
```

OUTPUT :



2. Course Schedule II

Question:
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.
Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

Example 1:
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3] or [0,1,2,3]

Example 2:
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]

Solution :

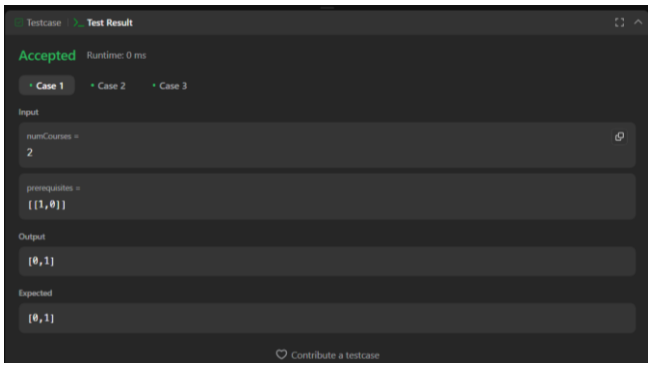
```
int* findOrder(int numCourses, int** prerequisites, int prerequisitesSize, int*
prerequisitesColSize, int* returnSize){
  int* indegree = calloc(numCourses, sizeof(int));
  int** graph = malloc(numCourses * sizeof(int*));
```

```
int* graphColSize = calloc(numCourses, sizeof(int));

for (int i = 0; i < numCourses; i++) {
    graph[i] = malloc(numCourses * sizeof(int)); // Max possible edges
}

for (int i = 0; i < prerequisitesSize; i++) {
    int to = prerequisites[i][0];
    int from = prerequisites[i][1];
    graph[from][graphColSize[from]++] = to;
    indegree[to]++;
}

int* queue = malloc(numCourses * sizeof(int));
int front = 0, rear = 0;

for (int i = 0; i < numCourses; i++) {
    if (indegree[i] == 0) queue[rear++] = i;
}

int* result = malloc(numCourses * sizeof(int));
int idx = 0;

while (front < rear) {
    int course = queue[front++];
    result[idx++] = course;
```

```
    for (int i = 0; i < graphColSize[course]; i++) {
        int next = graph[course][i];
        indegree[next]--;
        if (indegree[next] == 0) {
            queue[rear++] = next;
        }
    }
}

if (idx != numCourses) {
    *returnSize = 0;
    free(result);
    result = malloc(sizeof(int)); // Empty array
} else {
    *returnSize = idx;
}

// Free memory
for (int i = 0; i < numCourses; i++) {
    free(graph[i]);
}
free(graph);
free(graphColSize);
free(indegree);
free(queue);
```

```
    return result;
}
```

OUTPUT :



Backtracking

1. Find Minimum Time to Finish All Jobs

Question:

You are given an array jobs where jobs[i] is the time it takes to complete the ith job. There are k workers available, and jobs must be assigned such that the maximum working time among all workers is minimized.

Example:

Input: jobs = [3,2,3], k = 3

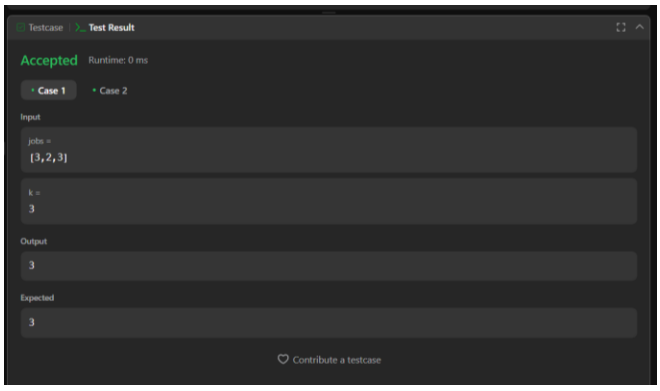
Output: 3

Solution:

```
#include <stdbool.h>

#include <stdlib.h>


bool backtrack(int* jobs, int jobsSize, int* workers, int k, int limit, int idx){

   if (idx == jobsSize) return true;

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

      if (workers[i] + jobs[idx] <= limit) {

         workers[i] += jobs[idx];

         if (backtrack(jobs, jobsSize, workers, k, limit, idx+1)) return true;

         workers[i] -= jobs[idx];

      }

      if (workers[i] == 0) break;

   }

   return false;

}


int minimumTimeRequired(int* jobs, int jobsSize, int k){
```

```
    int left = 0, right = 0;
    for (int i = 0; i < jobsSize; i++) {
        if (jobs[i] > left) left = jobs[i];
        right += jobs[i];
    }
    int workers[k];
    while (left < right) {

        int mid = (left + right) / 2;
        for (int i = 0; i < k; i++) workers[i] = 0;
        if (backtrack(jobs, jobsSize, workers, k, mid, 0)) right = mid;
        else left = mid + 1;
    }
    return left;
}
```

OUTPUT :



2.Combination Sum

Problem Statement:

Given an array of distinct integers candidates and a target integer target,

return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order.

The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

Sample Input:

candidates = [2,3,6,7], target = 7

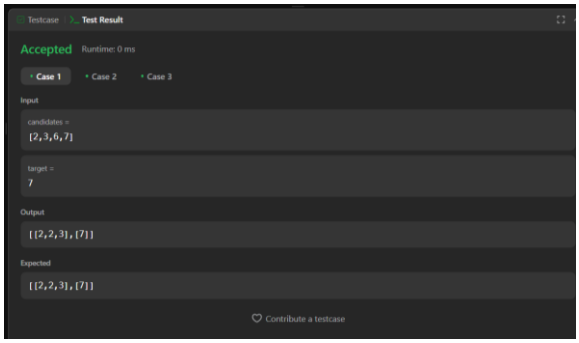
Sample Output:

[[2,2,3],[7]]

```
#include <stdio.h>
#include <stdlib.h>

void dfs(int* candidates, int candidatesSize, int target, int* temp, int tempSize, int** result, int* returnSize, int* returnColumnSizes, int start) {
  if (target == 0) {
    result[*returnSize] = (int*)malloc(tempSize * sizeof(int));
    for (int i = 0; i < tempSize; i++) {
      result[*returnSize][i] = temp[i];
    }
    returnColumnSizes[*returnSize] = tempSize;
    (*returnSize)++;
    return;
  }
```

```
    for (int i = start; i < candidatesSize; i++) {
        if (candidates[i] <= target) {
            temp[tempSize] = candidates[i];
            dfs(candidates, candidatesSize, target - candidates[i], temp, tempSize + 1, result,
returnSize, returnColumnSizes, i);
        }
    }
}

int** combinationSum(int* candidates, int candidatesSize, int target, int* returnSize, int**
returnColumnSizes) {
    int** result = (int**)malloc(1000 * sizeof(int*)); // assuming max 1000 combinations
    *returnColumnSizes = (int*)malloc(1000 * sizeof(int));
    int* temp = (int*)malloc(1000 * sizeof(int));
    *returnSize = 0;

    dfs(candidates, candidatesSize, target, temp, 0, result, returnSize, *returnColumnSizes, 0);

    free(temp);
    return result;
}
```

OUTPUT :



3. Subsets

Problem Statement:
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.

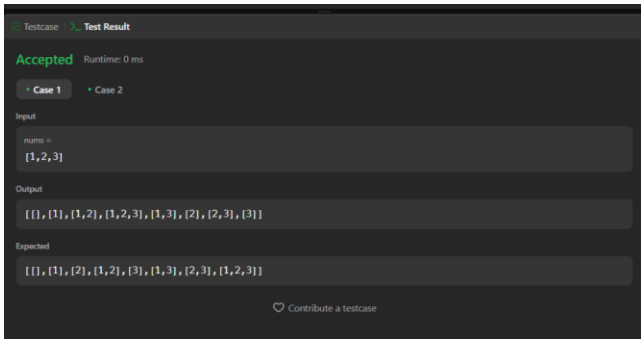
Sample Input:
nums = [1,2,3]

Sample Output:
[[3],[1],[2],[1,2,3],[1,3],[2,3],[1,2],[]]

```
#include <stdio.h>
#include <stdlib.h>

void dfsSubsets(int* nums, int numsSize, int** result, int* returnSize, int* temp, int
tempSize, int** returnColumnSizes, int start) {
  result[*returnSize] = (int*)malloc(tempSize * sizeof(int));
  for (int i = 0; i < tempSize; i++) {
    result[*returnSize][i] = temp[i];
  }
  (*returnColumnSizes)[*returnSize] = tempSize;
  (*returnSize)++;

  for (int i = start; i < numsSize; i++) {
    temp[tempSize] = nums[i];
    dfsSubsets(nums, numsSize, result, returnSize, temp, tempSize + 1,
returnColumnSizes, i + 1);
  }
}

int** subsets(int* nums, int numsSize, int* returnSize, int** returnColumnSizes) {
  int** result = (int**)malloc(1000 * sizeof(int*));
  *returnColumnSizes = (int*)malloc(1000 * sizeof(int));
  int* temp = (int*)malloc(1000 * sizeof(int));
  *returnSize = 0;

  dfsSubsets(nums, numsSize, result, returnSize, temp, 0, returnColumnSizes, 0);

  free(temp);
  return result;
```

}

OUTPUT :


Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Input
nums =
[1,2,3]
Output
[[],[1],[1,2],[1,2,3],[1,3],[2],[2,3],[3]]
Expected
[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
Contribute a testcase

Hash Map / Frequency Counting

1. Partition Equal Subset Sum

Question:

Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Example:

Input: [1,5,11,5]

Output: true

Explanation: The array can be partitioned as [1,5,5] and [11]

Solution:

```
#include <stdbool.h>

#include <string.h>

bool canPartition(int* nums, int numsSize){

  int sum = 0;

  for (int i = 0; i < numsSize; i++) sum += nums[i];

  if (sum % 2 != 0) return false;

  int target = sum / 2;


  bool dp[target + 1];

  memset(dp, false, sizeof(dp));

  dp[0] = true;


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

    for (int j = target; j >= nums[i]; j--) {

      dp[j] = dp[j] || dp[j - nums[i]];

    }
```

```
    }
    return dp[target];
}
```

OUTPUT :

2. Top K Frequent Elements

Problem Statement:

Given an integer array nums and an integer k, return the k most frequent elements.

You may return the answer in any order.

Sample Input:

nums = [1,1,1,2,2,3], k = 2

Sample Output:

[1,2]

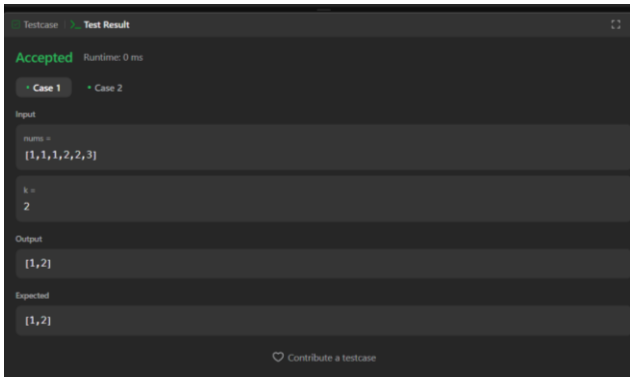
Solution :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Pair {
    int num;
    int freq;
};
```

```
int cmp(const void* a, const void* b) {
   return ((struct Pair*)b)->freq - ((struct Pair*)a)->freq;
}
int* topKFrequent(int* nums, int numsSize, int k, int* returnSize) {
   struct Pair* map = (struct Pair*)malloc(numsSize * sizeof(struct Pair));
   int mapSize = 0;

   for (int i = 0; i < numsSize; i++) {
      int found = 0;
      for (int j = 0; j < mapSize; j++) {
         if (map[j].num == nums[i]) {
            map[j].freq++;
            found = 1;
            break;
         }
      }
      if (!found) {
         map[mapSize].num = nums[i];
         map[mapSize].freq = 1;
         mapSize++;
      }
   }

   qsort(map, mapSize, sizeof(struct Pair), cmp);
```

```
    int* result = (int*)malloc(k * sizeof(int));
    for (int i = 0; i < k; i++) {
        result[i] = map[i].num;
    }
    *returnSize = k;
    free(map);
    return result;
}
```

OUTPUT :



Greedy + Heap

1. **Task Scheduler**

Problem Statement:

Given a characters array tasks, representing the tasks a CPU needs to do, where each letter represents a different task. Tasks could be done in any order. Each task is done in one unit of time. For each unit of time, the CPU can either work on a task or be idle.

However, there is a non-negative integer n that represents the cooldown period between two same tasks (the same letter), that is that there must be at least n units of time between any two same tasks.

Return the least number of units of times that the CPU will take to finish all the given tasks.

Sample Input:

tasks = ['A','A','A','B','B','B'], n = 2

Sample Output:

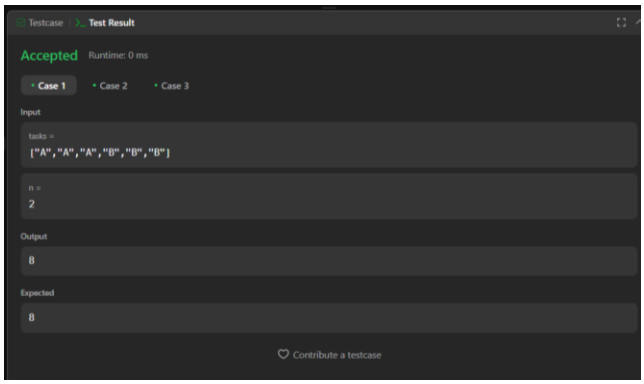
8

Solution :

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int leastInterval(char* tasks, int tasksSize, int n) {

  int freq[26] = {0};

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

    freq[tasks[i] - 'A']++;

  }


  int maxFreq = 0, maxCount = 0;

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

    if (freq[i] > maxFreq) {
```

```
            maxFreq = freq[i];
            maxCount = 1;
        } else if (freq[i] == maxFreq) {
            maxCount++;
        }
    }

    int partCount = maxFreq - 1;
    int partLength = n - (maxCount - 1);
    int emptySlots = partCount * partLength;
    int remainingTasks = tasksSize - maxFreq * maxCount;
    int idles = emptySlots > remainingTasks ? emptySlots - remainingTasks : 0;

    return tasksSize + idles;
}
```

OUTPUT :



2. Last Stone Weight

Problem Statement:

You are given an array of integers stones where stones[i] is the weight of the ith stone.

We are playing a game with the stones. On each turn, we choose the heaviest two stones and smash them together. Suppose the heaviest two stones have weights x and y with x <= y. The result of this smash is:

- If x == y, both stones are destroyed.

- If x != y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x.

At the end, there is at most one stone left.

Return the weight of the last remaining stone. If there are no stones left, return 0.

Sample Input:

stones = [2,7,4,1,8,1]

Sample Output:

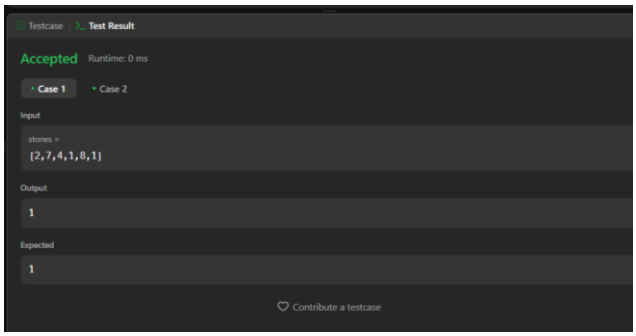
1

Solution :

```
int cmpMaxHeap(const void* a, const void* b) {
   return *(int*)b - *(int*)a;
}


int lastStoneWeight(int* stones, int stonesSize) {
   while (1) {
      qsort(stones, stonesSize, sizeof(int), cmpMaxHeap);


      if (stonesSize == 1) return stones[0];


      if (stones[0] == stones[1]) {
         for (int i = 2; i < stonesSize; i++) stones[i - 2] = stones[i];
         stonesSize -= 2;
      } else {
         stones[0] -= stones[1];
```

```
            for (int i = 2; i < stonesSize; i++) stones[i - 1] = stones[i];
            stonesSize--;
        }
    }
    return 0;
}
```

OUTPUT :



3. **IPO**

Problem Statement:

Suppose LeetCode will start its IPO soon. In order to sell a good price of its shares to Venture Capital, LeetCode would like to work on some projects to increase its capital before the IPO.

Given the initial capital w, the number of projects k, profits[i] and capital[i] for each project, return the maximized capital you can get after finishing at most k distinct projects.

Sample Input:

k = 2, w = 0, profits = [1,2,3], capital = [0,1,1]

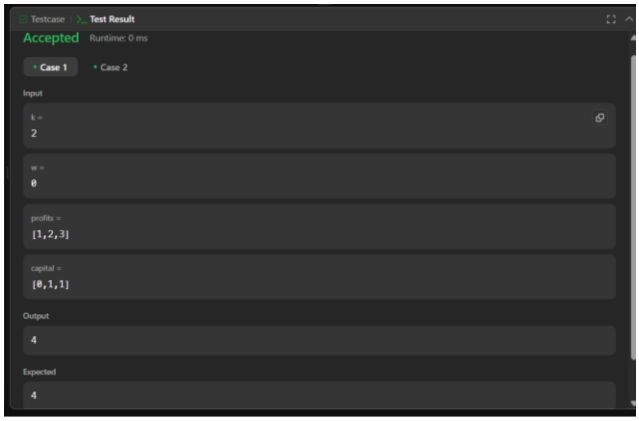
Sample Output:

4

Solution :

```
int findMaximizedCapital(int k, int w, int* profits, int profitsSize, int* capital, int capitalSize)
{
    int size = profitsSize < capitalSize ? profitsSize : capitalSize;
    for (int i = 0; i < k; i++) {
        int maxProfit = -1, idx = -1;
        for (int j = 0; j < size; j++) {
            if (capital[j] <= w && profits[j] > maxProfit) {
                maxProfit = profits[j];
                idx = j;
            }
        }
        if (idx == -1) break;
        w += profits[idx];
        profits[idx] = -1; // Mark as used
    }
    return w;
}
```

OUTPUT :



Prefix Tree

1. **Implement Trie (Prefix Tree)**

Problem Statement:

A trie (pronounced as "try") or prefix tree is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. 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.

Sample Input:

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

Sample Output:

True

False

True

True

Solution :

```
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#define CHAR_SIZE 26

typedef struct TrieNode {
   struct TrieNode* children[CHAR_SIZE];
   bool isEnd;
} TrieNode;

typedef struct {
   TrieNode* root;
} Trie;

// Helper function to create a TrieNode
TrieNode* createNode() {
   TrieNode* node = (TrieNode*)malloc(sizeof(TrieNode));
   node->isEnd = false;
   for (int i = 0; i < CHAR_SIZE; i++) {
      node->children[i] = NULL;
   }
   return node;
}

Trie* trieCreate() {
```

```
    Trie* obj = (Trie*)malloc(sizeof(Trie));
    obj->root = createNode();
    return obj;
}

void trieInsert(Trie* obj, char* word) {
    TrieNode* curr = obj->root;
    for (int i = 0; word[i]; i++) {
        int index = word[i] - 'a';
        if (!curr->children[index]) {
            curr->children[index] = createNode();
        }
        curr = curr->children[index];
    }
    curr->isEnd = true;
}

bool trieSearch(Trie* obj, char* word) {
    TrieNode* curr = obj->root;
    for (int i = 0; word[i]; i++) {
        int index = word[i] - 'a';
        if (!curr->children[index]) return false;
        curr = curr->children[index];
    }
    return curr->isEnd;
}
```

```
bool trieStartsWith(Trie* obj, char* prefix) {
   TrieNode* curr = obj->root;
   for (int i = 0; prefix[i]; i++) {
      int index = prefix[i] - 'a';
      if (!curr->children[index]) return false;
      curr = curr->children[index];
   }
   return true;
}

// Helper function to recursively free nodes
void freeNode(TrieNode* node) {
   for (int i = 0; i < CHAR_SIZE; i++) {
      if (node->children[i]) {
         freeNode(node->children[i]);
      }
   }
   free(node);
}

void trieFree(Trie* obj) {
   freeNode(obj->root);
   free(obj);
}
);
```

OUTPUT :

2. **Replace Words**

Problem Statement:

In English, we have a concept called root, which can be followed by some other words to form another longer word - let's call this word successor. Given a dictionary of roots and a sentence, replace all the successors in the sentence with the root forming it. If a successor can be replaced by multiple roots, replace it with the root that has the shortest length.

Sample Input:
dictionary = ["cat","bat","rat"]
sentence = "the cattle was rattled by the battery"

Sample Output:
"the cat was rat by the bat"

Solution :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

typedef struct Trie {
  struct Trie* children[26];
  bool isEnd;
} Trie;

Trie* createNode() {
  Trie* node = (Trie*)malloc(sizeof(Trie));
  for (int i = 0; i < 26; ++i)
    node->children[i] = NULL;
  node->isEnd = false;
  return node;
```

```
}

void insert(Trie* root, char* word) {
  Trie* curr = root;
  while (*word) {
    int idx = *word - 'a';
    if (!curr->children[idx])
      curr->children[idx] = createNode();
    curr = curr->children[idx];
    word++;
  }
  curr->isEnd = true;
}

char* findRoot(Trie* root, char* word) {
  Trie* curr = root;
  char* p = word;
  int len = 0;
  while (*p) {
    int idx = *p - 'a';
    if (!curr->children[idx])
      break;
    curr = curr->children[idx];
    len++;
    if (curr->isEnd)
      break;
    p++;
  }
  if (curr->isEnd) {
    char* rootWord = (char*)malloc(len + 1);
    strncpy(rootWord, word, len);
    rootWord[len] = '\0';
    return rootWord;
  }
  return strdup(word);
}

char* replaceWords(char** dictionary, int dictSize, char* sentence) {
  Trie* root = createNode();
  for (int i = 0; i < dictSize; i++)
    insert(root, dictionary[i]);

  char* result = (char*)malloc(10000);
```

```
    result[0] = '\0';

    char* temp = strdup(sentence);
    char* token = strtok(temp, " ");
    while (token) {
        char* replacement = findRoot(root, token);
        strcat(result, replacement);
        token = strtok(NULL, " ");
        if (token)
            strcat(result, " ");
        free(replacement);
    }
    free(temp);
    return result;
}
```

OUTPUT :

3. Word Search II

Problem Statement:

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.

Sample Input:

board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]]

words = ["oath","pea","eat","rain"]

Sample Output:

["oath","eat"]

Solution :

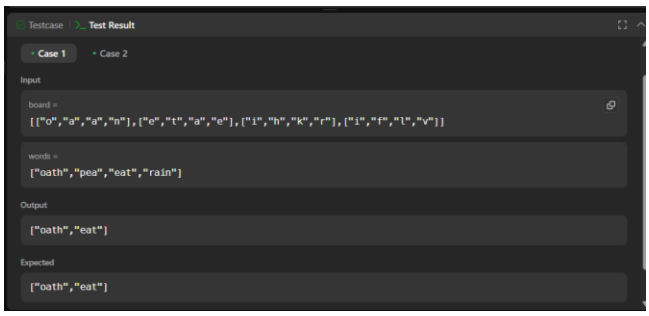
```
#define MAX_WORD_LEN 100
typedef struct TrieNode {
   struct TrieNode* children[26];
   bool isEnd;
} TrieNode;

TrieNode* createNode() {
   TrieNode* node = (TrieNode*)malloc(sizeof(TrieNode));
   node->isEnd = false;
   for (int i = 0; i < 26; i++) node->children[i] = NULL;
   return node;
}

void insert(TrieNode* root, char* word) {
   TrieNode* curr = root;
   for (int i = 0; word[i]; i++) {
      int idx = word[i] - 'a';
      if (!curr->children[idx]) curr->children[idx] = createNode();
      curr = curr->children[idx];
   }
   curr->isEnd = true;
}
```

```
typedef struct {
    char** words;
    int count;
} WordList;

void dfs(char** board, int rows, int* cols, int i, int j, TrieNode* node, char* path, int depth,
bool** visited, WordList* result) {
    if (i < 0 || j < 0 || i >= rows || j >= cols[i] || visited[i][j]) return;
    char c = board[i][j];
    if (!node->children[c - 'a']) return;

    visited[i][j] = true;
    path[depth] = c;
    TrieNode* next = node->children[c - 'a'];

    if (next->isEnd) {
        path[depth + 1] = '\0';
        result->words[result->count++] = strdup(path);
        next->isEnd = false;
    }

    dfs(board, rows, cols, i + 1, j, next, path, depth + 1, visited, result);
    dfs(board, rows, cols, i - 1, j, next, path, depth + 1, visited, result);
    dfs(board, rows, cols, i, j + 1, next, path, depth + 1, visited, result);
    dfs(board, rows, cols, i, j - 1, next, path, depth + 1, visited, result);
```

```
    visited[i][j] = false;
}

char** findWords(char** board, int boardSize, int* boardColSize, char** words, int
wordsSize, int* returnSize) {
    TrieNode* root = createNode();
    for (int i = 0; i < wordsSize; i++) insert(root, words[i]);

    WordList result;
    result.words = (char**)malloc(1000 * sizeof(char*));
    result.count = 0;

    bool** visited = (bool**)malloc(boardSize * sizeof(bool*));
    for (int i = 0; i < boardSize; i++)
        visited[i] = (bool*)calloc(boardColSize[i], sizeof(bool));

    char path[MAX_WORD_LEN];
    for (int i = 0; i < boardSize; i++)
        for (int j = 0; j < boardColSize[i]; j++)
            dfs(board, boardSize, boardColSize, i, j, root, path, 0, visited, &result);

    for (int i = 0; i < boardSize; i++) free(visited[i]);
    free(visited);

    *returnSize = result.count;
    return result.words;
```

}

OUTPUT :



GRAPHS

1. **Rotting Oranges**

   Question :
   You are given an m x n grid where each cell can have one of three values:

- 0 representing an empty cell,
- 1 representing a fresh orange, or
- 2 representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange*.
If *this is impossible, return* -1.

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

Output: 4

Solution :

```
#define MAX 10000

int orangesRotting(int** grid, int gridSize, int* gridColSize){

   int m = gridSize, n = gridColSize[0];

   int queue[MAX][2], front = 0, rear = 0, fresh = 0, time = 0;
```

```
// Enqueue all rotten oranges
for (int i = 0; i < m; i++) {
   for (int j = 0; j < n; j++) {
      if (grid[i][j] == 2) {
         queue[rear][0] = i;
         queue[rear][1] = j;
         rear++;
      } else if (grid[i][j] == 1) {
         fresh++;
      }
   }
}

int dirs[4][2] = {{0,1},{1,0},{0,-1},{-1,0}};

while (front < rear) {
   int size = rear - front;
   int newRot = 0;
   for (int i = 0; i < size; i++) {
      int x = queue[front][0], y = queue[front][1];
      front++;
      for (int d = 0; d < 4; d++) {
         int nx = x + dirs[d][0], ny = y + dirs[d][1];
         if (nx >= 0 && ny >= 0 && nx < m && ny < n && grid[nx][ny] == 1) {
            grid[nx][ny] = 2;
```

```
                queue[rear][0] = nx;
                queue[rear][1] = ny;
                rear++;
                fresh--;
                newRot = 1;
            }
          }
        }
        if (newRot) time++;
    }

    return fresh == 0 ? time : -1;
}
```

OUTPUT :

2. **Find the Town Judge**

   Question :
   In a town, there are n people labeled from 1 to n. There is a rumor that one of these people is secretly the town judge.

   If the town judge exists, then:

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

You are given an array trust where trust[i] = [a_i, b_i] representing that the person labeled a_i trusts the person labeled b_i. If a trust relationship does not exist in trust array, then such a trust relationship does not exist.

Return *the label of the town judge if the town judge exists and can be identified, or return* -1 *otherwise*.

Example 1:

Input: n = 2, trust = [[1,2]]

Output: 2

Solution :

```
int findJudge(int n, int** trust, int trustSize, int* trustColSize){
   int inDegree[1001] = {0};
   int outDegree[1001] = {0};

   for (int i = 0; i < trustSize; i++) {
      int a = trust[i][0], b = trust[i][1];
      outDegree[a]++;
      inDegree[b]++;
   }

   for (int i = 1; i <= n; i++) {
      if (inDegree[i] == n - 1 && outDegree[i] == 0)
         return i;
   }

   return -1;
}
```

OUTPUT :

3. Number of Provinces

Question :

There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.

A province is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the i^{th} city and the j^{th} city are directly connected, and isConnected[i][j] = 0 otherwise.

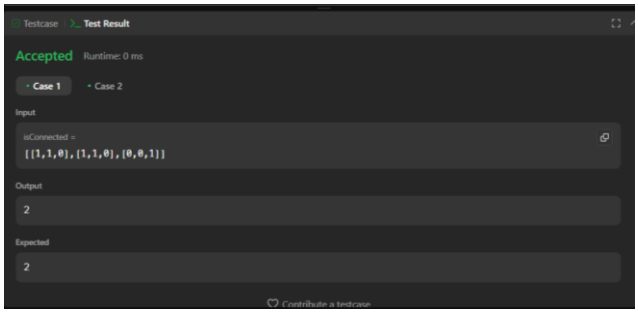
Return *the total number of provinces*.
Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]

Output: 2
Solution :

```
void dfs(int** isConnected, int n, int* visited, int i) {
   visited[i] = 1;
   for (int j = 0; j < n; j++) {
      if (isConnected[i][j] == 1 && !visited[j]) {
         dfs(isConnected, n, visited, j);
      }
   }
}
```

```
int findCircleNum(int** isConnected, int isConnectedSize, int* isConnectedColSize){
   int* visited = calloc(isConnectedSize, sizeof(int));
   int count = 0;
   for (int i = 0; i < isConnectedSize; i++) {
      if (!visited[i]) {
         dfs(isConnected, isConnectedSize, visited, i);
         count++;
      }
   }
   return count;
}
```

OUTPUT :



4. Course Schedule

Question :
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] = [a_i, b_i] indicates that you **must** take course b_i first if you want to take course a_i.

- 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.

Solution :

```
bool dfs(int** graph, int* graphColSize, int* visited, int course) {
   if (visited[course] == -1) return false;  // cycle
   if (visited[course] == 1) return true;    // already checked

   visited[course] = -1;  // mark as visiting
   for (int i = 0; i < graphColSize[course]; i++) {
      if (!dfs(graph, graphColSize, visited, graph[course][i]))
         return false;
   }
   visited[course] = 1;  // mark as visited
   return true;
}

bool canFinish(int numCourses, int** prerequisites, int prerequisitesSize, int*
prerequisitesColSize){
   int** graph = calloc(numCourses, sizeof(int*));
   int* graphColSize = calloc(numCourses, sizeof(int));

   for (int i = 0; i < prerequisitesSize; i++) {
      int a = prerequisites[i][0], b = prerequisites[i][1];
      graphColSize[b]++;
   }
```

```
    for (int i = 0; i < numCourses; i++) {
        graph[i] = calloc(graphColSize[i], sizeof(int));
        graphColSize[i] = 0;
    }

    for (int i = 0; i < prerequisitesSize; i++) {
        int a = prerequisites[i][0], b = prerequisites[i][1];
        graph[b][graphColSize[b]++] = a;
    }

    int* visited = calloc(numCourses, sizeof(int));
    for (int i = 0; i < numCourses; i++) {
        if (!dfs(graph, graphColSize, visited, i)) return false;
    }

    return true;
}
```

OUTPUT :