VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
On

ANALYSIS AND DESIGN OF ALGORITHMS (23CS4PCADA)

Submitted by

Dama Yohitesh Naveen Sai(1BM23CS085)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
February-May 2025

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

This is to certify that the Lab work entitled **“ANALYSIS AND DESIGN OF ALGORITHMS”** carried out by Dama Yohitesh Naveen Sai **(1BM23CS085)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of Analysis and Design of Algorithms Lab - **(23CS4PCADA)** work prescribed for the said degree.

Dr. K. Panimozhi
Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Sl. No.	Experiment Title	Page No.
1	a. Write a program to obtain the Topological ordering of vertices in a given digraph. b. LeetCode Program related to Topological sorting	1
2	Implement Johnson Trotter algorithm to generate permutations.	6
3	a. Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort. b. LeetCode Program related to Binary Search Tree.	9
4	a. Sort a given set of N integer elements using Quick Sort technique and compute its time taken. b. LeetCode Program related to Binary Trees.	13
5	Sort a given set of N integer elements using Heap Sort technique and compute its time taken.	19
6	a. Implement 0/1 Knapsack problem using dynamic programming. b. LeetCode Program related to Knapsack problem or Dynamic Programming.	21
7	a. Implement All Pair Shortest paths problem using Floyd's algorithm. b. LeetCode Program related to shortest distance calculation.	25
8	a. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm. b. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.	30
9	a. Fractional Knapsack using Greedy technique. b. LeetCode Program related to Greedy Technique algorithms.	35
10	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.	40
11	Implement "N-Queens Problem" using Backtracking.	43

Course outcomes:

CO1	Analyze time complexity of Recursive and Non-recursive algorithms using asymptotic notations.
CO2	Apply various design techniques for the given problem.
CO3	Apply the knowledge of complexity classes P, NP, and NP-Complete and prove certain problems are NP-Complete
CO4	Design efficient algorithms and conduct practical experiments to solve problems.

Lab program 1

a. Write a program to obtain the Topological ordering of vertices in a given digraph.

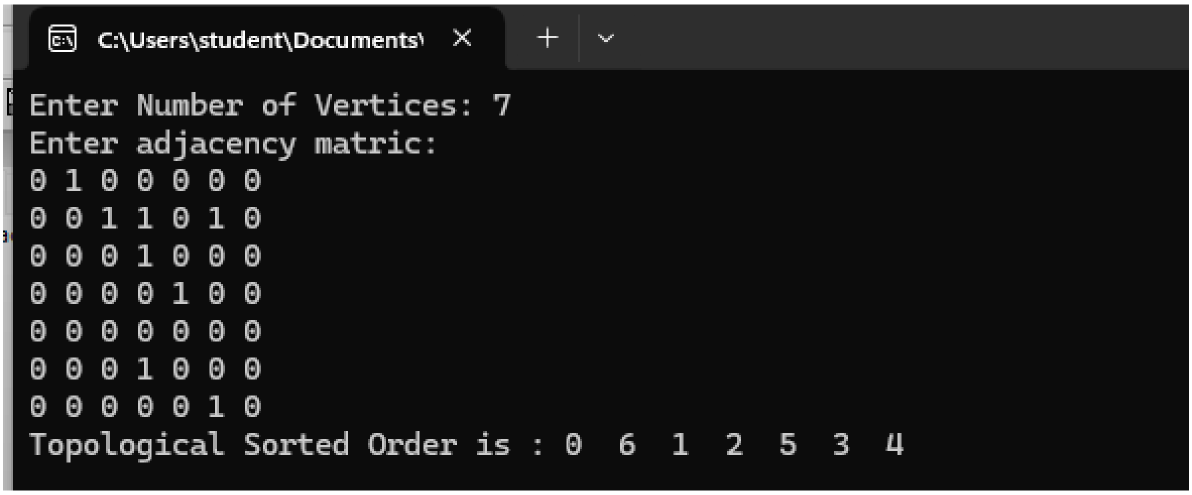
Code :

```
#include<stdio.h>

#include<stdlib.h>

#define SIZE 100

void topological_sort(int graph[SIZE][SIZE],int n){

    int indegree[n];

    for(int i = 0; i<n; i++) indegree[i]=0;


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

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

            if(graph[i][j]==1){

                indegree[j]++;

            }

        }

    }


    int source[n];

    int front = 0;

    int last = 0;


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

        if(indegree[i]==0) source[last++]=i;

    }


    int sorted[n];

    int ind = 0;
```

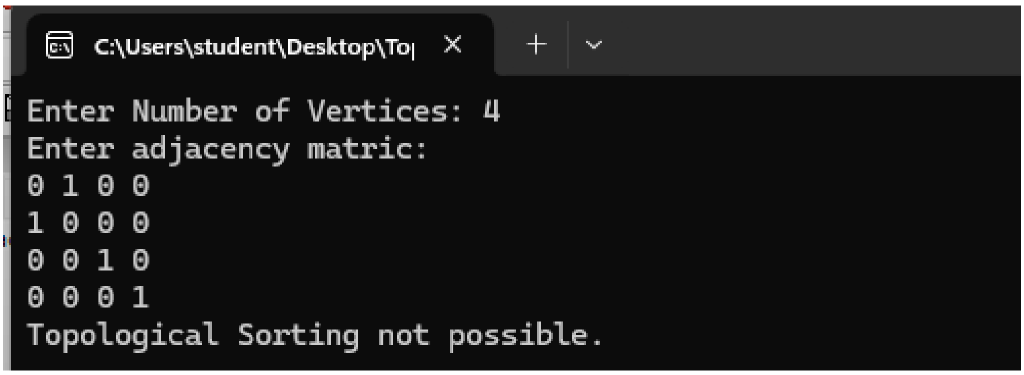
```
    while(front<last){
        int temp = source[front++];
        sorted[ind++]= temp;
        for(int i = 0; i<n; i++){
            if(graph[temp][i]==1){
                indegree[i]--;
                if(indegree[i]==0)source[last++]=i;
            }
        }
    }

    if(ind == n){
        printf("Topological Sorted Order is :");
        for(int i = 0; i<n;i++){
          printf(" %d ",sorted[i]);
        }
        printf("\n");
    }
    else{
        printf("Topological Sorting not possible.\n");
    }

}

int main(){
    int graph[SIZE][SIZE];
    int n;

    printf("Enter Number of Vertices: ");
```

```
    scanf("%d",&n);


    printf("Enter adjacency matric:\n");
    for(int i = 0;i<n;i++){
        for(int j = 0; j<n;j++){
            scanf("%d",&graph[i][j]);
        }
    }
    topological_sort(graph,n);
    return 0;
}
```

<u>Output:</u>



```
C:\Users\student\Documents\

Enter Number of Vertices: 7
Enter adjacency matric:
0 1 0 0 0 0 0
0 0 1 1 0 1 0
0 0 0 1 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 0 1 0
Topological Sorted Order is : 0  6  1  2  5  3  4
```



```
C:\Users\student\Desktop\To

Enter Number of Vertices: 4
Enter adjacency matric:
0 1 0 0
1 0 0 0
0 0 1 0
0 0 0 1
Topological Sorting not possible.
```

b. LeetCode Program related to Topological sorting

LeetCode 851. Loud and Rich

There is a group of n people labeled from 0 to n - 1 where each person has a different amount of money and a different level of quietness.

You are given an array richer where richer[i] = [ai, bi] indicates that ai has more money than bi and an integer array quiet where quiet[i] is the quietness of the ith person. All the given data in richer are logically correct (i.e., the data will not lead you to a situation where x is richer than y and y is richer than x at the same time).

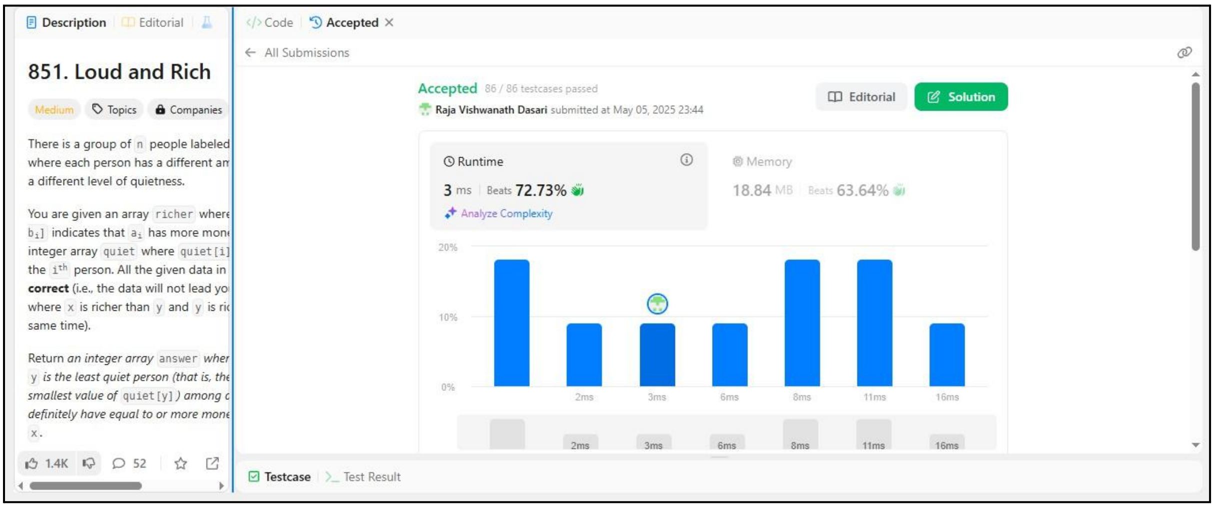
Return an integer array answer where answer[x] = y if y is the least quiet person (that is, the person y with the smallest value of quiet[y]) among all people who definitely have equal to or more money than the person x.

<u>Submission Code:</u>

```
#define MAXN 510

int* loudAndRich(int** richer, int richerSize, int* richerColSize, int* quiet, int quietSize, int* returnSize) {
    int n = quietSize;
    int graph[MAXN][MAXN] = {0};
    int indegree[MAXN] = {0};
    int answer[MAXN];
    int queue[MAXN], front = 0, rear = 0;

    for (int i = 0; i < richerSize; i++) {
        int a = richer[i][0], b = richer[i][1];
        if (!graph[a][b]) {
            graph[a][b] = 1;
            indegree[b]++;
        }
    }

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

    while (front < rear) {
        int u = queue[front++];
```

```
        for (int v = 0; v < n; v++) {
            if (graph[u][v]) {
                if (quiet[answer[u]] < quiet[answer[v]]) {
                    answer[v] = answer[u];
                }
                indegree[v]--;
                if (indegree[v] == 0) {
                    queue[rear++] = v;
                }
            }
        }
    }

    int* result = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        result[i] = answer[i];
    }
    *returnSize = n;
    return result;
}
```

Result:



Lab Program 2

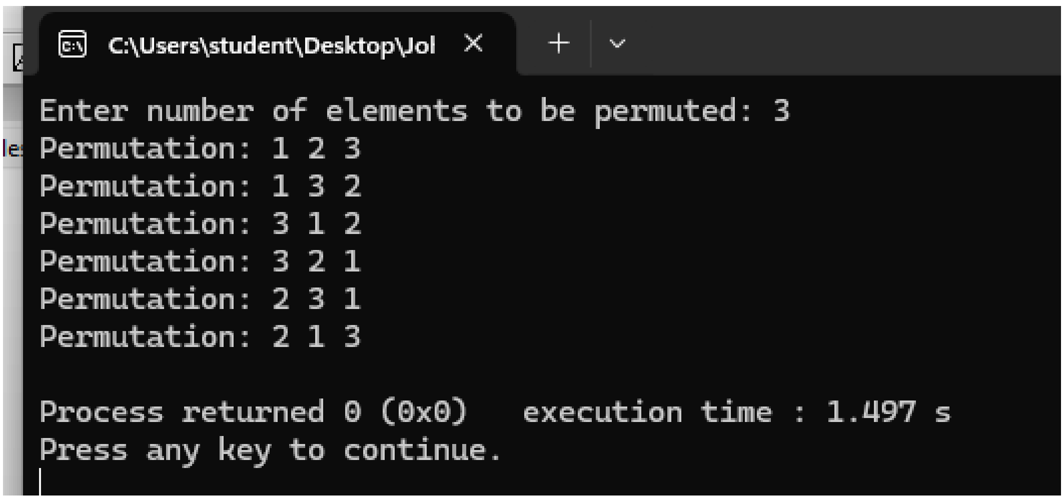
Implement Johnson Trotter algorithm to generate permutations.

Code:

```
#include<stdio.h>
#include<stdlib.h>
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
void print_perm(int p[],int n){
    printf("Permutation: ");
    for(int i = 0;i<n; i++){
        printf("%d ",p[i]);
    }
    printf("\n");
}
void johnson_trotter(int n){
    int p[n];
    int d[n];
    for(int i = 0;i<n;i++){
        p[i] = i+1;
        d[i] = -1;
    }
    print_perm(p,n);

    while(1){
        int largest_mob = -1;
        int largest_ind = -1;

        for(int i=0; i<n;i++){
            if(d[i] == -1 && i>0 && p[i]>p[i-1]){
                if(p[i]>largest_mob){
                    largest_mob = p[i];
                    largest_ind = i;
                }
            }
            else if(d[i] == 1 && i<n-1 && p[i]>p[i+1]){
                if(p[i]>largest_mob){
                    largest_mob = p[i];
```

```
                largest_ind = i;
            }
        }
    }
    if(largest_mob == -1) break;
    int ind = largest_ind;
    if(d[ind]==-1){
        swap(&p[ind],&p[ind-1]);
        swap(&d[ind],&d[ind-1]);
    }
    else if(d[ind]==1){
        swap(&p[ind],&p[ind+1]);
        swap(&d[ind],&d[ind+1]);
    }
    for(int j = 0;j<n;j++){
        if(p[j]>largest_mob){
            d[j]= d[j]*-1;
        }
    }
  print_perm(p,n);
 }
}

int main(){
    int n;
    printf("Enter number of elements to be permuted: ");
    scanf("%d",&n);
    johnson_trotter(n);
}
```

Output:



```
C:\Users\student\Desktop\Jol
Enter number of elements to be permuted: 3
Permutation: 1 2 3
Permutation: 1 3 2
Permutation: 3 1 2
Permutation: 3 2 1
Permutation: 2 3 1
Permutation: 2 1 3

Process returned 0 (0x0)   execution time : 1.497 s
Press any key to continue.
```

C:\Users\student\Desktop\Jol

```
Enter number of elements to be permuted: 4
Permutation: 1 2 3 4
Permutation: 1 2 4 3
Permutation: 1 4 2 3
Permutation: 4 1 2 3
Permutation: 4 1 3 2
Permutation: 1 4 3 2
Permutation: 1 3 4 2
Permutation: 1 3 2 4
Permutation: 3 1 2 4
Permutation: 3 1 4 2
Permutation: 3 4 1 2
Permutation: 4 3 1 2
Permutation: 4 3 2 1
Permutation: 3 4 2 1
Permutation: 3 2 4 1
Permutation: 3 2 1 4
Permutation: 2 3 1 4
Permutation: 2 3 4 1
Permutation: 2 4 3 1
Permutation: 4 2 3 1
Permutation: 4 2 1 3
Permutation: 2 4 1 3
Permutation: 2 1 4 3
Permutation: 2 1 3 4

Process returned 0 (0x0)   execution time : 5.506 s
Press any key to continue.
```

Lab Program 3

a. Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.

Code:

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

// Merge function
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}

// Merge Sort function
void merge_sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);
        merge(arr, left, mid, right);
```

```
    }
}

int main() {
    int n_values[] =
{100000,200000,300000,400000,500000,600000,700000,800000,900000,1000000};
    int num_tests = sizeof(n_values) / sizeof(n_values[0]);

    for (int i = 0; i < num_tests; i++) {
        int n = n_values[i];
        int *array = (int *)malloc(n * sizeof(int));

        // Average Case: Randomized Input
        for (int j = 0; j < n; j++) array[j] = rand() % 1000000;
        clock_t start = clock();
        merge_sort(array, 0, n - 1);
        clock_t end = clock();
        printf("n = %d | Avg: %.6f\n", n, (double)(end - start) / CLOCKS_PER_SEC);

        free(array);
    }

    return 0;
}
```

Output:

```
Merge Sort Time Complexity Analysis:
n = 100000 | Avg: 0.037664
n = 200000 | Avg: 0.079346
n = 300000 | Avg: 0.120634
n = 400000 | Avg: 0.164329
n = 500000 | Avg: 0.209077
n = 600000 | Avg: 0.251556
n = 700000 | Avg: 0.294781
n = 800000 | Avg: 0.339544
n = 900000 | Avg: 0.386991
n = 1000000 | Avg: 0.429733
```

b. LeetCode Program related to Binary Search Tree.

LeetCode 230. Kth Smallest Element in a BST

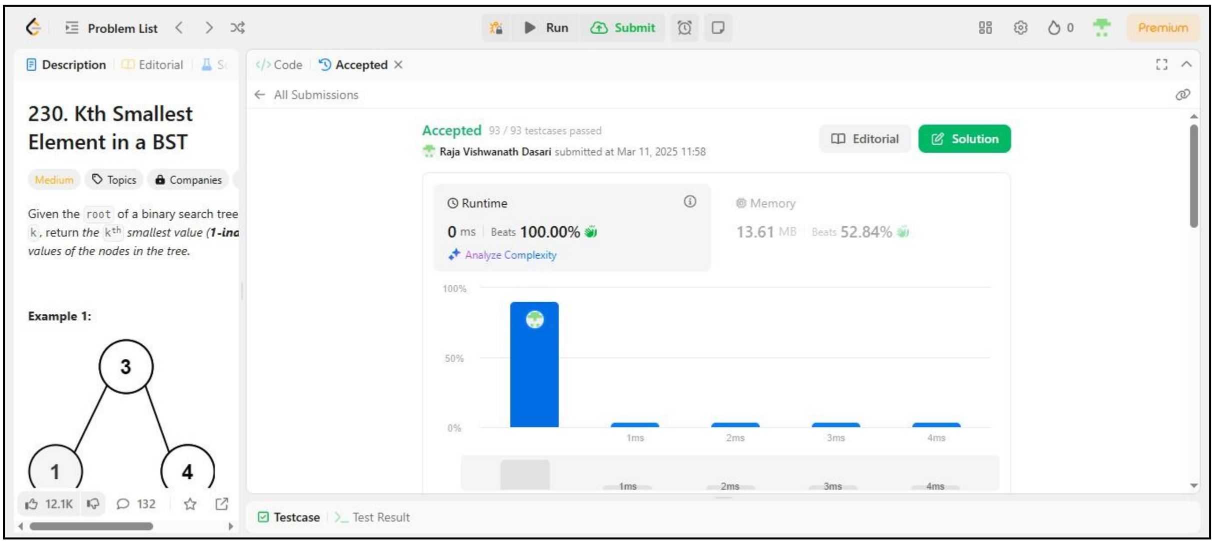
Given the root of a binary search tree, and an integer k, return the kth smallest value (1-indexed) of all the values of the nodes in the tree.

<u>Submission Code:</u>

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
#include <stdio.h>
#include <stdlib.h>

int kthSmallest(struct TreeNode* root, int k) {
    int findMin(struct TreeNode* root){
        while(root->left!=NULL){
            root = root->left;
        }
        return root->val;
    }
    struct TreeNode* delete_min(struct TreeNode* root){
        struct TreeNode* parent = root;
        struct TreeNode* child = root;
        if(root==NULL){
            return root;
        }
        while(child->left!=NULL){
            parent = child;
            child = child->left;
        }
        parent->left = child->right;
        return root;
    }
    for(int i = 0 ; i<k-1; i++){
        root = delete_min(root);
    }
```

```
    int r = findMin(root);
    return r;
}
```

Result:



Lab Program 4

a. Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

Code:

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

// Swap function
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Median-of-three pivot selection for best case
int median_of_three(int arr[], int low, int high) {
    int mid = low + (high - low) / 2;
    if (arr[low] > arr[mid]) swap(&arr[low], &arr[mid]);
    if (arr[low] > arr[high]) swap(&arr[low], &arr[high]);
    if (arr[mid] > arr[high]) swap(&arr[mid], &arr[high]);
    return mid;  // Return index of the median element
}

// Partition function using last element as pivot
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Worst and average case use last element as pivot
```

```
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// Partition function using median-of-three pivot for best case
int partition_median(int arr[], int low, int high) {
    int median = median_of_three(arr, low, high);
    swap(&arr[median], &arr[high]);  // Move median pivot to the end
    return partition(arr, low, high);
}

// Quick sort with normal partitioning
void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}
```

```
// Quick sort with median pivot for best case
void quick_sort_best(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition_median(arr, low, high);
        quick_sort_best(arr, low, pi - 1);
        quick_sort_best(arr, pi + 1, high);
    }
}

int main() {
    int n_values[] = {10000,20000,30000,40000,50000,60000,70000,80000,90000,100000};
    int num_tests = sizeof(n_values) / sizeof(n_values[0]);

    for (int i = 0; i < num_tests; i++) {
        int n = n_values[i];
        int *array = (int *)malloc(n * sizeof(int));

        //  Best Case: Median Pivot (Balanced Partitioning)
        for (int j = 0; j < n; j++) array[j] = j;
        clock_t start = clock();
        quick_sort_best(array, 0, n - 1);
        clock_t end = clock();
        printf("n = %d | Best: %.6f | ", n, (double)(end - start) / CLOCKS_PER_SEC);

        //  Average Case: Randomized Input
        for (int j = 0; j < n; j++) array[j] = rand() % 100000;
        start = clock();
        quick_sort(array, 0, n - 1);
        end = clock();
```

```
        printf("Avg: %.6f | ", (double)(end - start) / CLOCKS_PER_SEC);

        //  Worst Case: Already Sorted in Ascending Order (Last Element Pivot)
        for (int j = 0; j < n; j++) array[j] = j;
        start = clock();
        quick_sort(array, 0, n - 1);
        end = clock();
        printf("Worst: %.6f\n", (double)(end - start) / CLOCKS_PER_SEC);

        free(array);
    }

    return 0;
}
```

Output:

```
n = 10000 | Best: 0.000733 | Avg: 0.001543 | Worst: 0.349421
n = 20000 | Best: 0.001842 | Avg: 0.003396 | Worst: 1.364989
n = 30000 | Best: 0.002432 | Avg: 0.005338 | Worst: 3.761610
n = 40000 | Best: 0.003336 | Avg: 0.007314 | Worst: 5.478223
n = 50000 | Best: 0.004516 | Avg: 0.009029 | Worst: 9.220020
n = 60000 | Best: 0.005158 | Avg: 0.011119 | Worst: 13.013741
n = 70000 | Best: 0.005970 | Avg: 0.013162 | Worst: 17.445868
n = 80000 | Best: 0.007140 | Avg: 0.015433 | Worst: 23.256661
n = 90000 | Best: 0.008138 | Avg: 0.018496 | Worst: 29.237170
n = 100000 | Best: 0.012229 | Avg: 0.024076 | Worst: 35.982174
```

b. LeetCode Program related to Binary Trees.

LeetCode 124. 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.

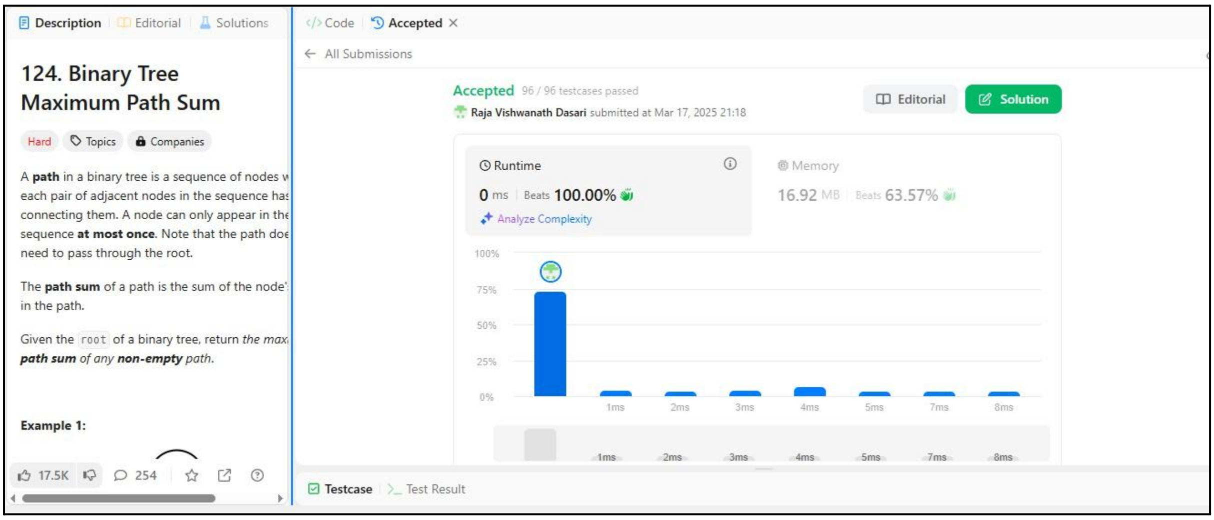
The path sum of a path is the sum of the node's values in the path.

Given the root of a binary tree, return the maximum path sum of any non-empty path.

Submission Code:

```
struct TreeNode {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
};

int dfs(struct TreeNode* root, int* res) {
    if (root == NULL) {
        return 0;
    }

    int left = dfs(root->left, res);
    int right = dfs(root->right, res);

    left = (left > 0) ? left : 0;
    right = (right > 0) ? right : 0;

    *res = (*res > left + right + root->val) ? *res : (left + right + root->val);

    return root->val + ((left > right) ? left : right);
}

int maxPathSum(struct TreeNode* root) {
    int res = -999999999;
    dfs(root, &res);
    return res;
}
```

Result:



Description
Editorial
Solutions
124. Binary Tree Maximum Path Sum
Hard
Topics
Companies
Code
Accepted
All Submissions
Accepted 96 / 96 testcases passed
Raja Vishwanath Dasari submitted at Mar 17, 2025 21:18
Editorial
Solution
Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
16.92 MB | Beats 63.57%
Example 1:
17.5K
254
Testcase
Test Result

Lab Program 5

Sort a given set of N integer elements using Heap Sort technique and compute its time taken.

<u>Code:</u>

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

void heapify(int arr[], int n, int i) {
    int l = 2 * i + 1, r = 2 * i + 2, m = i;
    if (l < n && arr[l] > arr[m]) m = l;
    if (r < n && arr[r] > arr[m]) m = r;
    if (m != i) {
        int t = arr[i];
        arr[i] = arr[m];
        arr[m] = t;
        heapify(arr, n, m);
    }
}

void heapsort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        int t = arr[0];
        arr[0] = arr[i];
        arr[i] = t;
        heapify(arr, i, 0);
    }
}

int main() {
    int n_values[] =
{100000,200000,300000,400000,500000,600000,700000,800000,900000,1000000};
    int num_tests = sizeof(n_values) / sizeof(n_values[0]);
    printf("Heap Sort Time Complexity Analysis: \n");
    for (int i = 0; i < num_tests; i++) {
        int n = n_values[i];
        int *array = (int *)malloc(n * sizeof(int));
        for (int j = 0; j < n; j++) array[j] = rand() % 1000000;
```

```
        clock_t start = clock();
        heapsort(array, n);
        clock_t end = clock();

        printf("n = %d | Avg: %.6f sec\n", n, (double)(end - start) / CLOCKS_PER_SEC);
        free(array);
    }
    return 0;
}
```

Output:

```
Heap Sort Time Complexity Analysis:
n = 100000 | Avg: 0.032810 sec
n = 200000 | Avg: 0.076087 sec
n = 300000 | Avg: 0.109885 sec
n = 400000 | Avg: 0.151516 sec
n = 500000 | Avg: 0.198166 sec
n = 600000 | Avg: 0.235795 sec
n = 700000 | Avg: 0.294872 sec
n = 800000 | Avg: 0.331766 sec
n = 900000 | Avg: 0.369062 sec
n = 1000000 | Avg: 0.435496 sec
```

Lab Program 6

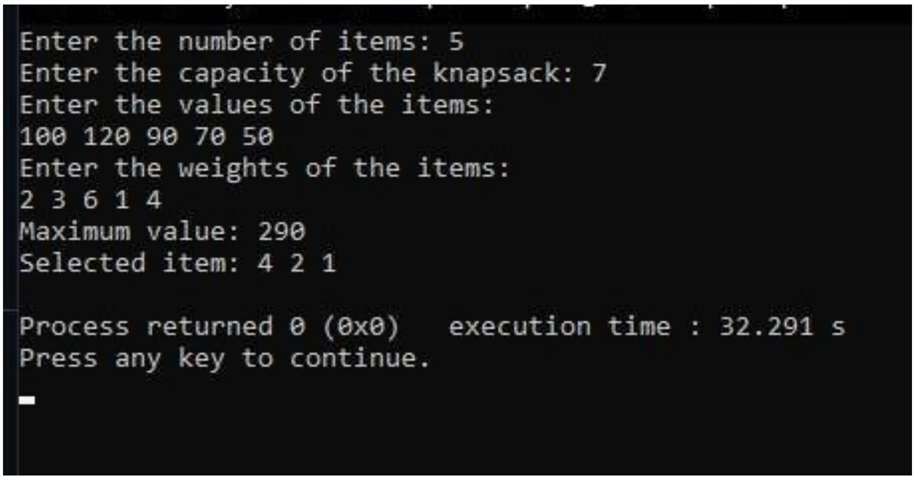
a. Implement 0/1 Knapsack problem using dynamic programming

Code:

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

int max(int a,int b){
    int m = a>b?a:b;
    return m;
}

void knapsack(int v[], int w[], int n, int W){
    int dp[n+1][W+1];
    for(int i = 0;i<=n; i++){
        for(int j = 0; j<=W; j++){
            dp[i][j] = 0;
        }
    }

    for(int i = 1; i<=n; i++){
        for(int j = 1; j<=W; j++){
            if(w[i-1]<=j){
                dp[i][j] = max(dp[i-1][j], v[i-1]+dp[i-1][j-w[i-1]]);
            }
            else{
                dp[i][j]=dp[i-1][j];
            }
        }
    }
    printf("Maximum value: %d\n", dp[n][W]);

    printf("Selected item: ");
    int res = dp[n][W];
    int j = W;
    for (int i = n; i > 0 && res > 0; i--) {
        if (res != dp[i-1][j]) {
            printf("%d ", i);
            res -= v[i-1];
            j -= w[i-1];
```

```
        }
    }
    printf("\n");
}

int main() {
    int n, W;
    int values[100], weights[100];
    printf("Enter the number of items: ");
    scanf("%d", &n);
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &W);
    printf("Enter the values of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &values[i]);
    }
    printf("Enter the weights of the items:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &weights[i]);
    }

    knapsack(values, weights, n, W);

    return 0;
}
```

Output:



b. LeetCode Program related to Knapsack problem or Dynamic Programming.

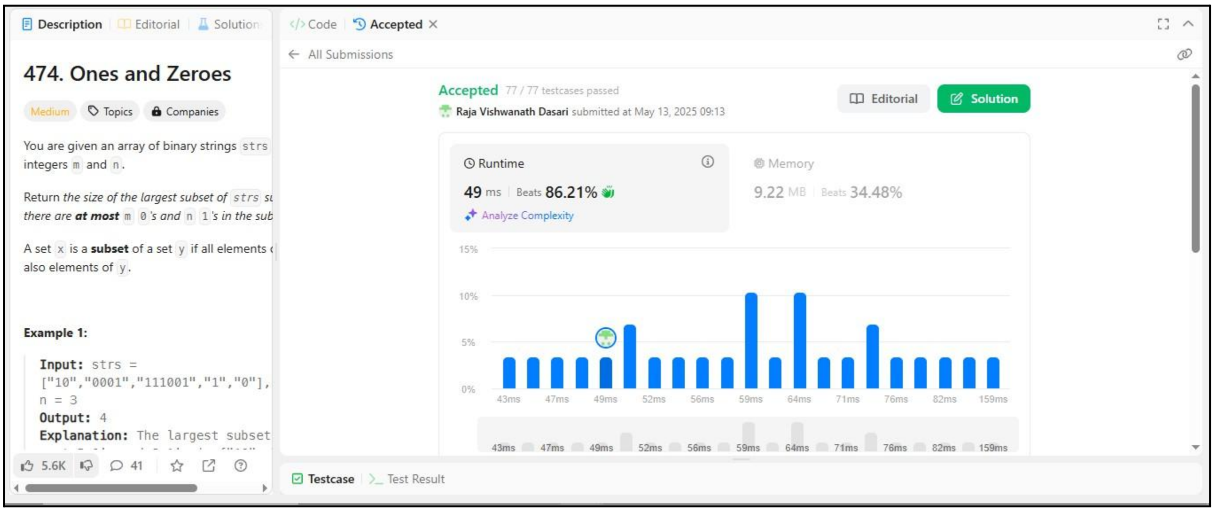
LeetCode 474. Ones and Zeroes

You are given an array of binary strings strs and two integers m and n.
Return the size of the largest subset of strs such that there are at most m 0's and n 1's in the subset.
A set x is a subset of a set y if all elements of x are also elements of y.

<u>Submission Code:</u>

```
#include <string.h>

int findMaxForm(char** strs, int strsSize, int m, int n) {
    int dp[101][101] = {0};

    for (int k = 0; k < strsSize; ++k) {
        int zeros = 0, ones = 0;
        for (int i = 0; strs[k][i]; ++i) {
            if (strs[k][i] == '0') zeros++;
            else ones++;
        }
        for (int i = m; i >= zeros; --i) {
            for (int j = n; j >= ones; --j) {
                if (dp[i][j] < dp[i - zeros][j - ones] + 1) {
                    dp[i][j] = dp[i - zeros][j - ones] + 1;
                }
            }
        }
    }

    return dp[m][n];
}
```

Result:


Description
Editorial
Solutions
Code
Accepted
All Submissions
474. Ones and Zeroes
Medium
Topics
Companies
Accepted 77 / 77 testcases passed
Raja Vishwanath Dasari submitted at May 13, 2025 09:13
Editorial
Solution
Runtime
49 ms | Beats 86.21%
Analyze Complexity
Memory
9.22 MB | Beats 34.48%
Example 1:
Testcase
Test Result

Lab Program 7

a. Implement All Pair Shortest paths problem using Floyd’s algorithm.

<u>Code:</u>

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

#define   SIZE   100
int min(int a, int b){
   int m = a>b?b:a;
   return m;
}
void floyd_warshall(int graph[SIZE][SIZE],int n){
   int D[n][n];
   for(int i = 0 ; i<n;i++){
      for(int j = 0; j<n ;j++){
         D[i][j] = graph[i][j];
      }
   }

   for(int k = 0;k<n;k++){
      for(int i = 0 ; i<n;i++){
         for(int j = 0; j<n ;j++){
            if(D[i][k]!= INT_MAX && D[k][j]!=INT_MAX){
               D[i][j]= min(D[i][j], D[i][k]+D[k][j]);
            }
         }
      }
   }
   printf("The final matrix D is:\n");
   for(int i = 0 ; i<n;i++){
      for(int j = 0; j<n ;j++){
         if(D[i][j]!=INT_MAX)printf("%d ",D[i][j]);
         else printf("INF ");
      }
      printf("\n");
   }
   for(int i = 0 ; i<n;i++){
      for(int j = 0; j<n ;j++){
```

```
            if(D[i][j]!=0 && D[i][j]!=INT_MAX){
                printf("Shortest Path from %d to %d : %d\n", i,j,D[i][j]);
            }
        }
    }
}

int main(){
    int n;
    int graph[SIZE][SIZE];

    printf("Enter number of vertices :");
    scanf("%d",&n);
    printf("For no direct edge enter -1\n");
    for(int i =0;i<n;i++){
        for(int j =0; j<n;j++) scanf("%d",&graph[i][j]);
    }

    for(int i =0;i<n;i++){
        for(int j =0; j<n;j++){
            if(graph[i][j]==-1) graph[i][j]=INT_MAX;
        }
    }
    floyd_warshall(graph,n);
    return 0;
}
```

Output:

```
Enter number of vertices :4
For no direct edge enter -1
0 -1 3 -1
2 0 -1 -1
-1 7 0 1
6 -1 -1 0
The final matrix D is:
0 10 3 4
2 0 5 6
7 7 0 1
6 16 9 0
Shortest Path from 0 to 1 : 10
Shortest Path from 0 to 2 : 3
Shortest Path from 0 to 3 : 4
Shortest Path from 1 to 0 : 2
Shortest Path from 1 to 2 : 5
Shortest Path from 1 to 3 : 6
Shortest Path from 2 to 0 : 7
Shortest Path from 2 to 1 : 7
Shortest Path from 2 to 3 : 1
Shortest Path from 3 to 0 : 6
Shortest Path from 3 to 1 : 16
Shortest Path from 3 to 2 : 9
```

```
Enter number of vertices :5
For no direct edge enter -1
0 6 2 4 -1
-1 0 -1 -1 -1
-1 3 0 -1 1
-1 1 -1 0 -1
-1 1 -1 -1 0
The final matrix D is:
0 4 2 4 3
INF 0 INF INF INF
INF 2 0 INF 1
INF 1 INF 0 INF
INF 1 INF INF 0
Shortest Path from 0 to 1 : 4
Shortest Path from 0 to 2 : 2
Shortest Path from 0 to 3 : 4
Shortest Path from 0 to 4 : 3
Shortest Path from 2 to 1 : 2
Shortest Path from 2 to 4 : 1
Shortest Path from 3 to 1 : 1
Shortest Path from 4 to 1 : 1
```

b. LeetCode Program related to shortest distance calculation.

LeetCode 1334. Find the City With the Smallest Number of Neighbors at a Threshold Distance

There are n cities numbered from 0 to n-1. Given the array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between cities fromi and toi, and given the integer distanceThreshold.

Return the city with the smallest number of cities that are reachable through some path and whose distance is at most distanceThreshold, If there are multiple such cities, return the city with the greatest number.

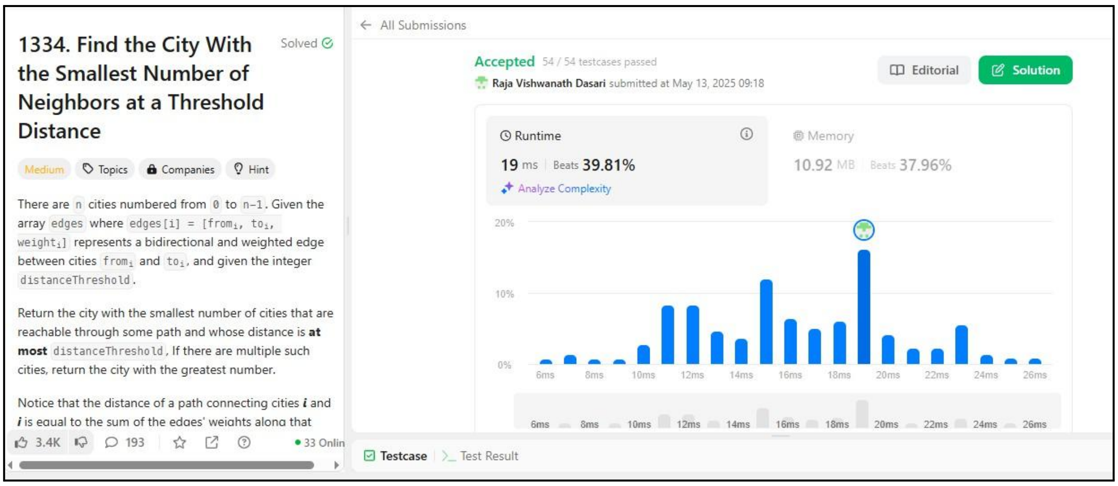
Notice that the distance of a path connecting cities i and j is equal to the sum of the edges' weights along that path.

Submission Code:

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

#define INF 1000000000 // A large number representing infinity

int findTheCity(int n, int** edges, int edgesSize, int* edgesColSize, int distanceThreshold) {
    int dist[100][100]; // n <= 100

    // Initialize distances
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dist[i][j] = (i == j) ? 0 : INF;
        }
    }

    // Set initial edge weights
    for (int i = 0; i < edgesSize; i++) {
        int u = edges[i][0];
        int v = edges[i][1];
        int w = edges[i][2];
        dist[u][v] = w;
        dist[v][u] = w; // because it's bidirectional
    }

    // Floyd-Warshall algorithm
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
```

```
            for (int j = 0; j < n; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    int result = -1;
    int minReachable = n + 1; // Start with max possible

    for (int i = 0; i < n; i++) {
        int count = 0;
        for (int j = 0; j < n; j++) {
            if (i != j && dist[i][j] <= distanceThreshold) {
                count++;
            }
        }

        // Choose the city with the smallest number of reachable cities.
        // If tied, choose the larger index.
        if (count <= minReachable) {
            minReachable = count;
            result = i;
        }
    }

    return result;
}
```

Result:



Lab Program 8

a. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

Code:

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

#define MAX 100

int minKey(int key[], int mstSet[], int V) {
    int min = INT_MAX, min_index = -1;
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}

void printMST(int parent[], int graph[MAX][MAX], int V) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
    }
}

void primMST(int graph[MAX][MAX], int V) {
    int parent[MAX];
    int key[MAX];
    int mstSet[MAX];

    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }

    key[0] = 0;
    parent[0] = -1;
```

```
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet, V);
        mstSet[u] = 1;

        for (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    printMST(parent, graph, V);
}

int main() {
    int V, graph[MAX][MAX];

    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    printf("Enter the adjacency matrix (0 if no edge):\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    primMST(graph, V);

    return 0;
}
```

Output:

```
Enter the number of vertices: 5
Enter the adjacency matrix (0 if no edge):
0 10 0 0 5
10 0 12 13 0
0 12 0 20 0
0 13 20 0 6
5 0 0 6 0
Edge    Weight
0 - 1   10
1 - 2   12
4 - 3   6
0 - 4   5

Process returned 0 (0x0)   execution time : 142.404 s
Press any key to continue.
```

b. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal’s algorithm

Code:

```c
#include <stdio.h>

#define MAX 100

int parent[MAX];

int find(int i) {
    while (parent[i] != i)
        i = parent[i];
    return i;
}

void unionSet(int i, int j) {
    int a = find(i);
    int b = find(j);
    parent[b] = a;
}

int main() {
    int V;
    int graph[MAX][MAX];
```

```
    printf("Enter number of vertices: ");
    scanf("%d", &V);

    printf("Enter the adjacency matrix (0 if no edge):\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    for (int i = 0; i < V; i++)
        parent[i] = i;

    int edgeCount = 0, minCost = 0;

    printf("Edge \tWeight\n");

    while (edgeCount < V - 1) {
        int min = 99999, a = -1, b = -1;

        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (graph[i][j] != 0 && graph[i][j] < min && find(i) != find(j)) {
                    min = graph[i][j];
                    a = i;
                    b = j;
                }
            }
        }

        if (a != -1 && b != -1) {
            unionSet(a, b);
            printf("%d - %d \t%d\n", a, b, min);
            minCost += min;
            edgeCount++;
        } else {
            break;
        }
    }

    printf("Total weight of MST: %d\n", minCost);

    return 0;
}
```

Output:

```
Enter number of vertices: 5
Enter the adjacency matrix (0 if no edge):
0 10 0 0 5
10 0 12 13 0
0 12 0 20 0
0 13 20 0 6
5 0 0 6 0
Edge    Weight
0 - 4   5
3 - 4   6
0 - 1   10
1 - 2   12
Total weight of MST: 33

Process returned 0 (0x0)   execution time : 114.168 s
Press any key to continue.
```

Lab Program 9

a. Fractional Knapsack using Greedy technique.

<u>Code:</u>

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

int partition(double arr[][3], int low, int high) {
    double pivot = arr[high][2];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j][2] > pivot) {
            i++;
            double temp0 = arr[i][0], temp1 = arr[i][1], temp2 = arr[i][2];
            arr[i][0] = arr[j][0];
            arr[i][1] = arr[j][1];
            arr[i][2] = arr[j][2];
            arr[j][0] = temp0;
            arr[j][1] = temp1;
            arr[j][2] = temp2;
        }
    }
    double temp0 = arr[i + 1][0], temp1 = arr[i + 1][1], temp2 = arr[i + 1][2];
    arr[i + 1][0] = arr[high][0];
    arr[i + 1][1] = arr[high][1];
    arr[i + 1][2] = arr[high][2];
    arr[high][0] = temp0;
    arr[high][1] = temp1;
    arr[high][2] = temp2;

    return i + 1;
}

void quick_sort(double arr[][3], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}
```

```
void fractional_knapsack(int v[], int w[], int n, int W) {
    double items[n][3];
    for (int i = 0; i < n; i++) {
        items[i][0] = v[i];
        items[i][1] = w[i];
        items[i][2] = (double)v[i] / (double)w[i];
    }

    quick_sort(items, 0, n - 1);

    double K = W;
    double total_val = 0.0;
    double x[n];
    for (int i = 0; i < n; i++) {
        x[i] = 0.0;
    }
    for (int i = 0; i < n && K > 0; i++) {
        if (items[i][1] <= K) {
            total_val += items[i][0];
            K -= items[i][1];
            x[i] = 1.0;
        } else {
            total_val += K * items[i][2];
            x[i] = K / items[i][1];
            K = 0;
        }
    }

    printf("Total Value: %f\n", total_val);
    printf("Item fractions: ");
    for (int i = 0; i < n; i++) {
        printf("%f ", x[i]);
    }
    printf("\n");
}

int main(){
    int n, W;

    printf("Enter capacity of knapsack: ");
    scanf("%d", &W);

    printf("Enter number of items: ");
```

```
    scanf("%d", &n);

    if (n <= 0) {
        printf("Invalid number of items.\n");
        return 0;
    }

    int v[n];
    int w[n];

    printf("Enter weights of items: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &w[i]);
    }

    printf("Enter values of items: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &v[i]);
    }

    fractional_knapsack(v, w, n, W);

    return 0;
}
```

<u>Output:</u>

```
Enter capacity of knapsack: 50
Enter number of items: 3
Enter weights of items: 10 20 30
Enter values of items: 60 100 120
Total Value: 240.000000
Item fractions: 1.000000 1.000000 0.666667

Process returned 0 (0x0)   execution time : 18.795 s
Press any key to continue.
```

b. LeetCode Program related to Greedy Technique algorithms.

LeetCode 1584. Min Cost to Connect All Points

You are given an array points representing integer coordinates of some points on a 2D-plane, where points[i] = [xi, yi].

The cost of connecting two points [xi, yi] and [xj, yj] is the manhattan distance between them: |xi - xj| + |yi - yj|, where |val| denotes the absolute value of val.

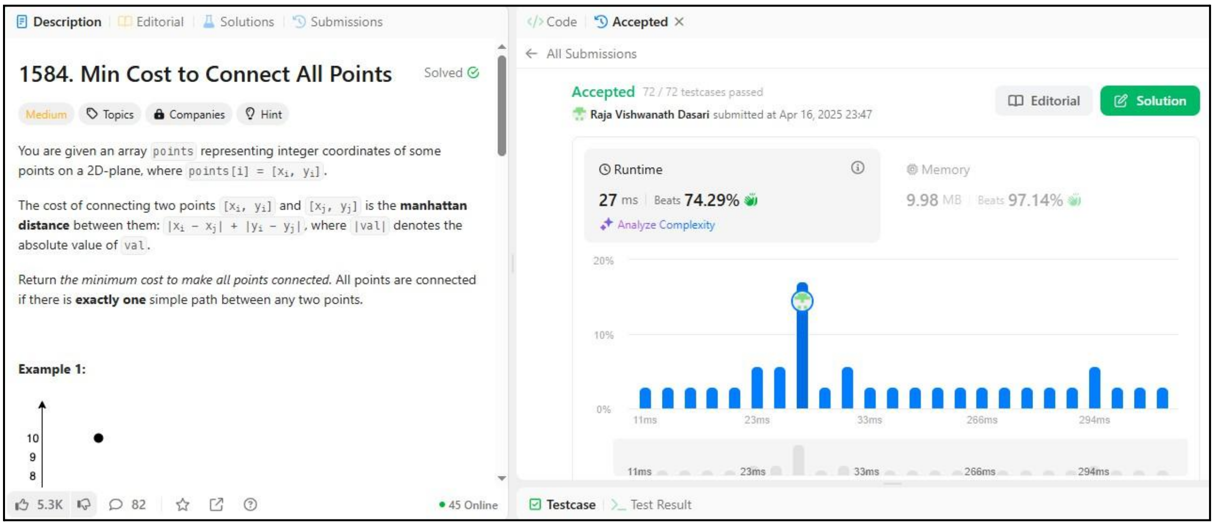
Return the minimum cost to make all points connected. All points are connected if there is exactly one simple path between any two points.

Submission Code:

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

#define MAX_POINTS 1000

int visited[MAX_POINTS];
int minDist[MAX_POINTS];

int absVal(int x) {
   return x < 0 ? -x : x;
}

int manhattan_distance(int* a, int* b) {
   return absVal(a[0] - b[0]) + absVal(a[1] - b[1]);
}

int minCostConnectPoints(int** points, int pointsSize, int* pointsColSize) {
   for (int i = 0; i < pointsSize; i++) {
      visited[i] = 0;
      minDist[i] = INT_MAX;
   }

   int totalCost = 0;
   minDist[0] = 0;

   for (int i = 0; i < pointsSize; i++) {
      int u = -1;
      for (int j = 0; j < pointsSize; j++) {
         if (!visited[j] && (u == -1 || minDist[j] < minDist[u])) {
            u = j;
```

```
            }
        }

        visited[u] = 1;
        totalCost += minDist[u];

        for (int v = 0; v < pointsSize; v++) {
            if (!visited[v]) {
                int dist = manhattan_distance(points[u], points[v]);
                if (dist < minDist[v]) {
                    minDist[v] = dist;
                }
            }
        }
    }

    return totalCost;
}
```

Result:



Lab Program 10

From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

<u>Code:</u>

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

#define MAX 100

int findMin(int dist[], int visited[], int V){
    int min = INT_MAX;
    int min_ind;
    for(int i=0; i<V;i++ ){
        if(visited[i]==0 &&  dist[i]<min){
            min = dist[i];
            min_ind = i;
        }
    }
    return min_ind;
}

void printPath(int prev[], int j) {
    if (prev[j] == -1) {
        printf("%d ", j);
        return;
    }
    printPath(prev, prev[j]);
    printf("-> %d ", j);
}

void djikstra(int graph[MAX][MAX],int src,int V){
    int dist[V];
    int visited[V];
    int prev[V];

    for(int i = 0;i<V;i++){
        dist[i]=INT_MAX;
        visited[i]=0;
```

```
        prev[i] = -1;
    }

    dist[src] = 0;

    for(int j = 0; j< V-1; j++){
        int u = findMin(dist, visited, V);
        visited[u] = 1;

        for(int v = 0; v<V ; v++){
            if(visited[v] == 0 && graph[u][v]!=0 && dist[u]!=INT_MAX){
                if(dist[u]+graph[u][v] < dist[v]){
                    dist[v] = dist[u]+graph[u][v];
                    prev[v] = u;
                }
            }
        }

    }

    printf("Vertex \t Shortest Distance\n");
    for(int i = 0; i<V;i++){
        printf("%d \t %d\t\t",i,dist[i]);
        printf("Shortest Path: ");
        printPath(prev, i);
        printf("\n");
    }

}

int main(){
    int V;
    int graph[MAX][MAX];
    int src;
    printf("Number of vertices: ");
    scanf("%d",&V);

    printf("Enter Weighted Adjacency Matrix ( 0 if no edge ):\n");
    for(int i = 0;i<V;i++){
        for(int j = 0; j<V; j++){
            scanf("%d",&graph[i][j]);
        }
    }
```

```
    printf("\nSource Vertex: ");
    scanf("%d",&src);

    djikstra(graph,src,V);

    return 0;

}
```

<u>Output:</u>

```
Number of vertices: 9
Enter Weighted Adjacency Matrix ( 0 if no edge ):
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 0 11 0
0 8 0 7 0 4 0 0 2
0 0 7 0 0 14 0 0 0
0 0 0 9 0 10 0 0 0
0 0 4 14 10 0 2 0 0
0 0 0 0 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0

Source Vertex: 0
Vertex   Shortest Distance
0        0              Shortest Path: 0
1        4              Shortest Path: 0 -> 1
2        12             Shortest Path: 0 -> 1 -> 2
3        19             Shortest Path: 0 -> 1 -> 2 -> 3
4        21             Shortest Path: 0 -> 7 -> 6 -> 5 -> 4
5        11             Shortest Path: 0 -> 7 -> 6 -> 5
6        9              Shortest Path: 0 -> 7 -> 6
7        8              Shortest Path: 0 -> 7
8        14             Shortest Path: 0 -> 1 -> 2 -> 8

Process returned 0 (0x0)   execution time : 5.754 s
Press any key to continue.
```

Lab Program 11

Implement "N-Queens Problem" using Backtracking.

<u>Code:</u>

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_N 10
void printBoard(int board[MAX_N][MAX_N], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", board[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
bool isSafe(int board[MAX_N][MAX_N], int row, int col, int n) {
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 1) {
            return false;
        }
        if (col - (row - i) >= 0 && board[i][col - (row - i)] == 1) {
            return false;
        }
        if (col + (row - i) < n && board[i][col + (row - i)] == 1) {
            return false;
        }
    }
    return true;
}
void solveNQueens(int board[MAX_N][MAX_N], int row, int n, int* solutionCount) {
    if (row == n) {
        (*solutionCount)++;
        printBoard(board, n);
        return;
    }

    for (int col = 0; col < n; col++) {
        if (isSafe(board, row, col, n)) {
            board[row][col] = 1;
```

```
            solveNQueens(board, row + 1, n, solutionCount);
            board[row][col] = 0;
        }
    }
}

int main() {
    int n;
    printf("Enter the size of the board (n): ");
    scanf("%d", &n);

    if (n < 1 || n > MAX_N) {
        printf("Please enter a number between 1 and %d.\n", MAX_N);
        return 1;
    }

    int board[MAX_N][MAX_N] = {0};
    int solutionCount = 0;

    solveNQueens(board, 0, n, &solutionCount);

    if (solutionCount == 0) {
        printf("No solution exists.\n");
    } else {
        printf("Total solutions found: %d\n", solutionCount);
    }

    return 0;
}
```

Output:

```
Enter the size of the board (n): 4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Total solutions found: 2
```