VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

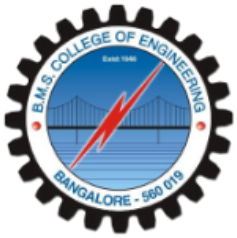
LAB REPORT
On

ANALYSIS AND DESIGN OF ALGORITHMS (23CS4PCADA)

Submitted by

RIDHIMA SUHANE
(1BM23CS266)

in partial fulfilment 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 RIDHIMA SUHANE **(1BM23CS266)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfilment 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. Shyamala G
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

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.

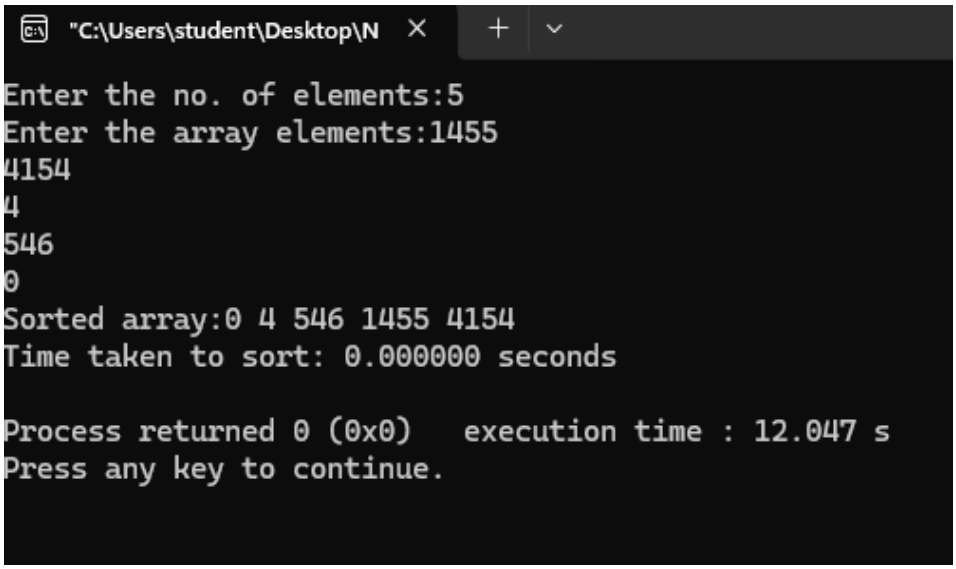
Lab program 1

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

```
#include<stdio.h>
#include<time.h>
int a[20],n;
void simple_sort(int [],int,int,int);
void merge_sort(int[],int,int);
int main()
{
   int i;
   clock_t start, end;
   double time_taken;
   printf("Enter the no. of elements:");
   scanf("%d", &n);
   printf("Enter the array elements:");
   for (i = 0; i < n; i++)
   {
         scanf("%d", &a[i]);
   }
   start = clock();
   merge_sort(a, 0, n - 1);
   end = clock();
   time_taken = (double)(end - start) / CLOCKS_PER_SEC;
   printf("Sorted array:");
   for (i = 0; i < n; i++)
   {
      printf("%d ", a[i]);
   }
   printf("\n");
   printf("Time taken to sort: %f seconds\n", time_taken);
   return 0;
}
void merge_sort(int a[],int low, int high)
{
   if(low<high)
   {
      int mid=(low+high)/2;
      merge_sort(a,low,mid);
      merge_sort(a,mid+1,high);
      simple_sort(a,low,mid,high);
   }
}
void simple_sort(int a[],int low, int mid, int high)
{
   int i=low,j=mid+1,k=low;
   int c[n];
   while(i<=mid && j<=high)

{
      if(a[i]<a[j])
      {
         c[k++]=a[i];
         i++;
      }
```

```
        else
        {
            c[k++]=a[j];
            j++;
        }
    }
    while(i<=mid)
    {
        c[k++]=a[i];
        i++;
    }
    while(j<=high)
    {
        c[k++]=a[j];
        j++;
    }
    for(i=low;i<=high;i++)
    {
        a[i]=c[i];
    }
}
```

OUTPUT-



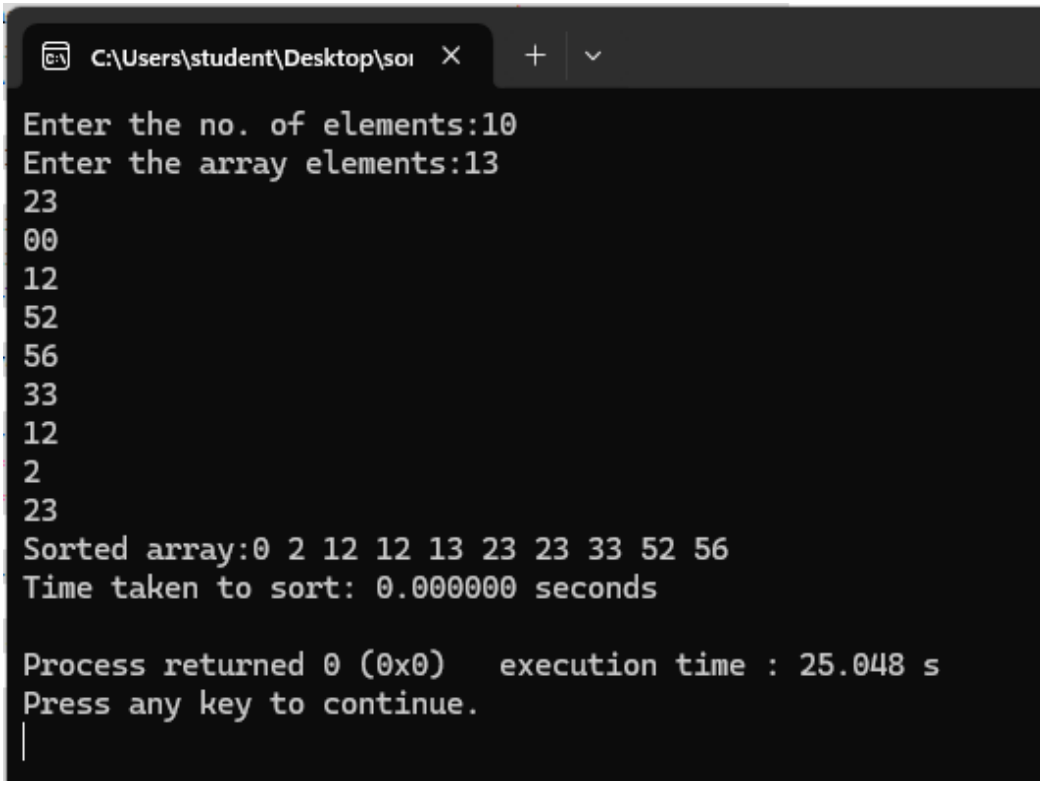
Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

```
#include <stdio.h>
#include<time.h>
int a[20],n;
int partition(int [],int, int);
void quick_sort(int [],int,int);
void swap(int*,int*);
int main()
{
    int i;
    clock_t start, end;
```

```
    double time_taken;
    printf("Enter the no. of elements:");
    scanf("%d", &n);
    printf("Enter the array elements:");
    for (i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    start = clock();
    quick_sort(a, 0, n - 1);
    end = clock();
    time_taken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Sorted array:");
    for (i = 0; i < n; i++)
    {
      printf("%d ", a[i]);
    }
    printf("\n");
    printf("Time taken to sort: %f seconds\n", time_taken);
    return 0;
}
void swap(int *a,int *b)
{
    int temp=*a;
    *a=*b;
    *b=temp;
}
void quick_sort(int a[],int low,int high)
{
    if(low<high)
    {
        int mid=partition(a,low,high);
        quick_sort(a,low,mid-1);
        quick_sort(a,mid+1,high);
    }
}

int partition(int a[],int low,int high)
{
    int pivot=a[low];
    int i=low;
    int j=high+1;
    while(i<=j){
        do
        {
            i=i+1;
        }
    while(a[i]<pivot && i<=high);
    do
    {
        j=j-1;
    }
    while(a[j]>pivot && j>=low);
    if(i<j)
    {
        swap(&a[i],&a[j]);
    }
}
swap(&a[j],&a[low]);
```

```
return j;
}
```

OUTPUT-



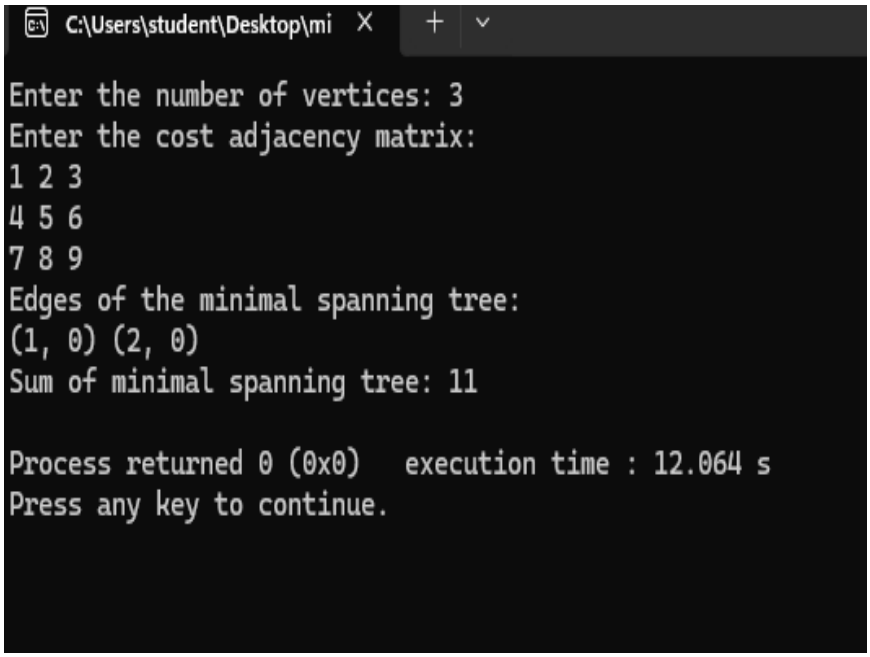
Lab program 2

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

```
#include <stdio.h>
int cost[10][10], n, t[10][2], sum;
void prims(int cost[10][10], int n);
int main()
{
   int i, j;
   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   printf("Enter the cost adjacency matrix:\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < n; j++)
      {
         scanf("%d", &cost[i][j]);
      }
   }
   prims(cost, n);
   printf("Edges of the minimal spanning tree:\n");
   for (i = 0; i < n - 1; i++)
   {
      printf("(%d, %d) ", t[i][0], t[i][1]);
   }
   printf("\nSum of minimal spanning tree: %d\n", sum);
   return 0;
}
void prims(int cost[10][10], int n)
{
   int i, j, u, v;
   int min, source;
   int p[10], d[10], s[10];
   min = 999;
   source = 0;
   for (i = 0; i < n; i++)
   {
      d[i] = cost[source][i];
      s[i] = 0;
      p[i] = source;
   }
   s[source] = 1;
   sum = 0;
   int k = 0;
   for (i = 0; i < n - 1; i++)
   {
      min = 999;
      u = -1;
      for (j = 0; j < n; j++)
      {
         if (s[j] == 0 && d[j] < min)
         {
            min = d[j];
            u = j;
         }
      }
```

```
        if (u != -1)
        {
            t[k][0] = u;
            t[k][1] = p[u];
            k++;
            sum += cost[u][p[u]];
            s[u] = 1;
            for (v = 0; v < n; v++)
            {
                if (s[v] == 0 && cost[u][v] < d[v])
                {
                    d[v] = cost[u][v];
                    p[v] = u;
                }
            }
        }
    }
}
```

OUTPUT-



Lab program 3

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

```
#include <stdio.h>
int cost[10][10], n, t[10][2], sum;
void kruskal(int cost[10][10], int n);
int find(int parent[10], int i);
int main()
{
   int i, j;
   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   printf("Enter the cost adjacency matrix:\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < n; j++)
      {
         scanf("%d", &cost[i][j]);
      }
   }
   kruskal(cost, n);
   printf("Edges of the minimal spanning tree:\n");
   for (i = 0; i < n - 1; i++)
   {
      printf("(%d, %d) ", t[i][0], t[i][1]);
   }
   printf("\nSum of minimal spanning tree: %d\n", sum);
   return 0;
}
void kruskal(int cost[10][10], int n)
{
   int min, u, v, count, k;
   int parent[10];
   k = 0;
   sum = 0;
   for (int i = 0; i < n; i++)
   {
      parent[i] = i;
   }
   count = 0;
   while (count < n - 1)
   {
      min = 999;
      u = -1;
      v = -1;
      for (int i = 0; i < n; i++)
      {
         for (int j = 0; j < n; j++)
         {
            if (find(parent, i) != find(parent, j) && cost[i][j] < min)
            {
               min = cost[i][j];
               u = i;
               v = j;
            }
```

```
            }
        }
        int root_u = find(parent, u);
        int root_v = find(parent, v);
        if (root_u != root_v)
        {
                parent[root_u] = root_v;
                t[k][0] = u;
                t[k][1] = v;
                sum += min;
                k++;
                count++;
        }
    }
}
int find(int parent[10], int i)
{
    while (parent[i] != i)
    {
        i = parent[i];
    }
    return i;
}
```

OUTPUT-

```
Enter the number of vertices: 4
Enter the cost adjacency matrix:
1 2 3 4
5 6 7 8
4 3 2 1
8 7 6 5
Edges of the minimal spanning tree:
(2, 3) (0, 1) (0, 2)
Sum of minimal spanning tree: 6

Process returned 0 (0x0)   execution time : 20.104 s
Press any key to continue.
```

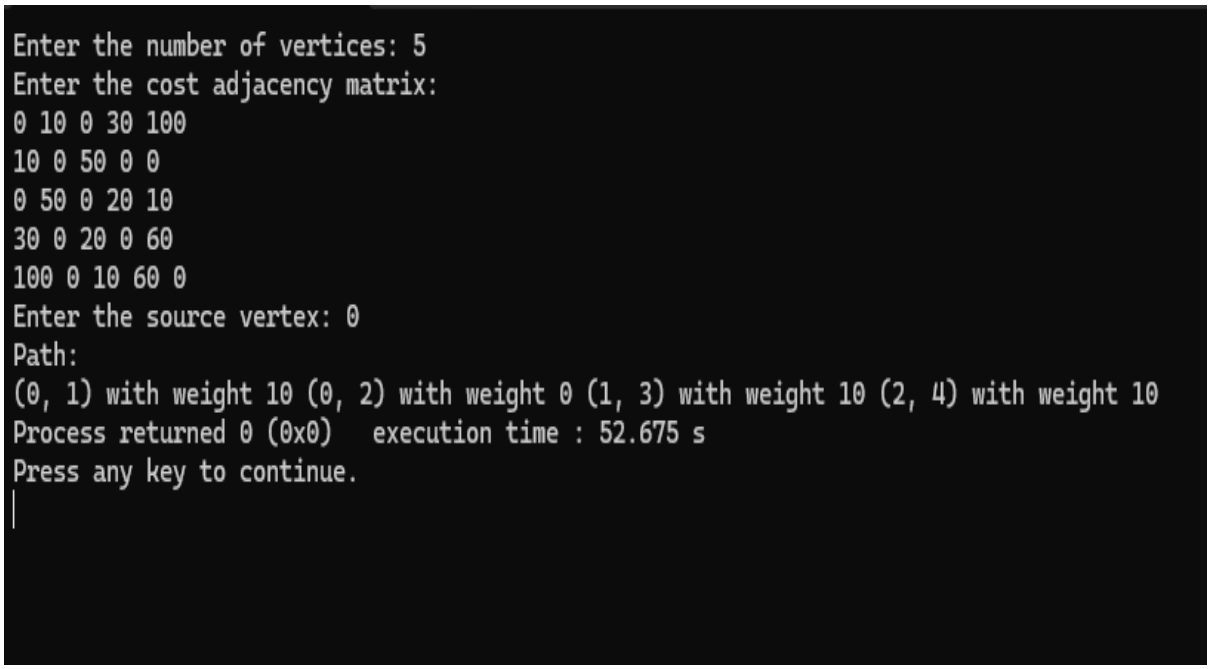
Lab program 4

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

```
#include <stdio.h>
int cost[10][10], n, result[10][2], weight[10];
void dijkstras(int [][10], int );
int main()
{
   int i, j, s;
   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   printf("Enter the cost adjacency matrix:\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < n; j++)
      {
         scanf("%d", &cost[i][j]);
      }
   }
   printf("Enter the source vertex: ");
   scanf("%d", &s);
   dijkstras(cost, s);
   printf("Path:\n");
   for (i = 1; i < n; i++)
   {
      printf("(%d, %d) with weight %d ", result[i][0], result[i][1], weight[result[i][1]]);
   }
   return 0;
}
void dijkstras(int cost[][10], int s)
{
   int d[10], p[10], visited[10];
   int i, j, min, u, v, k;
   for(i = 0; i < 10; i++)
   {
      d[i] = 999;
      visited[i] = 0;
      p[i] = s;
   }
   d[s] = 0;
   visited[s] = 1;
   for(i = 0; i < n; i++)
   {
      min = 999;
      u = 0;
      for(j = 0; j < n; j++)
      {
         if(visited[j] == 0)
         {
            if(d[j] < min)
            {
               min = d[j];
               u = j;
            }
```

```
            }
        }
        visited[u] = 1;
        for(v = 0; v < n; v++)
        {
            if(visited[v] == 0 && (d[u] + cost[u][v] < d[v]))
            {
                d[v] = d[u] + cost[u][v];
                p[v] = u;
            }
        }
    }
    for(i = 0; i < n; i++)
    {
        result[i][0] = p[i];
        result[i][1] = i;
        weight[i] = d[i];
    }
}
```

OUTPUT-



Lab program 5

Leetcode (minimum height trees and maximum binary trees)

310. Minimum Height Trees

Medium | Topics | Companies | Hint

A tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of `n` nodes labelled from `0` to `n - 1`, and an array of `n - 1` `edges` where `edges[i] = [a_i, b_i]` indicates that there is an undirected edge between the two nodes `a_i` and `b_i` in the tree, you can choose any node of the tree as the root. When you select a node `x` as the root, the result tree has height `h`. Among all possible rooted trees, those with minimum height (i.e. `min(h)`) are called **minimum height trees** (MHTs).

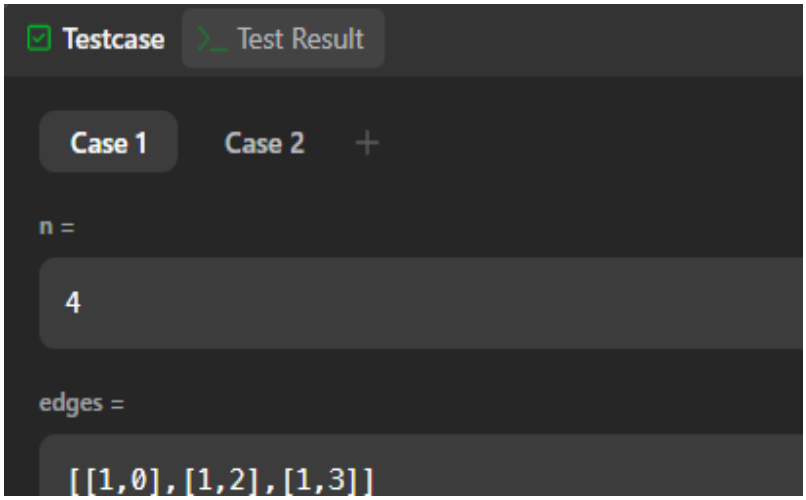
Return *a list of all **MHTs'** root labels*. You can return the answer in **any order**.

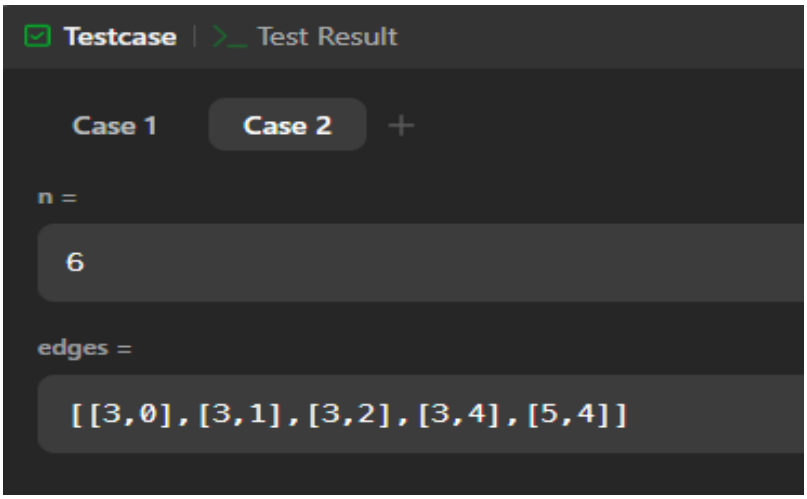
The **height** of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

```
#include <stdlib.h>
#define MAX_N 20000

int* findMinHeightTrees(int n, int** edges, int edgesSize, int* edgesColSize, int* returnSize) {
    if (n == 1) {
        int* res = (int*)malloc(sizeof(int));
        res[0] = 0;
        *returnSize = 1;
        return res;
    }

    int* graph[MAX_N];
    int graphSize[MAX_N] = {0};
    int degree[MAX_N] = {0};

    for (int i = 0; i < n; i++) {
        graph[i] = (int*)malloc(sizeof(int) * MAX_N);
    }

    for (int i = 0; i < edgesSize; i++) {
        int u = edges[i][0];
        int v = edges[i][1];
        graph[u][graphSize[u]++] = v;
        graph[v][graphSize[v]++] = u;
        degree[u]++;
        degree[v]++;
    }

    int* leaves = (int*)malloc(sizeof(int) * n);
    int front = 0, back = 0;
    for (int i = 0; i < n; i++) {
        if (degree[i] == 1) {
            leaves[back++] = i;
        }
    }
    int remainingNodes = n;
    while (remainingNodes > 2) {
        int leavesCount = back - front;
        remainingNodes -= leavesCount;
        for (int i = 0; i < leavesCount; i++) {
```

```
            int leaf = leaves[front++];
            for (int j = 0; j < graphSize[leaf]; j++) {
                int neighbor = graph[leaf][j];
                degree[neighbor]--;
                if (degree[neighbor] == 1) {
                    leaves[back++] = neighbor;
                }
            }
        }
    }
    *returnSize = back - front;
    int* result = (int*)malloc(sizeof(int) * (*returnSize));
    for (int i = 0; i < *returnSize; i++) {
        result[i] = leaves[front++];
    }
    for (int i = 0; i < n; i++) {
        free(graph[i]);
    }
    free(leaves);
    return result;
}
```

OUTPUT-





654. Maximum Binary Tree

Medium | Topics | Companies

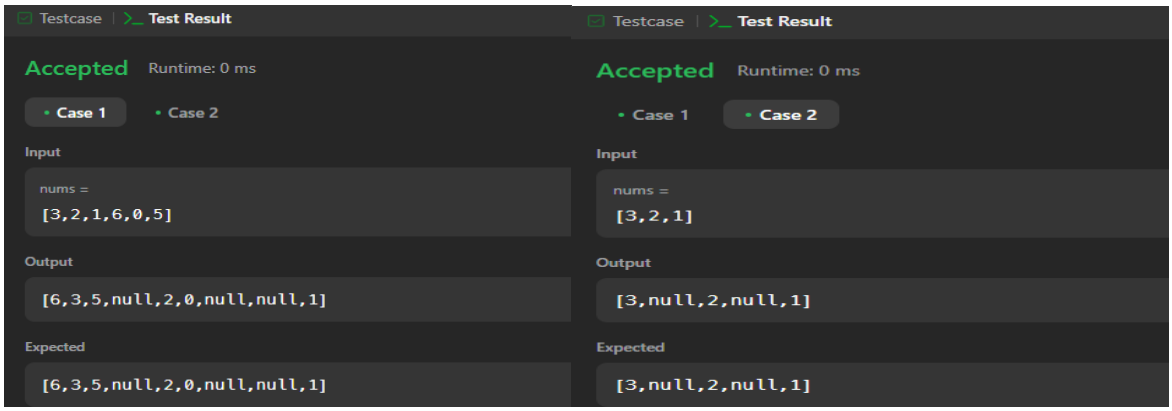
You are given an integer array `nums` with no duplicates. A **maximum binary tree** can be built recursively from `nums` using the following algorithm:

1. Create a root node whose value is the maximum value in `nums`.
2. Recursively build the left subtree on the **subarray prefix** to the **left** of the maximum value.
3. Recursively build the right subtree on the **subarray suffix** to the **right** of the maximum value.

Return *the **maximum binary tree** built from* `nums`.

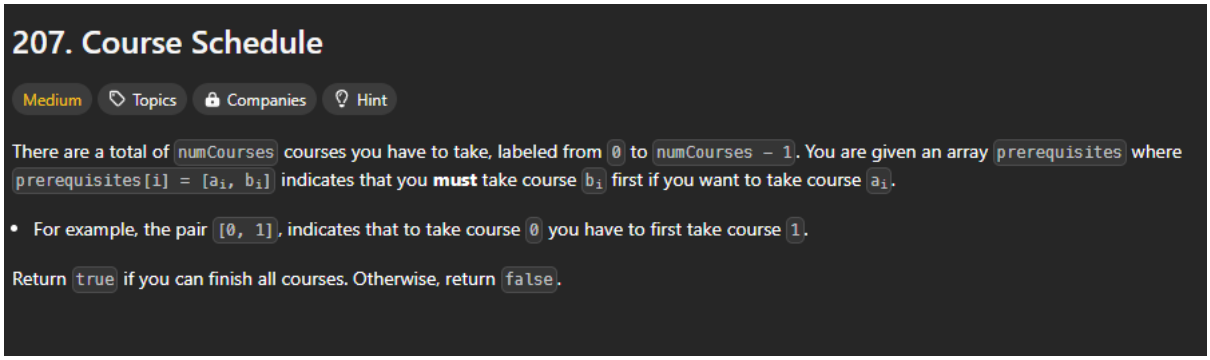
```
struct TreeNode* constructMaximumBinaryTree(int* nums, int numsSize){
    int max = -1, idx = -1;
    for (int i = 0; i < numsSize; i++){
        if (nums[i] > max){
            max = nums[i];
            idx = i;
        }
    }
    int size1 = idx;
    int size2 = numsSize - idx - 1;
    int * pref = nums;
    int * suff = nums + idx + 1;
    struct TreeNode * root = malloc(sizeof(struct TreeNode));
    root->val = max;
    if (size1 != 0){
        root->left = constructMaximumBinaryTree(pref, size1);
    }
    else root->left = NULL;
    if (size2 != 0){
        root->right = constructMaximumBinaryTree(suff, size2);
    }
    else root->right = NULL;
    return root;
}
```

OUTPUT-



Lab program 6

Leetcode (Topological sorting)



207. Course Schedule

Medium | Topics | Companies | Hint

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`.

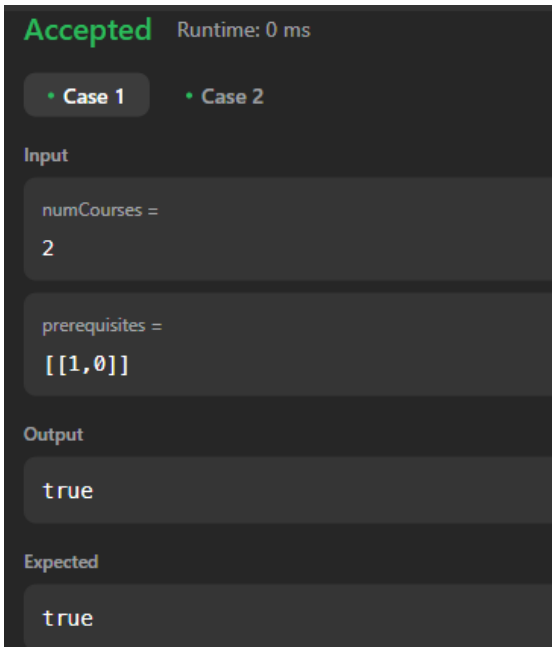
- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

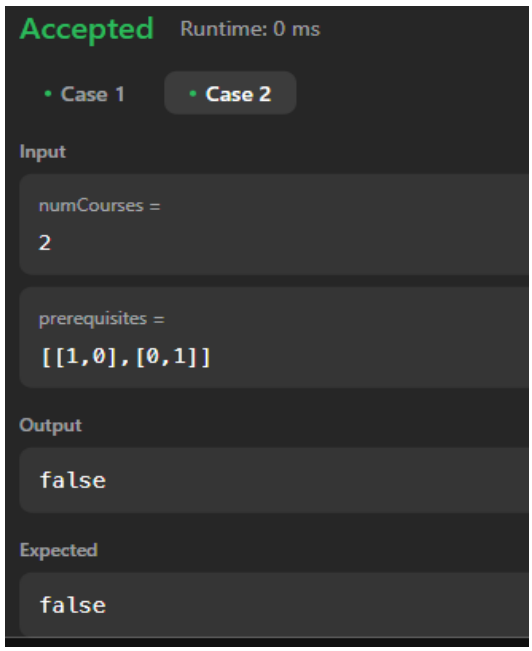
Return `true` if you can finish all courses. Otherwise, return `false`.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_COURSES 10000

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

    for (int i = 0; i < numCourses; i++) {
        graph[i] = (int*)malloc(MAX_COURSES * sizeof(int))
    }
    for (int i = 0; i < prerequisitesSize; i++) {
        int course = prerequisites[i][0];
        int prereq = prerequisites[i][1];
        graph[prereq][graphSize[prereq]++] = course;
        inDegree[course]++;
    }
    int* queue = (int*)malloc(numCourses * sizeof(int));
    int front = 0, rear = 0;
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            queue[rear++] = i;
        }
    }
    int count = 0;
    while (front < rear) {
        int course = queue[front++];
        count++;
        for (int i = 0; i < graphSize[course]; i++) {
            int neighbor = graph[course][i];
            inDegree[neighbor]--;
            if (inDegree[neighbor] == 0) {
                queue[rear++] = neighbor;
            }
```

```
        }
    }

    for (int i = 0; i < numCourses; i++) {
        free(graph[i]);
    }
    free(graph);
    free(graphSize);
    free(inDegree);
    free(queue);

    return count == numCourses;
}
```

OUTPUT-

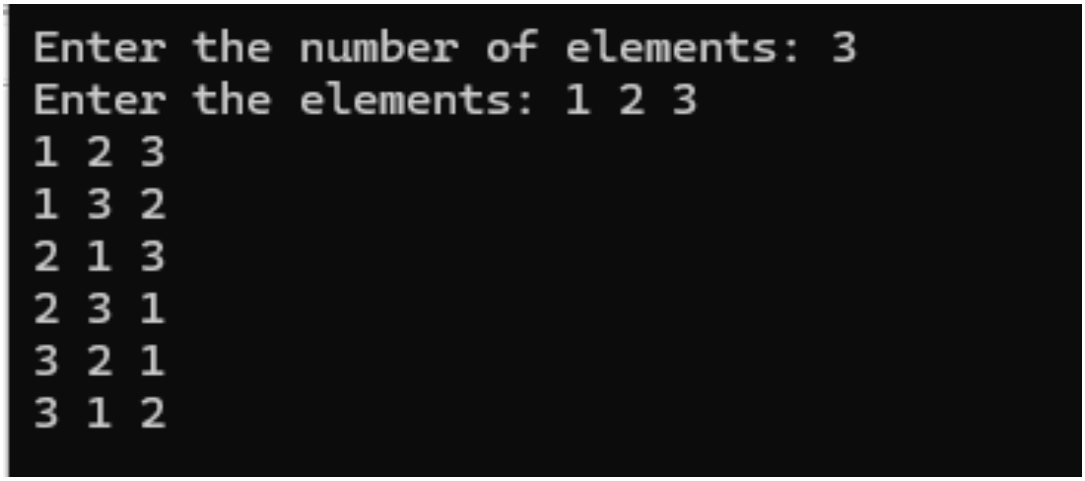




Implement Johnson Trotter algorithm to generate permutations.

```
#include <stdio.h>
#include <stdlib.h>
void swap(int* a, int* b) {
int temp = *a;
*a = *b;
*b = temp;
}
void generatePermutations(int arr[], int start, int end) {
if (start == end) {
for (int i = 0; i <= end; i++) {
printf("%d ", arr[i]);
}
printf("\n");
} else {
for (int i = start; i <= end; i++) {
swap(&arr[start], &arr[i]);
generatePermutations(arr, start + 1, end);
swap(&arr[start], &arr[i]); }
}
}
int main() {
int n;
printf("Enter the number of elements: ");
scanf("%d", &n);
int* arr = (int*)malloc(n * sizeof(int));
printf("Enter the elements: ");
for (int i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}
generatePermutations(arr, 0, n - 1);
free(arr);
return 0;
}
```

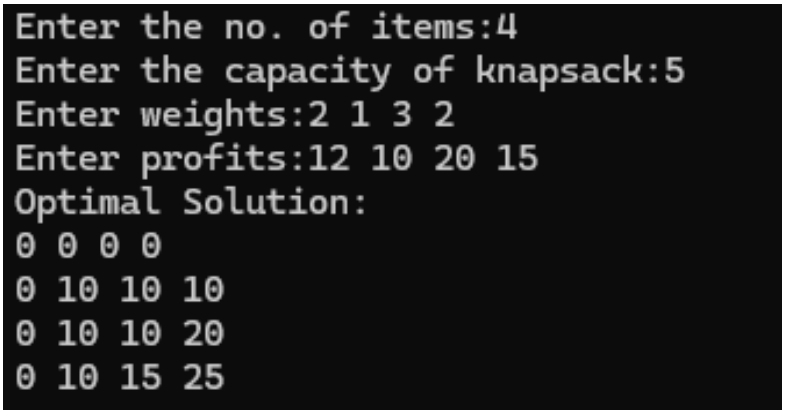
OUTPUT-



Implement 0/1 Knapsack problem using dynamic programming.

```
#include <stdio.h>
int n,m,w[10],p[10],v[10][10];
void knapsack(int,int,int[],int[]);
int max(int,int);
int main()
{
int i,j;
printf("Enter the no. of items:");
scanf("%d",&n);
printf("Enter the capacity of knapsack:");
scanf("%d",&m);
printf("Enter weights:");
for(i=0;i<n;i++){
scanf("%d",&w[i]);
}
printf("Enter profits:");
for(i=0;i<n;i++){
scanf("%d",&p[i]);
}
knapsack(n,m,w,p);
printf("Optimal Solution:\n");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("%d ",v[i][j]);
}
printf("\n");
}
return 0;
}
void knapsack(int n, int m, int w[],int p[]){
int i,j;
for(i=0;i<n;i++){
for(j=0;j<m;j++){
if(i==0 || j==0){
v[i][j]=0;
}else if(w[i]>j){
v[i][j]=v[i-1][j];
}else{
v[i][j]=max(v[i-1][j],((v[i-1][j-w[i]])+p[i]));
}
}
}
}
int max(int a,int b){
if(a>b){
return a;
}else{
return b;
}
}
```

OUTPUT-


Enter the no. of items:4
Enter the capacity of knapsack:5
Enter weights:2 1 3 2
Enter profits:12 10 20 15
Optimal Solution:
0 0 0 0
0 10 10 10
0 10 10 20
0 10 15 25

Lab program 7

Implement fractional Knapsack problem using Greedy technique.

```
#include <stdio.h>
void knapsack(int n, int p[], int w[], int W) {
   int used[n];
   for (int i = 0; i < n; ++i)
      used[i] = 0;

   int cur_w = W;
   float tot_v = 0.0;
   int i, maxi;

   while (cur_w > 0) {
      maxi = -1;
      for (i = 0; i < n; ++i) {
         if ((used[i] == 0) &&
            ((maxi == -1) || ((float)w[i] / p[i] > (float)w[maxi] / p[maxi]))) {
            maxi = i;
         }
      }

      used[maxi] = 1;

      if (w[maxi] <= cur_w) {
         cur_w -= w[maxi];
         tot_v += p[maxi];
         printf("Added object %d (%d, %d) completely in the bag. Space left: %d.\n",
               maxi + 1, w[maxi], p[maxi], cur_w);
      } else {
         int taken = cur_w;
         cur_w = 0;
         tot_v += (float)taken / w[maxi] * p[maxi];
         printf("Added %d%% (%d, %d) of object %d in the bag.\n",
               (int)((float)taken / w[maxi] * 100), w[maxi], p[maxi], maxi + 1);
      }
   }

   printf("Filled the bag with objects worth %.2f.\n", tot_v);
}

int main() {
   int n, W;

   printf("Enter the number of objects: ");
   scanf("%d", &n);

   int p[n], w[n];

   printf("Enter the profits of the objects: ");
   for (int i = 0; i < n; i++) {
      scanf("%d", &p[i]);
   }

   printf("Enter the weights of the objects: ");
```

```
    for (int i = 0; i < n; i++) {
        scanf("%d", &w[i]);
    }

    printf("Enter the maximum weight of the bag: ");
    scanf("%d", &W);

    knapsack(n, p, w, W);
    return 0;
}
```

OUTPUT-

```
Enter the number of objects: 7
Enter the profits of the objects: 5 10 15 7 8 9 4
Enter the weights of the objects: 1 3 5 4 1 3 2
Enter the maximum weight of the bag: 15
Added object 4 (4, 7) completely in the bag. Space left: 11.
Added object 7 (2, 4) completely in the bag. Space left: 9.
Added object 3 (5, 15) completely in the bag. Space left: 4.
Added object 6 (3, 9) completely in the bag. Space left: 1.
Added 33% (3, 10) of object 2 in the bag.
Filled the bag with objects worth 38.33.

Process returned 0 (0x0)   execution time : 72.736 s
Press any key to continue.
```

Leetcode

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

Medium | Topics | Companies | Hint

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

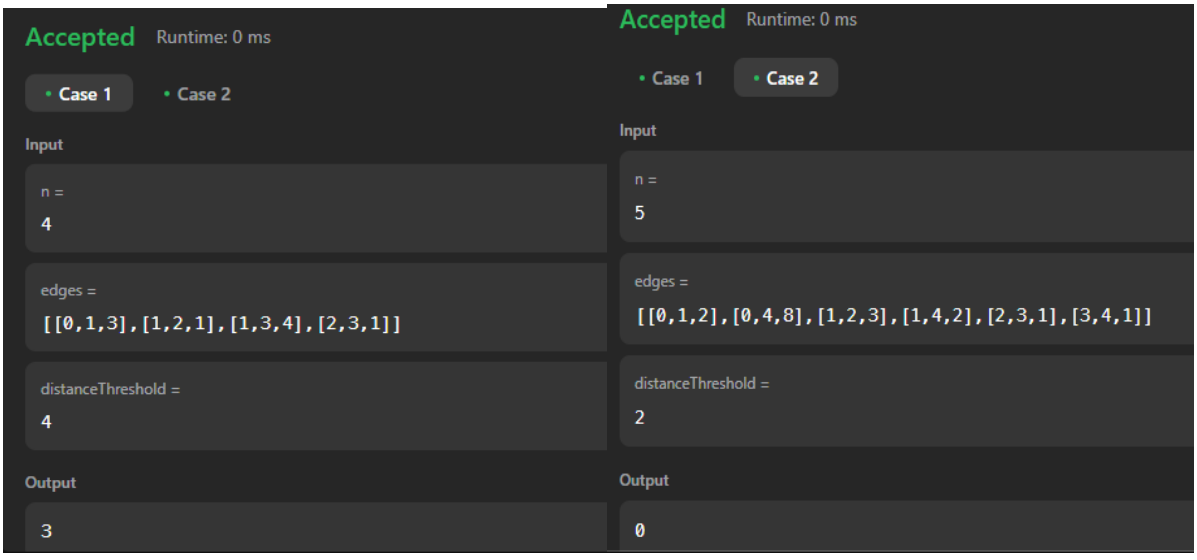
Return the city with the smallest number of cities that are reachable through some path and whose distance is **at most** `distanceThreshold`, If there are multiple such cities, return the city with the greatest number.

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

```
int findTheCity(int n, int** edges, int edgesSize, int* edgesColSize, int distanceThreshold) {
    short i,j,k,**dist=(short**)malloc(sizeof(short*)*n);

    for(i=0;i<n;i++){
        dist[i]=(short*)malloc(2*n);
        for(j=0;j<n;j++)
```

```
            dist[i][j]=i==j?0:SHRT_MAX;
    }

    for(i=0;i<edgesSize;i++){
        dist[edges[i][0]][edges[i][1]]=edges[i][2];
        dist[edges[i][1]][edges[i][0]]=edges[i][2];
    }

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            for(k=0;k<n;k++)
                dist[j][k]=dist[j][k]<dist[j][i]+dist[i][k]?dist[j][k]:dist[j][i]+dist[i][k];

    short *reached=(short*)malloc(2*n);
    for(i=0;i<n;i++){
        reached[i]=0;
        for(j=0;j<n;j++)
            dist[i][j]<=distanceThreshold?reached[i]++:1;
    }
    for(i=0;i<n;free(dist[i++]));
    free(dist);

    short res=n-1,min=reached[n-1];
    for(i=n-1;i>=0;i--)
        if(reached[i]<min){
            min=reached[i];
            res=i;
        }
    free(reached);

    return res;
}
```

OUTPUT –



Lab program 8

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

```
#include<stdio.h>
int a[10], n;
void heapify(int[], int);
int main() {
   printf("Enter the number of array elements:");
   scanf("%d", &n);

   int i;
   printf("Enter array elements:");
   for(i = 0; i < n; i++) {
      scanf("%d", &a[i]);
   }

   heapify(a, n);

   printf("Array elements:");
   for(i = 0; i < n; i++) {
      printf(" %d", a[i]);
   }

   return 0;
}

void heapify(int a[], int n) {
   int k;
   for(k = 1; k < n; k++) {
      int key = a[k];
      int c = k;
      int p = (c - 1) / 2;
      while(c > 0 && key > a[p]) {
         a[c] = a[p];
         c = p;
         p = (c - 1) / 2;
      }
      a[c] = key;
   }
}
```

OUTPUT:

```
Enter the number of array elements:7
Enter array elements:50 25 30 75 100 45 80
Array elements: 100 75 80 25 50 30 45
Process returned 0 (0x0)   execution time : 66.658 s
```

Implement “N-Queens Problem” using Backtracking.

```
#include <stdio.h>
#include <stdbool.h>
bool place(int[], int);
void printSolution(int[], int);
void nQueens(int);

int main() {
   int n;
   printf("Enter the number of queens: ");
   scanf("%d", &n);
   nQueens(n);
   return 0;
}

void nQueens(int n) {
   int x[10];
   int count = 0;
   int k = 1;

   while (k != 0) {
      x[k] = x[k] + 1;
      while (x[k] <= n && !place(x, k)) {
         x[k] = x[k] + 1;
      }

      if (x[k] <= n) {
         if (k == n) {
            printSolution(x, n);
            printf("Solution found\n");
            count++;
         } else {
            k++;
            x[k] = 0;
         }
      } else {
         k--;
      }
   }
   printf("Total solutions: %d\n", count);
}

bool place(int x[10], int k) {
   int i;
   for (i = 1; i < k; i++) {
      if ((x[i] == x[k]) || (i - x[i] == k - x[k]) || (i + x[i] == k + x[k])) {
         return false;
      }
   }
   return true;
}

void printSolution(int x[10], int n) {
   int i;
   for (i = 1; i <= n; i++) {
      printf("%d ", x[i]);
```

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

OUTPUT: