

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

On

ANALYSIS AND DESIGN OF ALGORITHMS (23CS4PCADA)

Submitted by

JAYASHREE TARAI (1BM24CS407)

**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 **JAYASHREE TARAI (1BM24CS407)**, 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.

RAMYA K M
Assistant 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	Write program to obtain the Topological ordering of vertices in a given digraph. LeetCode Program related to Topological sorting	5
2	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. LeetCode Program related to sorting.	9
3	Sort a given set of N integer elements using Quick Sort technique and compute its time taken. LeetCode Program related to sorting.	13
4	Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.	16
5	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.	22
6	Implement Johnson Trotter algorithm to generate permutations.	25
7	Implement Fractional Knapsack using Greedy technique. LeetCode Program related to Greedy Technique algorithms.	29
8	Implement 0/1 Knapsack problem using dynamic programming. LeetCode Program related to Knapsack problem or Dynamic Programming.	32
9	Sort a given set of N integer elements using Heap Sort technique and compute its time taken.	36
10	Implement All Pair Shortest paths problem using Floyd's algorithm. LeetCode Program related to shortest distance calculation.	38
11	Implement "N-Queens Problem" using Backtracking.	41

GITHUB LINK :<https://github.com/Jayashreecse/ADA-LAB/tree/main>

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 1.1:

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

Program full details

Code

```
#include <stdio.h>

#include <stdbool.h>

#define MAX 100

int graph[MAX][MAX];

bool visited[MAX];

int stack[MAX];

int top = -1;

int n;

void push(int v) {
    stack[++top] = v;
}

void dfs(int node) {
    visited[node] = true;
    for (int i = 0; i < n; i++) {
        if (graph[node][i] == 1 && !visited[i]) {
            dfs(i);
        }
    }
}
```

```
    push(node);  
}
```

```
void topologicalSort() {
```

```
    for (int i = 0; i < n; i++) {  
        visited[i] = false;  
    }
```

```
    for (int i = 0; i < n; i++) {  
        if (!visited[i]) {  
            dfs(i);  
        }  
    }
```

```
    printf("Topological Order: ");
```

```
    while (top != -1) {  
        printf("%d ", stack[top--]);  
    }
```

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

```
}
```

```
int main() {
```

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

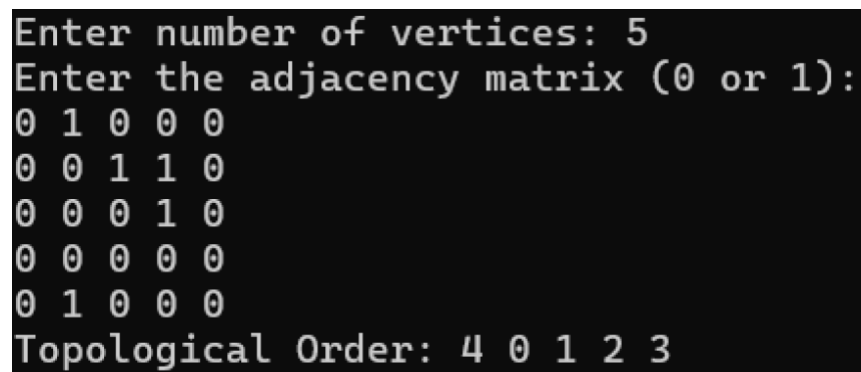
```
    scanf("%d", &n);
```

```
    printf("Enter the adjacency matrix (0 or 1):\n");
```

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

```
        scanf("%d", &graph[i][j]);  
    }  
}  
topologicalSort();  
return 0;  
}
```

Screenshot of Output



```
Enter number of vertices: 5  
Enter the adjacency matrix (0 or 1):  
0 1 0 0 0  
0 0 1 1 0  
0 0 0 1 0  
0 0 0 0 0  
0 1 0 0 0  
Topological Order: 4 0 1 2 3
```

4/4/25

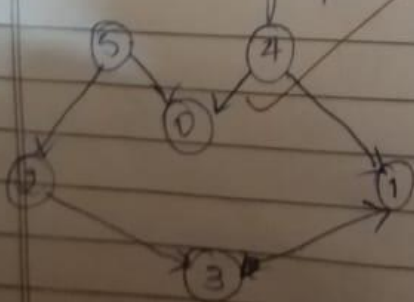
LAB-3

Topological sorting of vertices in a given digraph. Algorithm and step wise explanation.

Algorithm

- Step 1. Create a graph with n vertices and m -directed edges
- Step 2. Initialize a stack and a visited array of size n .
- Step 3. for each unvisited vertex in the graph, do the following:
 - Call the dfs function with the vertex as the parameter.
 - In the dfs function, mark the vertex as visited and recursively call the dfs function for all unvisited neighbors of the vertex.
 - Once all the neighbors have been visited, push the vertex onto the stack.
- Step 4. After all, vertices have been visited, pop elements from the stack and append them to the output list until the stack is empty.
- Step 5. the resulting list is the topologically sorted order of the graph.

ex: Consider graph



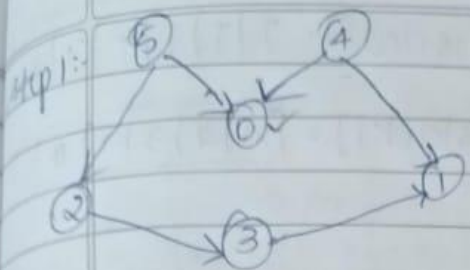
visited[] =

0	1	2	3	4	5
F	F	F	F	F	F

stack[] =

--	--	--	--	--	--

↑_{top}

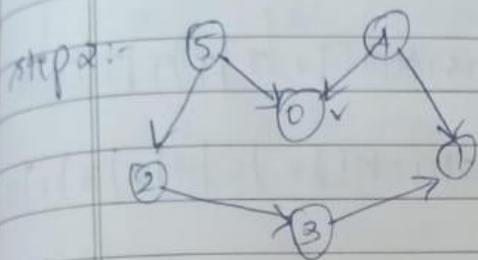


visited[] =

0	1	2	3	4	5
T	F	F	F	F	F

stack[] =

--	--	--	--	--	--

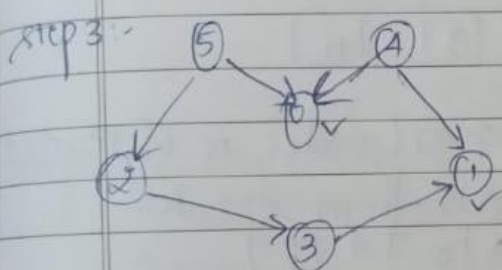


visited[] =

0	1	2	3	4	5
T	F	F	F	F	F

stack[] =

0					
---	--	--	--	--	--

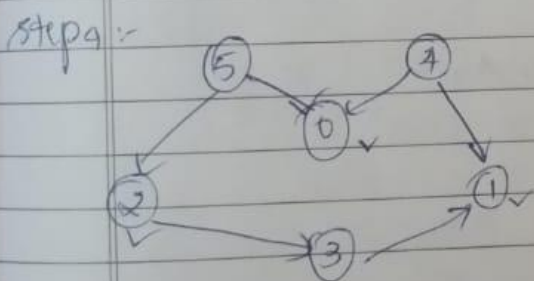


visited[] =

0	1	2	3	4	5
T	T	F	F	F	F

stack[] =

1	0				
---	---	--	--	--	--

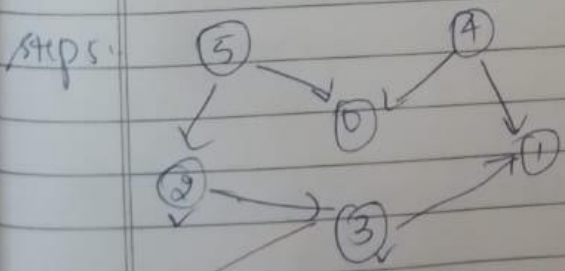


visited[] =

0	1	2	3	4	5
T	T	T	F	F	F

stack[] =

1	0	1			
---	---	---	--	--	--

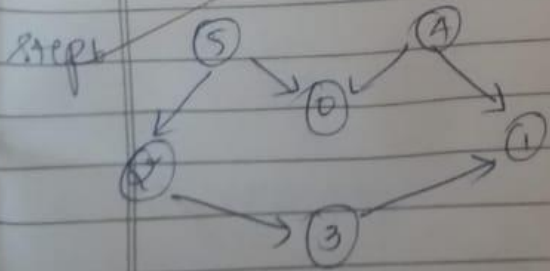


visited[] =

0	1	2	3	4	5
T	T	T	T	F	F

stack[] =

1	0	1	1		
---	---	---	---	--	--



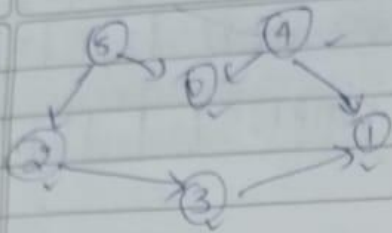
visited[] =

0	1	2	3	4	5
T	T	T	T	T	F

stack[] =

2	3	1	0	1	
---	---	---	---	---	--

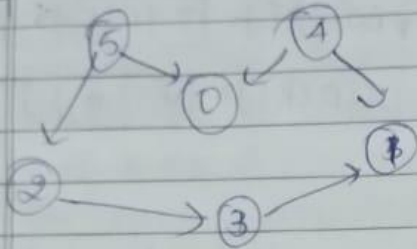
step 7



visited[] = T | T | T | T | T | F

stack[] = 4 | 2 | 3 | 1 | 0 |

step 7



visited[] = T | T | T | T | T | T

stack[] = 5 | 4 | 2 | 3 | 1 | 0

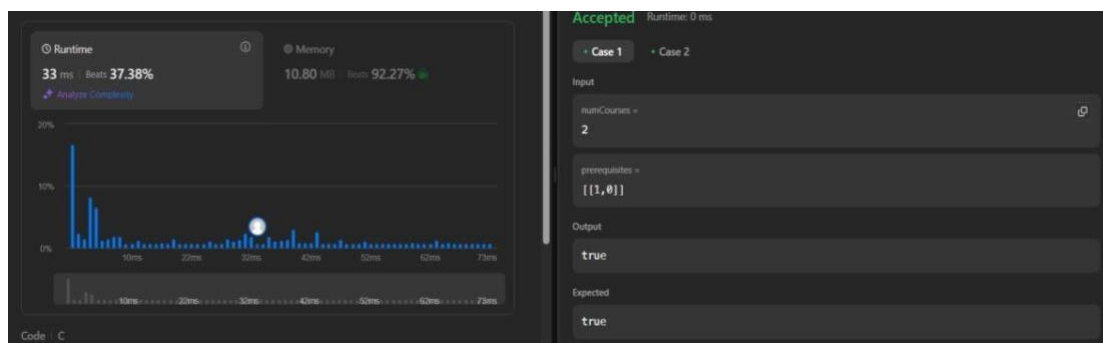
∴ in stack 5 | 4 | 2 | 3 | 1 | 0

∴ the final topological order is obtained by popping elements from stack
ie. 0 | 1 | 3 | 2 | 4 | 5

Output:- 0, 1, 3, 2, 4, 5

Lab program 1.2:

```
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> graph(numCourses);
        vector<int> indegree(numCourses, 0);
        for (const auto& pre : prerequisites) {
            graph[pre[1]].push_back(pre[0]);
            indegree[pre[0]]++;
        }
        queue<int> q;
        for (int i = 0; i < numCourses; ++i) {
            if (indegree[i] == 0) q.push(i);
        }
        int count = 0;
        while (!q.empty()) {
            int curr = q.front(); q.pop();
            count++;
            for (int next : graph[curr]) {
                indegree[next]--;
                if (indegree[next] == 0) q.push(next);
            }
        }
        return count == numCourses;
    }
};
```



Lab program 2:

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>

void merge(int arr[], int left, int right, int mid) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];

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

    i = 0;
    j = 0;
```

```
k = left;
```

```
while(i < n1 && j < n2) {
```

```
    if(L[i] <= R[j]) {
```

```
        arr[k] = L[i];
```

```
        i++;
```

```
    } else {
```

```
        arr[k] = R[j];
```

```
        j++;
```

```
    }
```

```
    k++;
```

```
}
```

```
while(i < n1) {
```

```
    arr[k] = L[i];
```

```
    i++;
```

```
    k++;
```

```
}
```

```
while(j < n2) {
```

```
    arr[k] = R[j];
```

```
    j++;
```

```
    k++;
```

```
}
```

```
}
```

```
void mergeSort(int arr[], int left, int right) {
```

```
    if(left < right) {
```

```
        int mid = left + (right - left) / 2;
```

```
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, right, mid);
    }
}
```

```
void print(int arr[], int size) {
    for(int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
int main() {
    int n;
    clock_t start, end;

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

    int arr[n];

    srand(time(NULL));

    for(int i = 0; i < n; i++) {
        arr[i] = rand() % 1000;
    }
}
```

```
}
```

```
printf("Original Array: ");
```

```
print(arr, n);
```

```
start = clock();
```

```
mergeSort(arr, 0, n - 1);
```

```
end = clock();
```

```
printf("Sorted Array: ");
```

```
print(arr, n);
```

```
printf("Time taken: %f seconds\n", 1000 * (double)(end - start) / CLOCKS_PER_SEC);
```

```
return 0;
```

```
}
```

Screenshot of Output

[illegible]

11/3/25

LAB-01 Sort a given set of N integers/elements using Merge Sort Technique and compute its time taken. Run the program for diff^r values of N and record the time taken to sort.

Merge Sort

```
- #include <stdio.h>
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];
    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];
    i = 0; j = 0; k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) { arr[k] = L[i]; ++i; }
        else { arr[k] = R[j]; ++j; }
        ++k;
    }
    while (j < n2) {
        arr[k] = R[j];
        ++j; ++k;
    }
    while (i < n1) {
        arr[k] = L[i];
        ++i; ++k;
    }
}
```

```

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);
    }
}

```

```

}
void run-esp(int n) {
    int arr[MAX_SIZE];
    for (int i = 0; i < n; ++i) {
        arr[i] = rand() % 1000;
    }
    clock_t start-time = clock();
    merge-sort(arr, 0, n - 1);
    clock_t end-time = clock();
    double time-taken = (double) (end-time - start-time) / CLOCKS_PER_SEC;
    printf("N: %d, Time taken: %.6f sec\n", n, time-taken);
}

```

```

}
int main() {
    srand(time(NULL));
    int n-values[100];
    printf("Enter no. of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (int i = 0; i < n; ++i) {
        scanf("%d", &n-values[i]);
    }
    merge-sort(n-values, 0, n);
}

```

Output: $N = 10$ - execution time = 0.00002 sec
 $N = 20$ - 0.000010 sec
 $N = 50$ - 0.000025 sec
 $N = 200$ - 0.000105 sec
 $N = 500$ - 0.000310 sec
 $N = 2000$ - 0.001502 sec
 $N = 5000$ - 0.003902 sec
 $N = 10000$ - 0.009804 sec

Algorithm:

Mergesort($A[0 \dots n-1]$)

if ($n > 1$)

copy $A[0 \dots [n/2]-1]$ to $B[0 \dots [n/2]-1]$

copy $A[n/2 \dots n-1]$ to $C[0 \dots [n/2]-1]$

Mergesort($B[0 \dots n/2-1]$)

Merge($C[0 \dots n/2-1]$)

Merge(B, C, A)

Merge($B[0 \dots p-1], C[0 \dots q-1], A[p \dots p+q-1]$)

$i \leftarrow 0, j \leftarrow 0, k \leftarrow 0$

while $i < p, j < q$ do

if $B[i] \leq C[j]$

$A[k] \leftarrow B[i]; i \leftarrow i+1$

else $A[k] \leftarrow C[j]; j \leftarrow j+1$

$k \leftarrow k+1;$

if $i = p$

copy $C[j \dots q-1]$ to $A[k \dots p+q-1]$

else copy $B[i \dots p-1]$ to $A[k, p+q-1]$

Enter no of element: 5

array of element: 61 25 43 26 34 43 61

Sort array: 25 26 34 43 61

left : 61 25 43

left : 61

right : 25 43

left : 25

right 43

merge [25 43]

merge [25 43 61]

right 26 34

left 26

Right 34

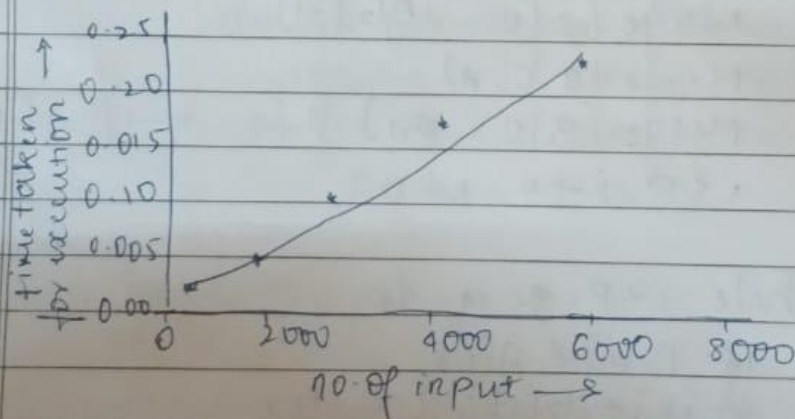
merge [25 43]

merge [25 43 61]

merge [25 43 61] [26 34]

final array [25 26 34 43 61]

Graph



Lab program 3:

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>
```

```
int partition(int arr[], int low, int high) {
```

```
    int pivot = arr[high];
```

```
    int i = low - 1;
```

```
    for (int j = low; j <= high - 1; j++) {
```

```
        if (arr[j] < pivot) {
```

```
            i++;
```

```
            int temp = arr[i];
```

```
            arr[i] = arr[j];
```

```
            arr[j] = temp;
```

```
        }
```

```
}
```

```
int temp = arr[i + 1];
```

```
arr[i + 1] = arr[high];
```

```
arr[high] = temp;
```

```
return (i + 1);
```

```
}
```

```
void quickSort(int arr[], int low, int high) {
```

```
    if (low < high) {
```

```
        int pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi - 1);
```

```
        quickSort(arr, pi + 1, high);
```

```
    }
```

```
}
```

```
void print(int arr[], int size) {
```

```
    for (int i = 0; i < size; i++) {
```

```
        printf("%d ", arr[i]);
```

```
    }
```

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

```
}
```

```
int main() {  
    int n;  
    clock_t start, end;  
  
    printf("Enter the number of elements in the array: ");  
    scanf("%d", &n);  
  
    int arr[n];  
  
    srand(time(NULL));  
  
    for (int i = 0; i < n; i++) {  
        arr[i] = rand() % 1001;  
    }  
  
    printf("Original Array: ");  
    print(arr, n);  
  
    start = clock();  
  
    quickSort(arr, 0, n - 1);
```


[illegible]

04/09/25
LAB-03

Quick Sort

```
#include <stdio.h>

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

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    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);
}

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

void printArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```

int main() {
    int n;
    printf("Enter no. of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; ++i) {
        scanf("%d", &arr[i]);
    }
    quicksort(arr, 0, n-1);
    printf("Sorted Array: \n");
    printArray(arr, n);
    return 0;
}

```

Output: Enter no. of elements: 9
 Enter elements: 20 2 9 7 12 15 1 6 8
 Sorted array:
 1 2 6 7 8 9 12 15 20

Algorithm:-

step 1: Choose a pivot element

step 2: Partition the array

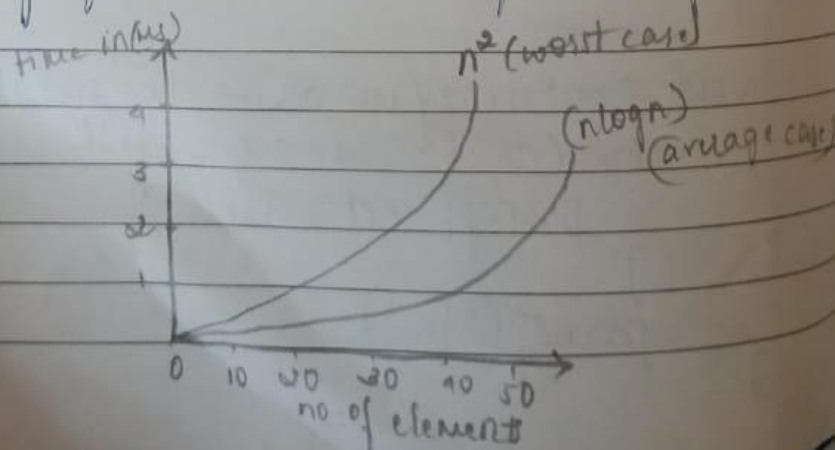
* Elements smaller to pivot move to its left

* Elements greater to pivot move to its right

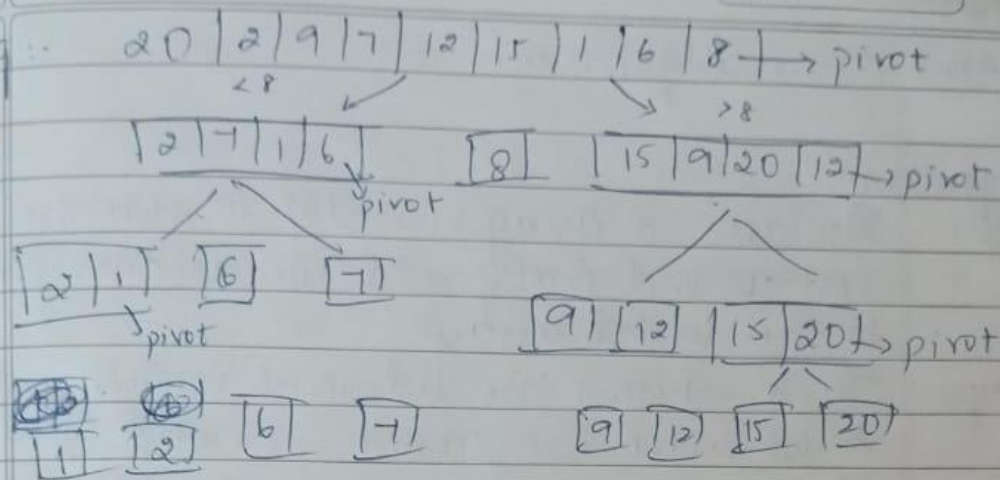
step 3: Recursively apply quick sort

apply quick sort on the left and right sequentially

Graph



Traing ::



sorted array = $\{1, 2, 6, 7, 9, 12, 15, 20\}$

Lab program 4:

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

Code

```
#include<stdio.h
```

```

#include<conio.h>

int cost[10][10],vt[10],et[10][10],vis[10],j,n;
int sum=0;
int x=1;
int e=0;
void prims();

void main()
{
    int i;

    printf("enter the number of vertices\n");
    scanf("%d",&n);
    printf("enter the cost adjacency matrix\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);
        }
        vis[i]=0;
    }
    prims();
    printf("edges of spanning tree\n");
    for(i=1;i<=e;i++)
    {
        printf("%d,%d\t",et[i][0],et[i][1]);
    }
}

```

```
printf("weight=%d\n",sum);  
getch();  
}
```

```
void prims()  
{  
    int s,min,m,k,u,v;  
    vt[x]=1;  
    vis[x]=1;  
    for(s=1;s<n;s++)  
    {  
        j=x;  
        min=999;  
        while(j>0)  
        {  
            k=vt[j];  
            for(m=2;m<=n;m++)  
            {  
                if(vis[m]==0)  
                {  
                    if(cost[k][m]<min)  
  
                        j--;
```

```
    }  
    vt[++x]=v;  
    et[s][0]=u;  
    et[s][1]=v;  
    e++;  
    vis[v]=1;  
    sum=sum+min;  
}  
}
```

Screenshot of Output

```
enter the number of vertices  
5  
enter the cost adjacency matrix  
999 2 999 6 999  
2 999 3 8 5  
999 3 999 999 7  
6 8 999 999 9  
999 5 7 9 999  
edges of spanning tree  
1,2      2,3      2,5      1,4      weight=16
```

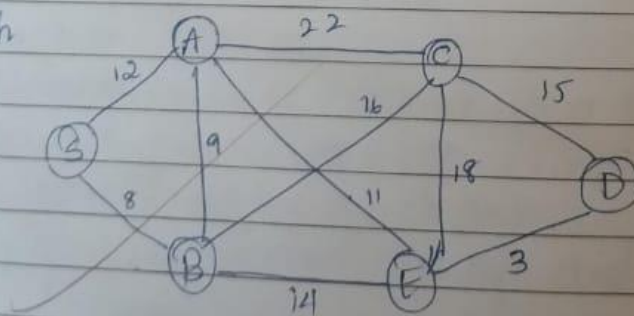
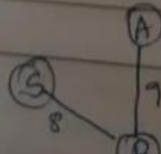

04/04/25

LAB 0 Prime algorithm

Algorithm:

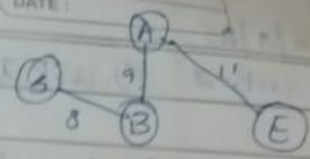
- step 1: Declare an array visited[] to store the visited vertices and firstly, add the arbitrary root to the visited array.
- step 2: Check whether the adjacent vertices of the last visited vertex are present in visited [] or not.
- step 3: If the vertices are not in the visited[] array, compare the cost of edges and add the least cost edge to output spanning tree.
- step 4: The adjacent unvisited vertex with the least cost edge is added into the visited[] array and the least cost edge is added to minimum spanning tree output.
- step 5: Steps 2 & 4 are repeated for all the unvisited in the graph to obtain the full minimum spanning tree output for the given graph.
- step 6: Calculate the cost of the minimum spanning tree obtained.

input graph

 $V = \{A\} : S \rightarrow B = 8$ $V = \{S, B\}$  $B \rightarrow A = 9 ; B \rightarrow C = 16 ; B \rightarrow E = 11$ $V = \{S, B, A\}$ 

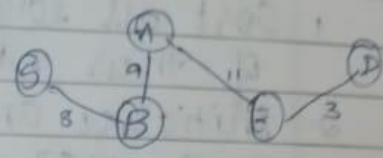
$A \rightarrow C = 20$; $A \rightarrow B = 9$; $A \rightarrow E = 11$

$V = \{S, B, A, E\}$



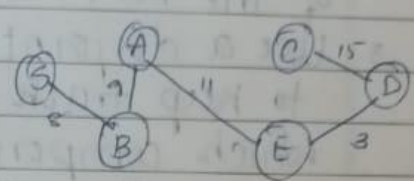
$E \rightarrow C = 18$; $E \rightarrow D = 3$

$V = \{S, B, A, E, D\}$

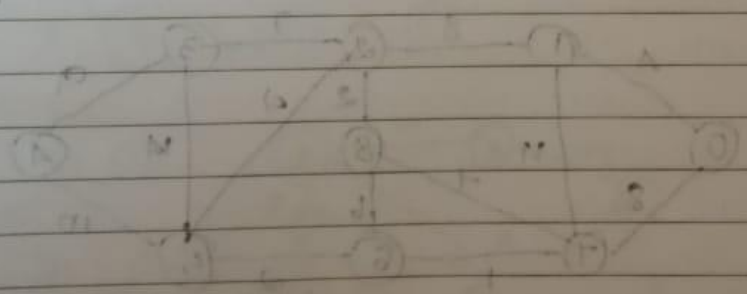


$D \rightarrow C = 15$; $E \rightarrow D = 3$

$V = \{S, B, A, E, D, C\}$



Spit



Lab program 5:

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

Code

```
#include<stdio.h>

#include<conio.h>


int find(int v,int parent[10])
{
    while(parent[v]!=v)
    {
        v=parent[v];
    }
    return v;
```

```
}
```

```
void union1(int i,int j,int parent[10])
```

```
{
```

```
    if(i<j)
```

```
        parent[j]=i;
```

```
    else
```

```
        parent[i]=j;
```

```
}
```

```
void kruskal(int n,int a[10][10])
```

```
{
```

```
    int count,k,min,sum,i,j,t[10][10],u,v,parent[10];
```

```
    count=0;
```

```
    k=0;
```

```
    sum=0;
```

```
    for(i=0;i<n;i++)
```

```
        parent[i]=i;
```

```
    while(count!=n-1)
```

```
    {
```

```
        min=999;
```

```
        for(i=0;i<n;i++)
```

```
        {
```

```
            for(j=0;j<n;j++)
```

```
            {
```

```
                if(a[i][j]<min && a[i][j]!=0)
```

```
                {
```

```
                    min=a[i][j];
```

```
                    u=i;
```

```
                    v=j;
```

```
                }
```

```

        }
    }
    i=find(u,parent);
    j=find(v,parent);
    if(i!=j)
    {
        union1(i,j,parent);
        t[k][0]=u;
        t[k][1]=v;
        k++;
        count++;
        sum=sum+a[u][v];
    }
    a[u][v]=a[v][u]=999;
}
if(count==n-1)
{
    printf("spanning tree\n");
    for(i=0;i<n-1;i++)
    {
        printf("%d %d\n",t[i][0],t[i][1]);
    }
    printf("cost of spanning tree=%d\n",sum);
}
else
    printf("spanning tree does not exist\n");
}

```

```

void main()

```

```

{
    int n,i,j,a[10][10];

```

```

clrscr();

printf("enter the number of nodes\n");

scanf("%d",&n);

printf("enter the adjacency matrix\n");

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

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

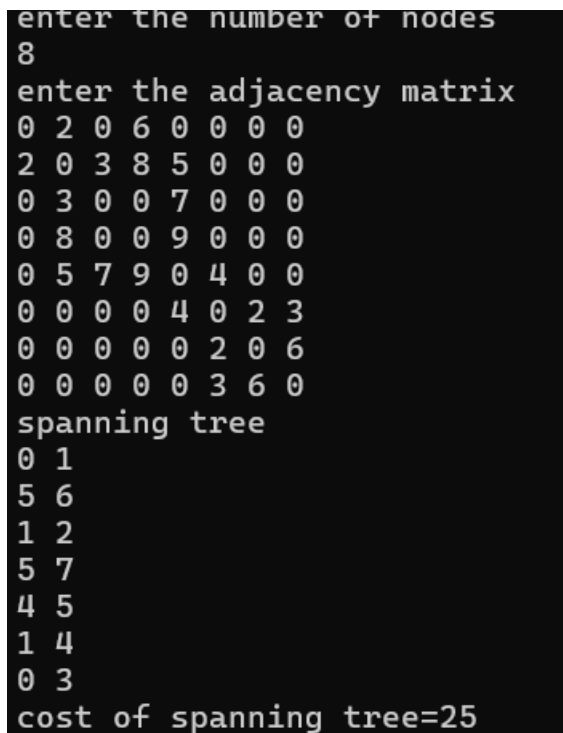
        scanf("%d",&a[i][j]);

kruskal(n,a);

getch();
}

```

Screenshot of Output



```

enter the number of nodes
8
enter the adjacency matrix
0 2 0 6 0 0 0 0
2 0 3 8 5 0 0 0
0 3 0 0 7 0 0 0
0 8 0 0 9 0 0 0
0 5 7 9 0 4 0 0
0 0 0 0 4 0 2 3
0 0 0 0 0 2 0 6
0 0 0 0 0 3 6 0
spanning tree
0 1
5 6
1 2
5 7
4 5
1 4
0 3
cost of spanning tree=25

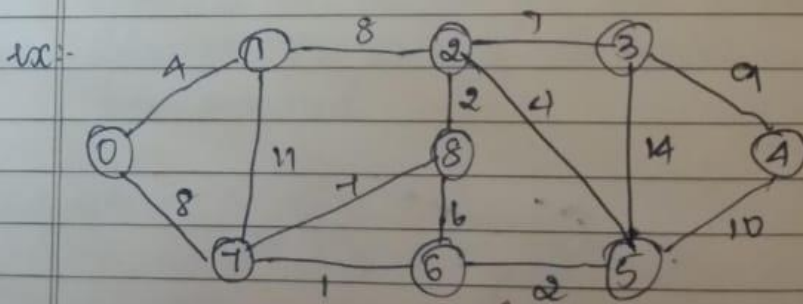
```

4/4/25

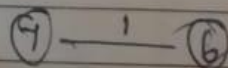
LAB-2

Kruskal's Algorithm

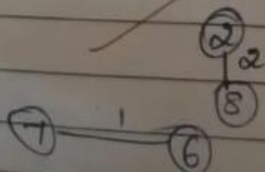
1. Sort all edges in non-decreasing order of their weights.
2. Initialize an empty set to store the edges of the MST.
3. Use a disjoint set (union-find) data structure to keep track of which vertices are in which components.
4. Iterate through the sorted edges:
 - For each edge (u, v) :
 - if u & v belong to different components, include this edge in MST and union the components.
 - if they are in same component, skip the edge.
5. Stop when the MST has $(v-1)$ edges.



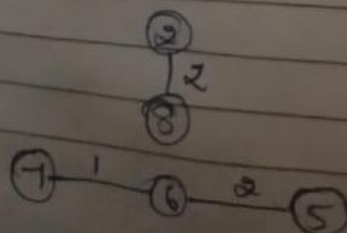
step 1:-



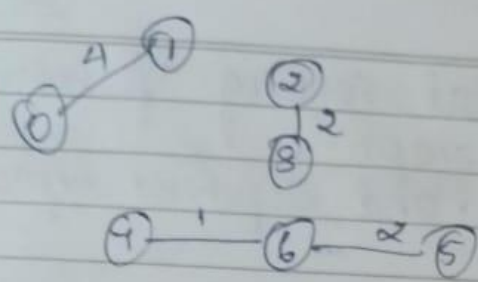
step 2:-



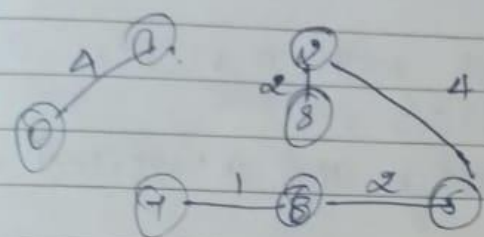
step 3:-



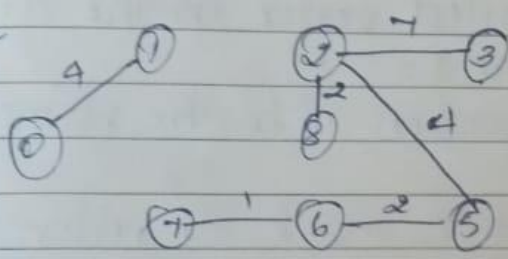
step 4:-



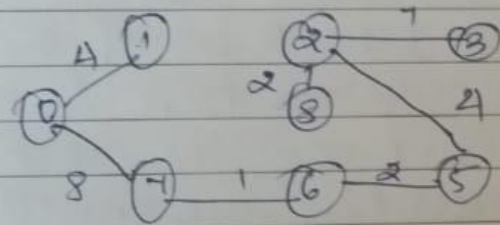
step 5:-



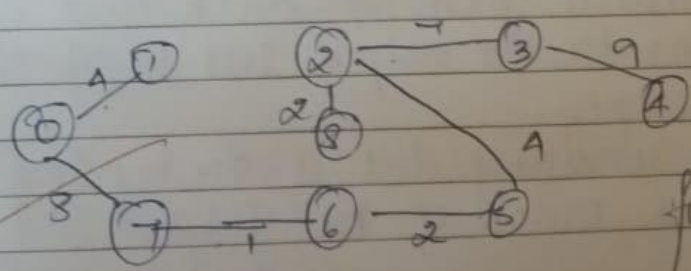
step 6:-



step 7:-



step 8:-



final MST

Lab program 6:

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

Code

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



```
#define INF 999
```

```
void dijkstra(int n, int cost[10][10], int src) {
```

```
    int i, j, u, dis[10], vis[10], min;
```

```
    // Initialize distances and visited flags
```

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

```
        dis[i] = cost[src][i];
```

```
        vis[i] = 0;
```

```
    }
```

```
    vis[src] = 1;
```

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

```
        min = INF;
```

```
        u = -1;
```

```
        // Find the unvisited vertex with the smallest distance
```

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

```
            if (vis[j] == 0 && dis[j] < min) {
```

```
                min = dis[j];
```

```
                u = j;
```

```
            }
```

```
        }
```

```
        if (u == -1) break; // All reachable vertices visited
```

```
        vis[u] = 1;
```

```

// Update distances to neighboring vertices
for (j = 1; j <= n; j++) {
    if (vis[j] == 0 && dis[u] + cost[u][j] < dis[j]) {
        dis[j] = dis[u] + cost[u][j];
    }
}

printf("Shortest paths from vertex %d:\n", src);
for (i = 1; i <= n; i++) {
    if (dis[i] == INF)
        printf("%d -> %d = INF\n", src, i);
    else
        printf("%d -> %d = %d\n", src, i, dis[i]);
}

}

int main() {
    int src, j, cost[10][10], n, i;

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

    printf("Enter the cost adjacency matrix (use 999 for no connection):\n");
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            scanf("%d", &cost[i][j]);
        }
    }
}

```

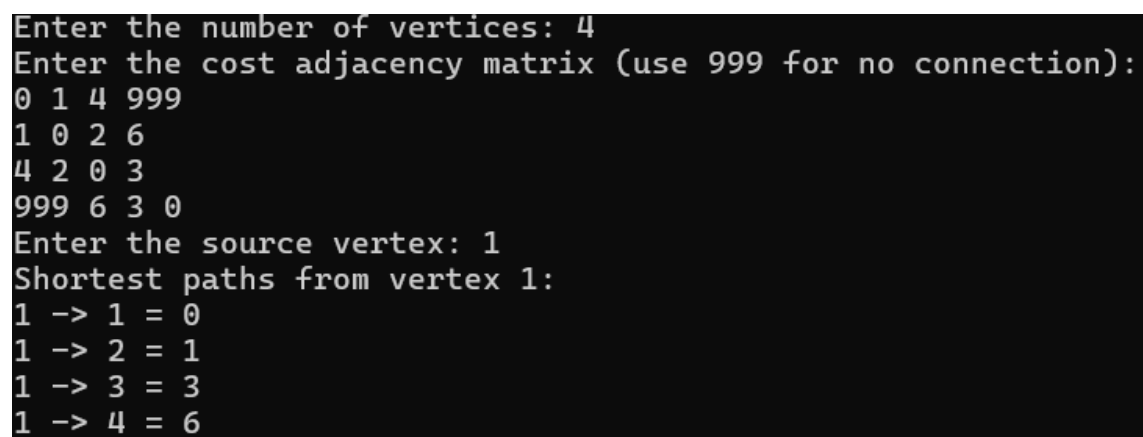
```
}

printf("Enter the source vertex: ");
scanf("%d", &src);

dijkstra(n, cost, src);

return 0;
}
```

Screenshot of Output



```
Enter the number of vertices: 4
Enter the cost adjacency matrix (use 999 for no connection):
0 1 4 999
1 0 2 6
4 2 0 3
999 6 3 0
Enter the source vertex: 1
Shortest paths from vertex 1:
1 -> 1 = 0
1 -> 2 = 1
1 -> 3 = 3
1 -> 4 = 6
```

17/05/25

PAGE NO :

DATE :

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

Algorithm.

step 1: set $\text{dist}[\text{source}] = 0$ and all other distances as infinity.

step 2: Push the source node into the min heap as a pair $\langle \text{distance}, \text{node} \rangle$ i.e., $\langle 0, \text{source} \rangle$.

step 3: Pop the top element (node with the smallest distance)

1. For each adjacent neighbor of the current node:

2. Calculate the distance using the formula.

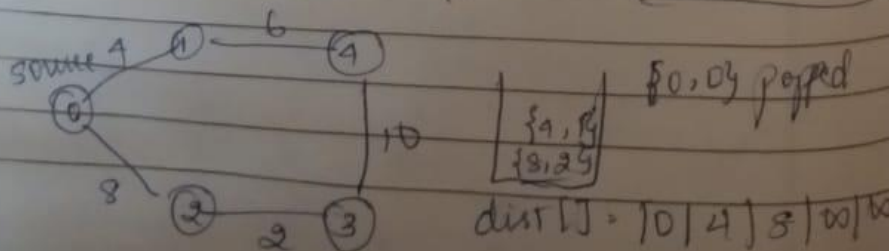
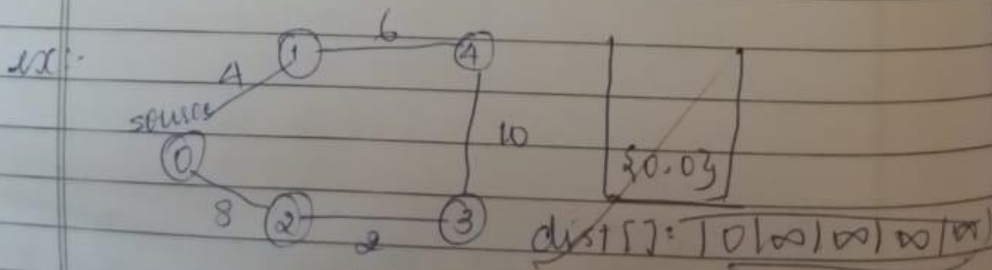
- $\text{dist}[v] = \text{dist}[u] + \text{weight}[u][v]$

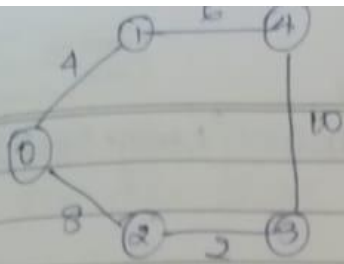
- If this new distance is shorter than the current $\text{dist}[v]$, update it.

- push the updated pair $\langle \text{dist}[v], v \rangle$ into the min heap.

step 4: Repeat step 3 until the min heap is empty.

step 5: Return the distance array, which holds the shortest distance from the source to all nodes.

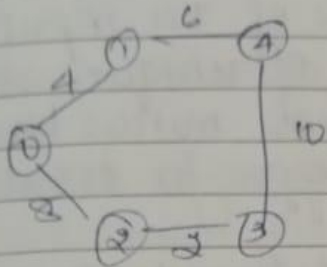




$\{8, 2\}$
 $\{10, 4\}$

PAGE NO
DATE

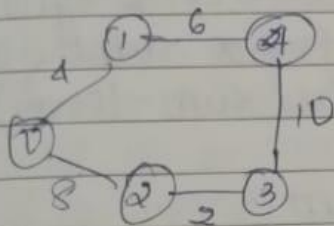
$\text{dist}[j] = [0 | 4 | 8 | \infty | 10]$



$\{10, 3\}$
 $\{10, 4\}$

$\{8, 2\}$ pop

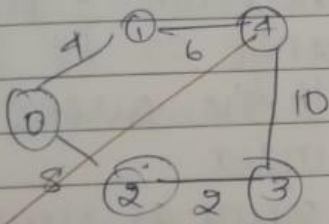
$\text{dist}[j] = [0 | 4 | 8 | 10 | 10]$



$\{10, 4\}$

$\{10, 3\}$ pop

$\text{dist}[j] = [0 | 4 | 8 | 10 | 10]$



$\{10, 4\}$

$\{10, 4\}$ pop

$\text{dist}[j] = [0 | 4 | 8 | 10 | 10]$

Sp. 8
 10/5/21

Lab program 7:

Implement Johnson Trotter algorithm to generate permutations.

Code

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

```
#define LEFT_TO_RIGHT 1
```

```
#define RIGHT_TO_LEFT 0
```

```

int searchArr(int a[], int n, int mobile) {
    for (int i = 0; i < n; i++)
        if (a[i] == mobile)
            return i + 1;
    return -1;
}

```

```

int getMobile(int a[], int dir[], int n) {
    int mobile_prev = 0, mobile = 0;

    for (int i = 0; i < n; i++) {
        if (dir[a[i] - 1] == RIGHT_TO_LEFT && i != 0) {
            if (a[i] > a[i - 1] && a[i] > mobile_prev) {
                mobile = a[i];
                mobile_prev = mobile;
            }
        }
        if (dir[a[i] - 1] == LEFT_TO_RIGHT && i != n - 1) {
            if (a[i] > a[i + 1] && a[i] > mobile_prev) {
                mobile = a[i];
                mobile_prev = mobile;
            }
        }
    }

    return mobile;
}

```

```

void printOnePerm(int a[], int dir[], int n) {

```

```
int mobile = getMobile(a, dir, n);
```

```
int pos = searchArr(a, n, mobile);
```

```
if (mobile == 0) return;
```

```
if (dir[a[pos - 1] - 1] == RIGHT_TO_LEFT) {
```

```
    int temp = a[pos - 1];
```

```
    a[pos - 1] = a[pos - 2];
```

```
    a[pos - 2] = temp;
```

```
} else if (dir[a[pos - 1] - 1] == LEFT_TO_RIGHT) {
```

```
    int temp = a[pos];
```

```
    a[pos] = a[pos - 1];
```

```
    a[pos - 1] = temp;
```

```
}
```

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

```
    if (a[i] > mobile) {
```

```
        dir[a[i] - 1] = !dir[a[i] - 1]; // toggle direction
```

```
    }
```

```
}
```

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

```
    printf("%d", a[i]);
```

```
    printf(" ");
```

```
}
```

```
int fact(int n) {
```

```
    int res = 1;
```

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



```

        res = res * i;

    return res;
}

void printPermutation(int n) {
    int a[n], dir[n];

    for (int i = 0; i < n; i++) {
        a[i] = i + 1;
        printf("%d", a[i]);
    }

    printf("\n");

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

    for (int i = 1; i < fact(n); i++)
        printOnePerm(a, dir, n);
}

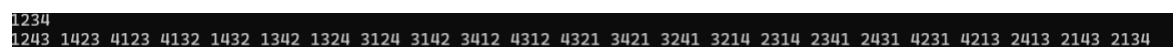
int main() {
    int n = 4;

    printPermutation(n);

    return 0;
}

```

Screenshot of Output



```

1234
1243 1423 4123 4132 1432 1342 1324 3124 3142 3412 4312 4321 3421 3241 3214 2314 2341 2431 4231 4213 2413 2143 2134

```

29/5/25

PAGE NO :

DATE :

Implement Johnson Trotter algorithm to generate permutations.

Algorithm

- Step 1: Find out the largest mobile integer in a particular sequence. A directed integer is said to be mobile if it is greater than its immediate neighbour in the direction it is looking at.
- Step 2: Switch this mobile integer and the adjacent integer to which its direction points.
- Step 3: Switch the direction of all the elements whose value is greater than the mobile integer value.
- Step 4: Repeat the step 1 until unless there is no mobile integer left in the sequence.

Ex: $\leftarrow 1 \leftarrow 2 \leftarrow 3$

Iteration 1 $\rightarrow \leftarrow 1 \leftarrow 2 \leftarrow 3$
 2 $\rightarrow \leftarrow 1 \leftarrow 3 \leftarrow 2$
 3 $\rightarrow \leftarrow 3 \leftarrow 1 \leftarrow 2$
 4 $\rightarrow \leftarrow 3 \leftarrow 2 \leftarrow 1$
 5 $\rightarrow \leftarrow 2 \leftarrow 3 \leftarrow 1$
 6 $\rightarrow \leftarrow 2 \leftarrow 1 \leftarrow 3$

\therefore Output: $\begin{matrix} 1 & 2 & 3 \\ 1 & 3 & 2 \\ 3 & 1 & 2 \\ 3 & 2 & 1 \\ 2 & 3 & 1 \\ 2 & 1 & 3 \end{matrix}$

program 8.1:

Implement Fractional Knapsack using Greedy technique.

Code

```
#include <stdio.h>

int main() {
    float weight[50], profit[50], ratio[50];
    float Totalvalue = 0.0, temp, capacity, amount;
    int n, i, j;

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

    for (i = 0; i < n; i++) {
        printf("Enter Weight and Profit for item[%d]:\n", i);
        scanf("%f %f", &weight[i], &profit[i]);
    }

    printf("Enter the capacity of knapsack:\n");
    scanf("%f", &capacity);

    // Calculate profit/weight ratio
    for (i = 0; i < n; i++)
        ratio[i] = profit[i] / weight[i];

    // Sort items by descending ratio
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (ratio[i] < ratio[j]) {
```

```

        // Swap ratio
        temp = ratio[i];
        ratio[i] = ratio[j];
        ratio[j] = temp;

        // Swap weight
        temp = weight[i];
        weight[i] = weight[j];
        weight[j] = temp;

        // Swap profit
        temp = profit[i];
        profit[i] = profit[j];
        profit[j] = temp;
    }
}

printf("\nKnapsack problem using Greedy Algorithm:\n");
for (i = 0; i < n; i++) {
    if (weight[i] <= capacity) {
        // Take full item
        printf("Item[%d] taken completely (100%%)\n", i);
        Totalvalue += profit[i];
        capacity -= weight[i];
    } else {
        // Take fraction of item
        float fraction = capacity / weight[i];
        Totalvalue += profit[i] * fraction;
    }
}

```

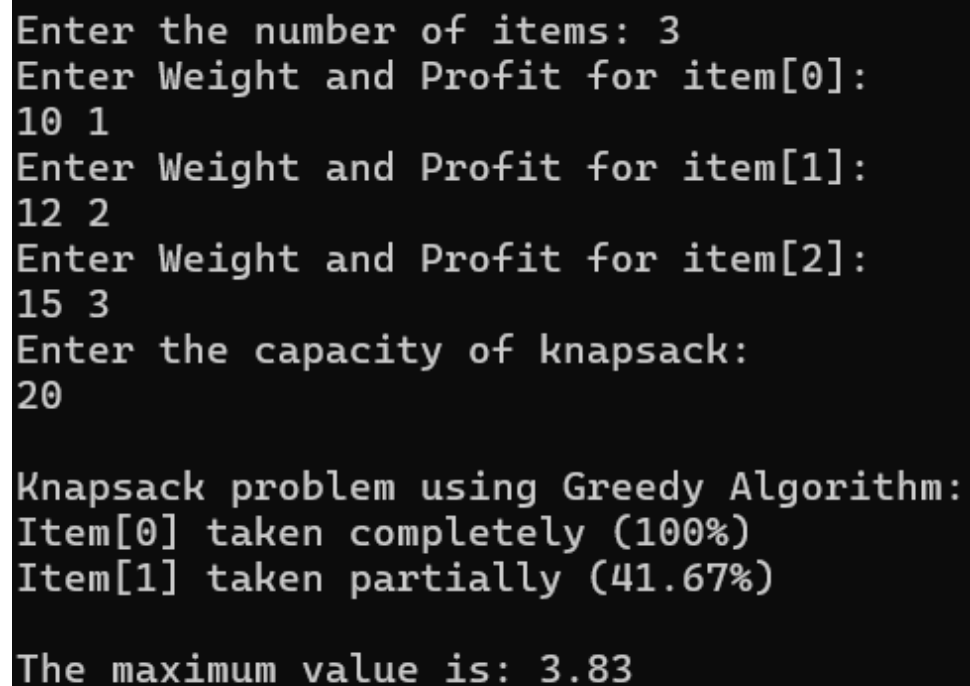
```

        printf("Item[%d] taken partially (%.2f%%)\n", i, fraction * 100);
        break; // Knapsack is now full
    }
}

printf("\nThe maximum value is: %.2f\n", Totalvalue);
return 0;
}

```

Screenshot of Output



```

Enter the number of items: 3
Enter Weight and Profit for item[0]:
10 1
Enter Weight and Profit for item[1]:
12 2
Enter Weight and Profit for item[2]:
15 3
Enter the capacity of knapsack:
20

Knapsack problem using Greedy Algorithm:
Item[0] taken completely (100%)
Item[1] taken partially (41.67%)

The maximum value is: 3.83

```

16/05/20

PAGE NO.
DATE

Implement Fractional knapsack using Greedy technique.

Algorithm:

- step 1: Calculate the ratio (profit/weight) for each item.
- step 2: Sort all the items in decreasing order of the ratio.
- step 3: Initialize res=0, current capacity = given capacity.
- step 4: Do the following for every item i in the sorted order:
 - * if the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result.
 - * Else add the current item as much as we can and break out of the loop.
- step 5: Return res.

Ex: $val[] = [60, 100, 120]$
 $wt[] = [10, 20, 30]$ capacity = 50
Sorting $\Rightarrow \{60, 100, 120\}, \{10, 20, 30\}$
iteration: $i=0$ weight = 10 < 50 \rightarrow add to the knapsack profit = 60 \rightarrow remaining cap = 40
 $i=1$: weight = 20 < 40 profit = 60 + 100 = 160
remaining capacity = 40 - 20 = 20
 $i=2$: weight = 30 > 20 \therefore fraction = $20/30 = 2/3 \times 120$
= 80 \therefore profit = 160 + 80 = 240 capacity = 240
 \therefore the final profit after capacity full is 240

Output: 240

Lab program 8.2:

LeetCode Program related to Greedy Technique algorithms

Code

```
char* largestOddNumber(char* num) {  
    int len = strlen(num);
```

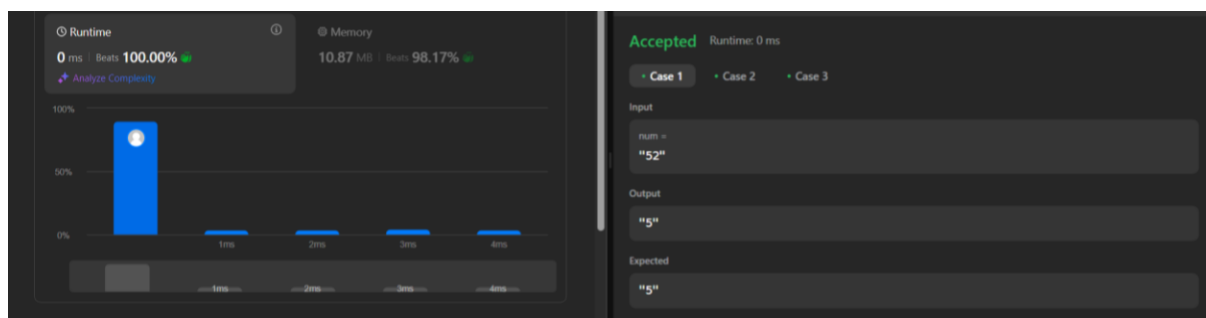
```

for (int i = len - 1; i >= 0; i--) {
    if ((num[i] - '0') % 2 == 1) {
        num[i + 1] = '\0'; // Truncate string at that position
        return num; // Return the longest odd-suffix (greedy)
    }
}

return ""; // No odd digit found
}

```

Screenshot of Output



Lab program 9.1:

Implement 0/1 Knapsack problem using dynamic programming.

Code

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

```
// Function to return the maximum of two numbers
```

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

```
// Function to solve the 0/1 Knapsack problem
```



```

int knapsack(int weight[], int profit[], int n, int capacity) {
    int i, w;
    int K[n + 1][capacity + 1];

    // Build the DP table K[][] bottom up
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (weight[i - 1] <= w)
                K[i][w] = max(profit[i - 1] + K[i - 1][w - weight[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    // Optional: Print the items included
    printf("\nItems included:\n");
    w = capacity;
    for (i = n; i > 0 && w > 0; i--) {
        if (K[i][w] != K[i - 1][w]) {
            printf("Item %d (Weight: %d, Profit: %d)\n", i, weight[i - 1], profit[i - 1]);
            w -= weight[i - 1];
        }
    }

    return K[n][capacity];
}

```

```
int main() {  
    int n, capacity;  
    int weight[50], profit[50];  
    int i;  
  
    printf("Enter number of items: ");  
    scanf("%d", &n);  
  
    printf("Enter weight and profit for each item:\n");  
    for (i = 0; i < n; i++) {  
        printf("Item[%d] - Weight Profit: ", i + 1);  
        scanf("%d %d", &weight[i], &profit[i]);  
    }  
  
    printf("Enter the capacity of knapsack: ");  
    scanf("%d", &capacity);  
  
    int maxProfit = knapsack(weight, profit, n, capacity);  
  
    printf("\nMaximum profit: %d\n", maxProfit);  
    return 0;  
}
```

Screenshot of Output

```
Enter number of items: 4
Enter weight and profit for each item:
Item[1] - Weight Profit: 2 12
Item[2] - Weight Profit: 3 15
Item[3] - Weight Profit: 1 25
Item[4] - Weight Profit: 2 10
Enter the capacity of knapsack: 4

Items included:
Item 3 (Weight: 1, Profit: 25)
Item 2 (Weight: 3, Profit: 15)

Maximum profit: 40
```

11/9/25
11B-3

PAGE NO :
DATE :

Implement 0/1 knapsack problem using dynamic programming.

Algorithm

Step 1. Create dp table: Initialize a $(n+1) \times (\text{capacity} + 1)$ table with zeroes.

Step 2. Fill dp table.

- For each item i from $1 \rightarrow n$:
For each capacity w from 1 to capacity:
 - if $\text{weight}[i-1] \leq w$ then
 $\text{dp}[i][w] = \max(\text{value}[i-1] + \text{dp}[i-1][w - \text{weight}[i-1]], \text{dp}[i-1][w])$
 - else: $\text{dp}[i][w] = \text{dp}[i-1][w]$

Step 3. Backtrack for items:-

- initialize $\text{included}[]$, $\text{current_capacity} = \text{capacity}$, $i = n$.
- while $i > 0$ and $\text{current_capacity} > 0$
 - if $\text{dp}[i][\text{current_capacity}] \neq \text{dp}[i-1][\text{current_capacity}]$
 - add $i-1$ to included .
 - $\text{current_capacity} -= \text{weight}[i-1]$
 - decrement i

Step 4. Return: $\text{dp}[n][\text{capacity}]$ and included .

Ex:-

weights	values
2	300
1	200
5	400
3	500

capacity = 10;

Output :- Tabulation Table :-

Weight	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	300	300	300	300	300	300	300	300	300
1	0	200	300	500	500	500	500	500	500	500	500
5	0	200	300	500	500	500	600	1000	900	900	900
3	0	200	300	500	700	800	1000	1000	1000	1000	1200

Maximum value in knapsack = 1200

Included item indices (0-based) : 0 2 3

Included items:

Value : 300 , Weight : 2

Value : 400 , Weight : 5

Value : 500 , Weight : 3

Here:

- $dp[4][10] = 1200$, $dp[3][10] = 900$ different so weight 3 is included. Move to $dp[3][10-3] = dp[3][7]$
 - $dp[3][7] = 1000$, $dp[2][7] = 500$ diff. so weight 5 is included. Move to $dp[2][7-5] = dp[2][2]$
 - $dp[2][2] = 300$, $dp[1][2] = 300$ same not included. Move to $dp[1][2] = 300$
 - $dp[1][2] = 300$, $dp[0][2] = 0$ diff. so included. Weight 2. Move to $dp[0][0]$
- Backtracking complete.

Lab program 9.2:

Code

```
class Solution(object):

    def fib(self, n):

        if n == 0:

            return 0

        if n == 1:

            return 1

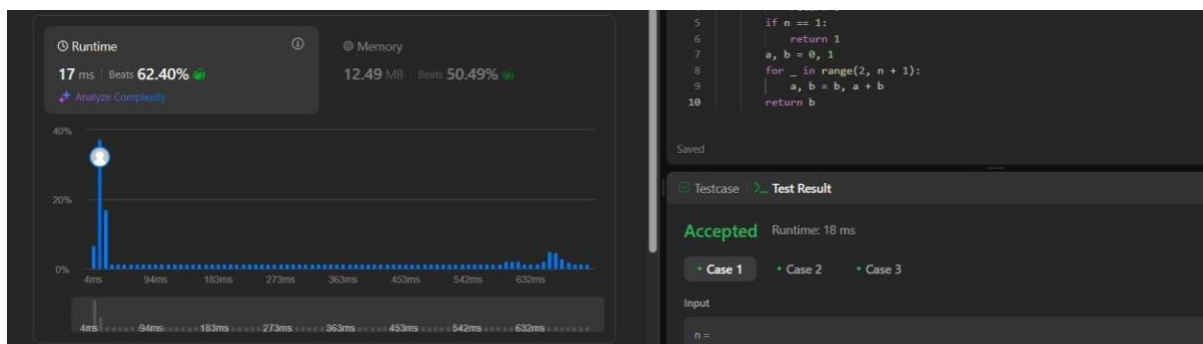
        a, b = 0, 1

        for _ in range(2, n + 1):

            a, b = b, a + b

        return b
```

Screenshot of Output



Lab program 10:

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

Code

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

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        heapify(arr, n, largest);
    }
}

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 temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        heapify(arr, i, 0);
    }
}
```

```

int main() {
    int arr[1000], n;
    clock_t start, end;
    double time_taken;

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

    printf("Enter %d integer elements:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    start = clock();

    heapSort(arr, n);

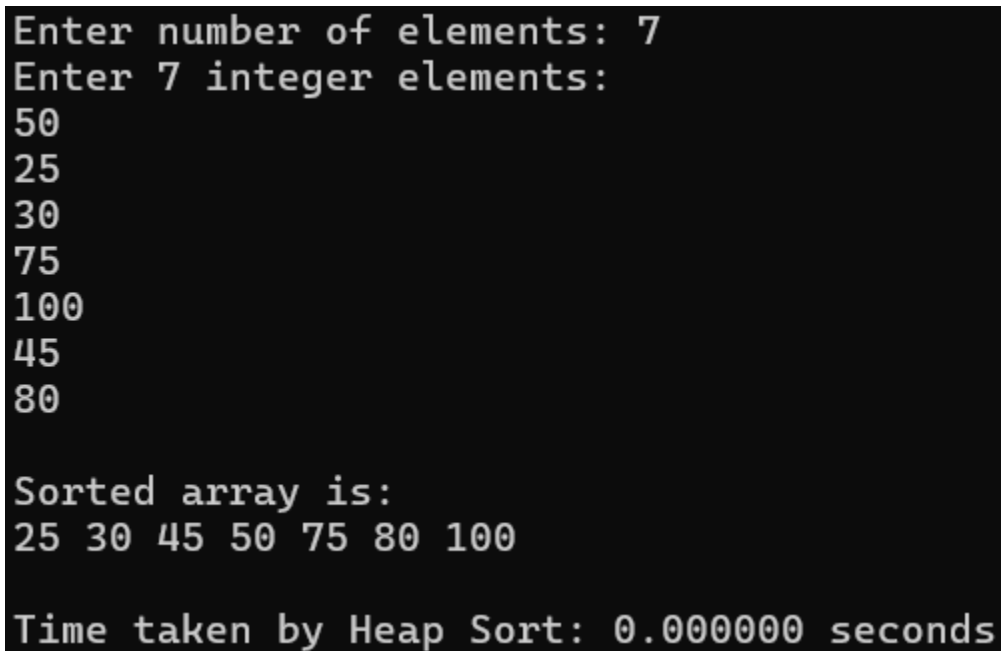
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nSorted array is:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    printf("\n\nTime taken by Heap Sort: %f seconds\n", time_taken);

    return 0;
}

```

Screenshot of Output



```

Enter number of elements: 7
Enter 7 integer elements:
50
25
30
75
100
45
80

Sorted array is:
25 30 45 50 75 80 100

Time taken by Heap Sort: 0.000000 seconds

```


9/15/25

PAGE NO:
DATE:

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

Algorithm::

Step 1: Rearrange array elements so that they form a Max Heap

Step 2: Repeat the following steps until the heap contains only one element:

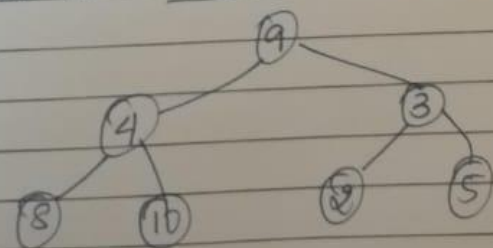
* Swap the root element of the heap (which is the largest element in current heap) with the last element of the heap.

* Remove the last element of the heap (which is now the current position). We mainly reduce heap size and do not remove element from the actual array.

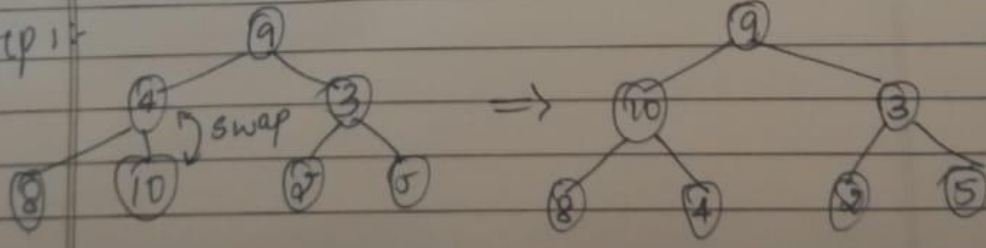
* Heapify the remaining elements of heap

Step 3: Finally we get the sorted array.

Ex: arr[] = [9, 4, 3, 8, 10, 2, 5]

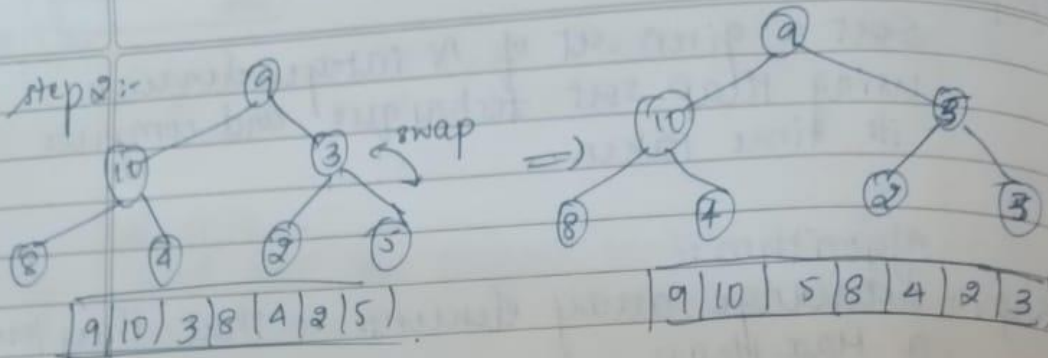


Step 1:

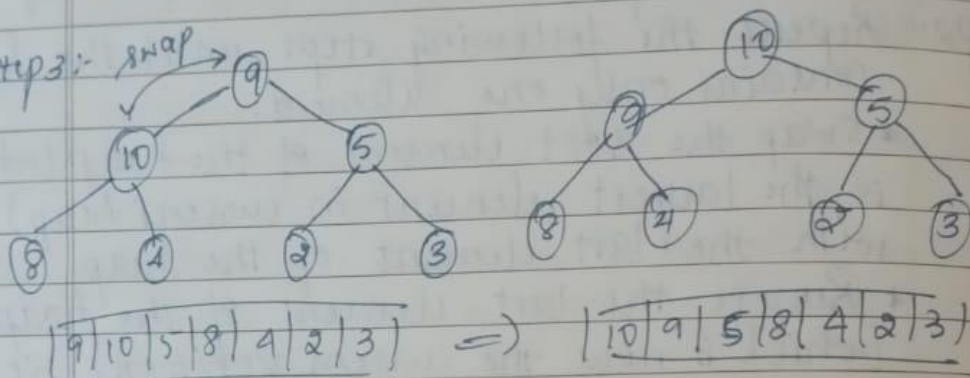


[9, 4, 3, 8, 10, 2, 5] \Rightarrow [5, 4, 3, 8, 10, 2, 9]

Step 2:-



Step 3:-



Output: Sorted array is

2 3 4 5 8 9 10

Lab program 11.1:

Implement All Pair Shortest paths problem using Floyd's algorithm.

Code

```
#include <stdio.h>

#define INF 99999 // Use a large number to represent infinity
#define MAX 100

void floydWarshall(int graph[MAX][MAX], int n) {
    int dist[MAX][MAX];
    int i, j, k;

    // Initialize the solution matrix same as input graph
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            dist[i][j] = graph[i][j];

    // Floyd-Warshall algorithm
    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the final shortest distance matrix
```

```

printf("\nAll-Pairs Shortest Paths (Floyd-Warshall):\n");

for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (dist[i][j] == INF)
            printf("INF ");
        else
            printf("%3d ", dist[i][j]);
    }
    printf("\n");
}

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

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

    printf("Enter the adjacency matrix (use 99999 for no direct path):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    floydWarshall(graph, n);

    return 0;
}

```

Screenshot of Output

```
Enter number of vertices: 4
Enter the adjacency matrix (use 99999 for no direct path):
0 4 3 9
99 0 1 99
99 990 99999
5 2 6 0
2 99 99999 99999
```

```
All-Pairs Shortest Paths (Floyd-Warshall):
0  4  3  8
8  0  1  6
7 11  5  5
2  6  0  2
```

1/5/25

Lab 3 Implement All pair shortest paths problem using Floyd's algorithm.

Algorithm :

Step 1: Use a matrix $dist[i][j]$ where $dist[i][j]$ holds the shortest distance from vertex i to vertex j .

Step 2: Initialize $dist[i][j]$ with:

- 0 if $i = j$
- weight of edge (i, j) if it exists
- ∞ if no direct edge exists

Step 3: update $dist[i][j]$ by considering each vertex k as an intermediate vertex.

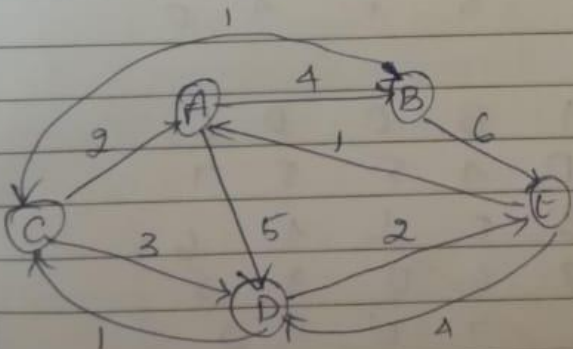
for k from 0 to $v-1$;

for i from 0 to $v-1$;

for j from 0 to $v-1$;

if $dist[i][k] + dist[k][j] < dist[i][j]$
 $dist[i][j] = dist[i][k] + dist[k][j]$

ex:-



Step 1: initial matrix :-

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

①	A	A	B	C	D	E	A	B	C	D	E	
Step 2	A	0	1	∞	5	∞	A	0	1	∞	5	∞
	B	∞	0	1	∞	6	B	∞	0	1	∞	6
	C	2	∞	0	3	∞	→ C	2	6	0	3	∞
	D	∞	∞	1	0	2	D	∞	∞	1	0	2
	E	1	∞	∞	4	0	E	1	5	∞	4	0

Step 3 ②

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

Step 4 ③

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

Step 5 ④

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 6 ⑤

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step ① Since all nodes have been treated as an intermediate node
 final result after applying algorithm

Output:-

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Sp. 7
 16/5/25

Lab program 11.2:

LeetCode Program related to shortest distance calculation

Code

class Solution:

```
def shortestPathLength(self, graph: List[List[int]]) -> int:
```

```
    n=len(graph)
```

```
    queue=deque([(i,1<<i) for i in range(n)])
```

```
    seen=set(queue)
```

```
    ans=0
```

```
    while queue:
```

```
        for _ in range(len(queue)):
```

```
            u,m=queue.popleft()
```

```
            if m==(1<<n)-1:
```

```
                return ans
```

```
            for v in graph[u]:
```

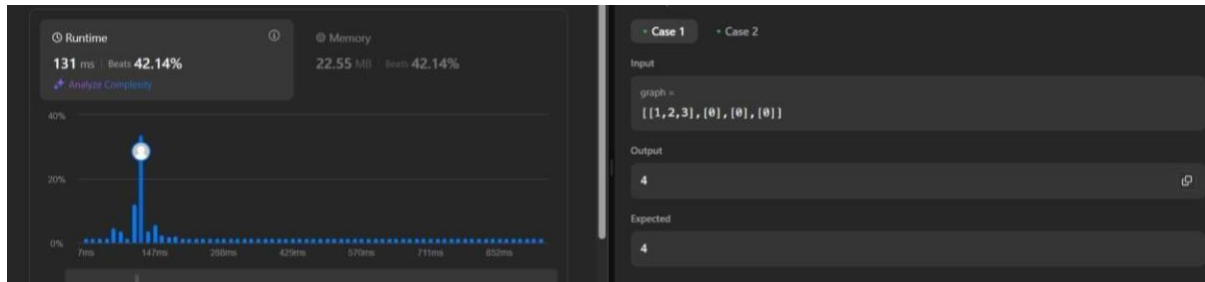
```
                if (v,m|1<<v) not in seen:
```

```
                    queue.append((v,m|1<<v))
```

```
                    seen.add((v,m|1<<v))
```

```
        ans+=1
```

Screenshot of Output



Lab program 12:

Implement “N-Queens Problem” using Backtracking.

Code

```
#include <stdio.h>

#include <math.h>

#define MAX 20

int board[MAX];

int found = 0;

// Function to print one solution
void printSolution(int n) {
    printf("One solution for %d-Queens:\n", n);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (board[i] == j)
                printf("Q ");
            else
```

```

        printf(". ");
    }
    printf("\n");
}
found = 1;
}

// Check if placing queen at (k, i) is safe
int isSafe(int k, int i) {
    for (int j = 1; j < k; j++) {
        if (board[j] == i || fabs(board[j] - i) == fabs(j - k))
            return 0;
    }
    return 1;
}

// Recursive backtracking to find one solution
void nQueens(int k, int n) {
    for (int i = 1; i <= n && !found; i++) {
        if (isSafe(k, i)) {
            board[k] = i;
            if (k == n)
                printSolution(n);
            else
                nQueens(k + 1, n);
        }
    }
}
}

```

```

int main() {
    int n;

    printf("Enter number of queens (N): ");
    scanf("%d", &n);

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

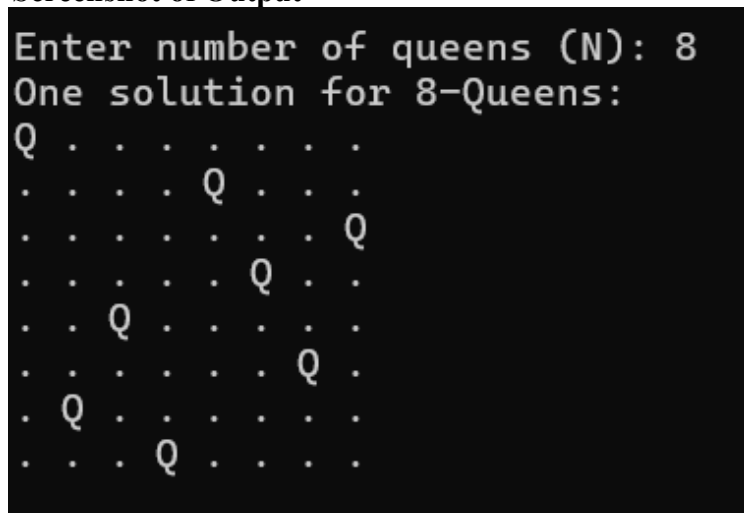
    nQueens(1, n);

    if (!found)
        printf("No solution exists for N = %d\n", n);

    return 0;
}

```

Screenshot of Output



```

Enter number of queens (N): 8
One solution for 8-Queens:
Q . . . . . . .
. . . Q . . .
. . . . . . Q
. . . . Q . .
. . Q . . . .
. . . . . Q .
. Q . . . . .
. . . Q . . .

```

23/05/20

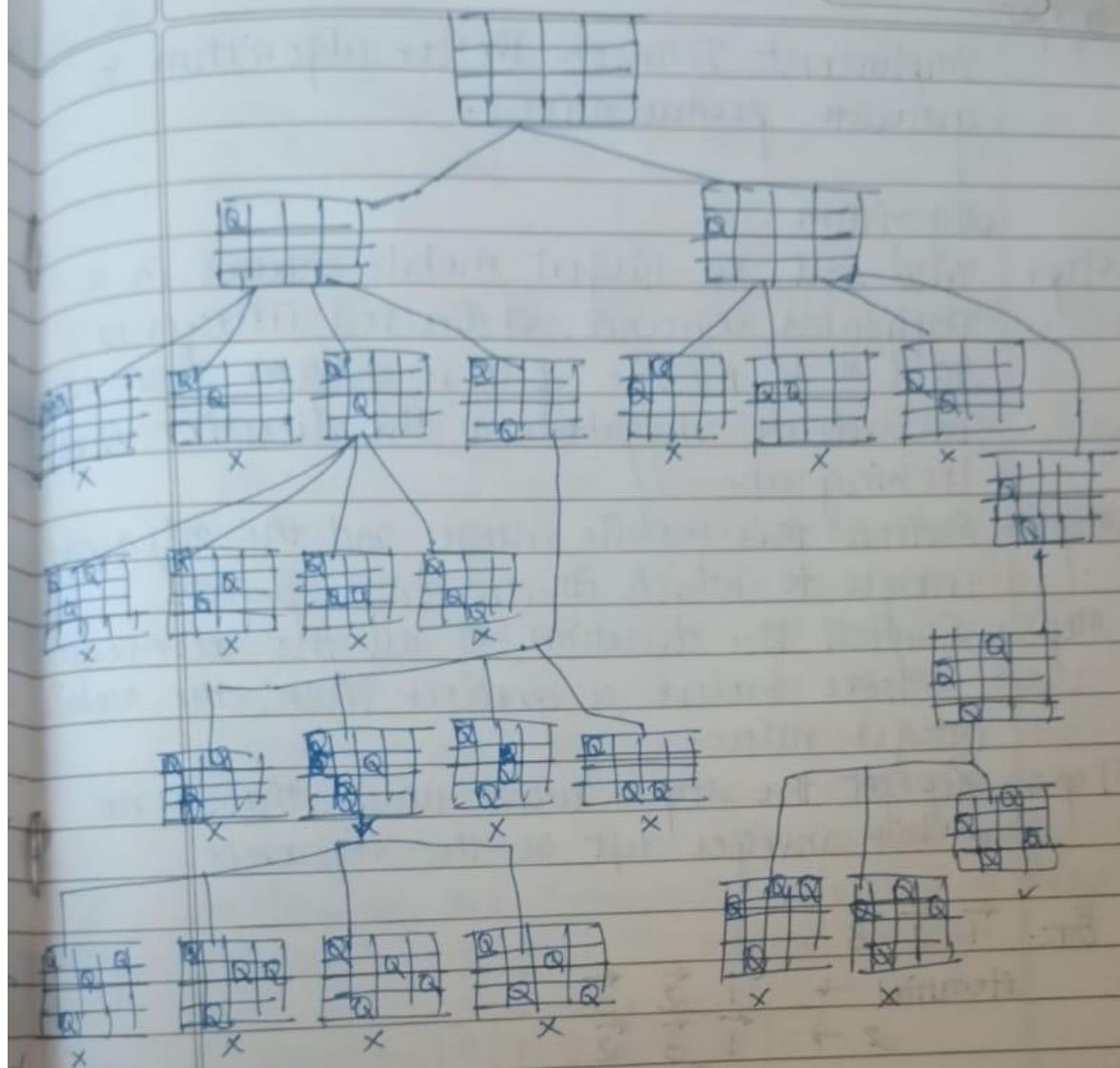
Implement 'N-Queens Problem' using Backtracking.

Algorithm:

- Step 1: Start from the leftmost column (column 0)
- Step 2: Try placing a queen in each row one by one in the current column.
- Step 3: For each placement, check if it's safe:
- * No queen in the same row to the left.
 - * No queen in the upper left diagonal.
 - * No queen in the lower left diagonal.
- Step 4: If it's safe, place the queen and move to the next column recursively.
- Step 5: If placing queen in any row in the current column leads to solution, return true.
- Step 6: If no row is safe in this column, backtrack: remove the queen and try it the next row.
- Step 7: Continue this until all N queens are placed.

Input: $n = 4$

Output: 2 4 1 3



\therefore final matrix is

			Q
Q			
	Q		
			Q

4 2 4 1 3