INDEX

PIONEER®

Name Tarun Kumar Dey

Class CSE Section B Roll No. A91005221040 Year 2021-25

Subject Analysis and Design of Algorithms.

S.No	Page No	Experiment Description	Experiment Date	Submission Date	Remarks Signature
1	1-2	Tower of Hanoi	19/09/23	26/09/23	26/09/23
2	3-4	linear search	19/09/23	26/09/23	
3	5-7	Merge sort	26/09/23	03/10/23	03/10/23
4	8-9	Fractional Knapsack	26/09/23	03/10/23	
5	10-12	Dijkstra Algo.	03/10/23	10/10/23	17/10/23
6	13-16	Prim's	10/10/23	17/10/23	
7	17-19	Kruskal's	10/10/23	17/10/23	
8	19-21	Travelling Salesman	17/10/23	31/10/23	07/11/23
9	22-26	N Queen	07/11/23	14/11/23	
10	27-28	Bellman Ford	31/10/23	07/11/23	
11	29-31	Quick Sort	07/11/23	14/11/23	21/11/23
12	32-33	BFS	14/11/23	21/11/23	
13	34-36	DFS	14/11/23	21/11/23	

DATE

EXPT. NO.

pioneerpaper.in
PAGE NO. 1

LAB:1

Wap to implement tower of hanoi Problem in c++.

```
#include <iostream>
using namespace std;
void Hanoi(int n, char A, char B, char C)
{
    if (n==0){
        return;
    }
    Hanoi (n-1, A, B, C);
    cout<<"move disk "<<n<<" from rod "<<A<<" to rod "
    <<C<<endl;
    Hanoi (n-1, B, C, A);
}
int main(){
    int N;
    cout<<"Enter number of disks: ";
    cin>>N;
    Hanoi (N, 'a', 'c', 'b');
    return 0;
}
```

DATE

EXPT. NO.

Algorithm:

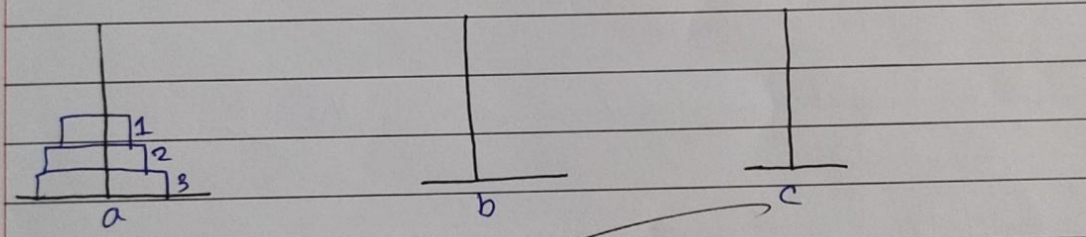
step 1: Create function tower_of_hanoi and pass N (no. of disk), best_rod, aux_rod, end_rod.

step 2: Make a function call for N-1 th disk.

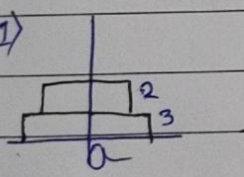
step 3: Print the current disk along with best_rod, end_rod.

step 4: Repeat step 2 and step 3 until the entire stack names to end_rod.

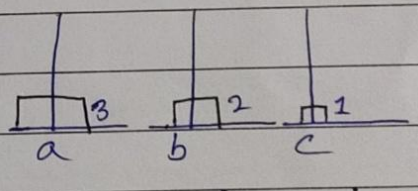
step 5: End.



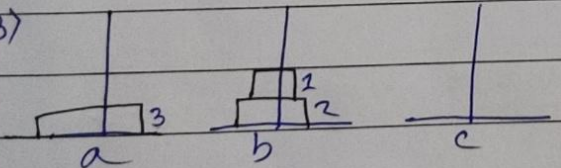
1)



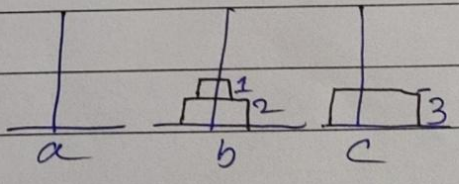
2)



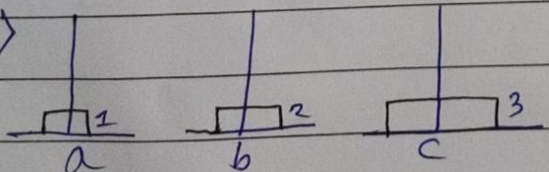
3)



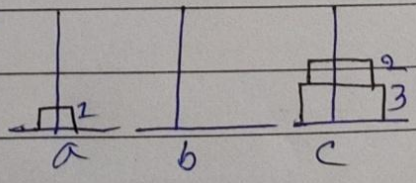
4)



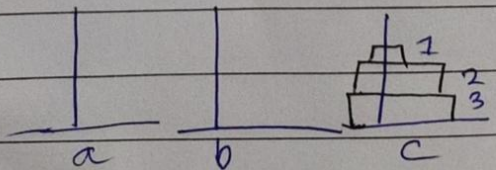
5)



6)

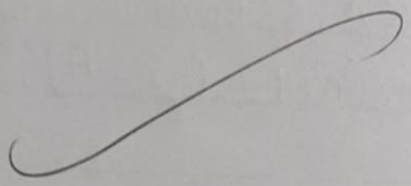


7)



Output:-

```
Enter number of disks : 3
Move disk 1 from rod a to rod c
Move disk 2 from rod a to rod b
move disk 1 from rod c to rod b
move disk 3 from rod a to rod c
move disk 1 from rod b to rod a
move disk 2 from rod b to rod c
move disk 1 from rod a to rod c
```



LAB: 2

pioneerpaper.in PAGE NO. 3

Write a Program to implement Linear Search in c++.

```
#include <iostream>
using namespace std;
int main() {
int arr[10], i, num, index;
cout << " Enter 10 Numbers : ";
for(i = 0; i<10; i++){
  cin >> arr[i];
}
cout<<" In Enter a Number to Search : ";
cin >> num;
for (i = 0; i < 10, i++){
  if (arr[i] == num){
    index = i;
    break;
  }
}
Cout << " In Found at Index No : " << index;
cout << endl;
return 0;
}
```

Output :-

```
Enter 10 Numbers : 98
65
45
34
76
12
32
92
63
52
Enter a Number to search :45
Found at index NO : 2
```

```
    right++;
  }
}
void ms(vector<int>& arr, int low, int high){
    if(low == high){
        return;
    }
    int mid = (low+high)/2;
    ms(arr, low, mid);
    ms(arr, mid+1, high);
    merge(arr, low, mid, high);
}
void mergesort(vector<int>& arr, int n){
    ms(arr, 0, n-1);
}
int main(){
    vector<int> arr;
    int n;
    cout<<"Enter array size:";
    cin>>n;
    for(int i=0; i<n; i++){
        arr.push_back(i);
    }
    cout<<"Unsorted Array:";
    for(int i=0; i<n; i++){
```

```
    cout << arr[i];
    }
    cout << endl;
    mergesort (arr, n);
    cout << "sorted Array";
    for (int i=0; i<n; i++){
        cout << arr[i];
    }
    cout << endl;
    return 0;
}
```

Algorithm :-

step 1 : create an unsorted array

step 2 : split the array from the middle into two part.

step 3 : Divide the subarrays until the array is unit length.

step 4 : sort the unit length subarrays.

step 5 : Merge the unit length sorted array.

step 6 : Print the entier sorted array.

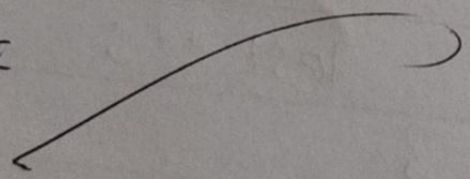
step 7 : End.

26/09/23

Output :-

```
Enter array size: 6
10
18
15
21
9
30
Unsorted array :
        10, 18, 15, 21, 9, 30

Sorted array :
        9, 10, 15, 18, 21, 30
```



LAB-3

pioneerpaper.in PAGE NO. 8

~~Knap~~ Fractional Knapsack Problem :-

```
#include <bits/stdc++.h>
using namespace std;
struct Item {
    int profit, weight;
    Item (int profit, int weight) {
        this -> profit = profit;
        this -> weight = weight;
    }
};
static bool cmp (struct Item a, struct Item b) {
double r1 = (double) a.profit / (double) a.weight;
double r2 = (double) b.profit / (double) b.weight;
return r1 > r2;
}
double fractionalKnapsack (int W, struct Item arr[],
                           int N) {
sort (arr, arr+N, cmp);
double finalvalue = 0.0;
for (int i = 0; i < N; i++) {
    if (arr[i].weight <= W) {
        W -= arr[i].weight;
        finalvalue += arr[i].profit;
    }
```

```
    else {
        finalvalue += arr[i].Profit * ((double)W/(double) arr[i].
                      weight);
        break;
    }
}
return finalvalue;
}
int main() {
    int W = 50;
    Item arr[] = {{60, 10},{100, 20}, {120, 30}};
    int N = sizeof(arr) / size of (arr[0]);
    cout << fractionalKnapsack(W, arr, N);
    return 0;
}
```

Algorithm:-

Step 1: Sort the given array of items according to weight/value (W/V) ratio in descending order.

Step 2: Start adding the item with the maximum W/V ratio.

Step 3: Add the whole item, if the current weight is less than the capacity, else add a portion of the item to the knapsack.

Step 4: Stop when weight is equal to the given Knapsack weight and count total values.

Output :

240

Prim's algorithm:-

(C++)

```
#include <bits/stdc++.h>
using namespace std;
#desine V5
int minKey (int Key[], bool mstSet[])
 {
   int min = INT_MAX, min_index;
   for (int v=0 ; v<V ; v++)
    if (mstSet[v] == false && Key[v] < min)
       min = Key[v], min_index =v;
   return min_index;
}
void printMST (int Parent[], int graph[v][v])
{
 count cout << "Edge \t width weight \n";
       for(int i =1; i<V ;i++)
        cout << Parent[i]<< " - "<< i << " \t" << graph[i]
        [Parent[i]] << "\n";
}
Void PrimMST (int graph[V][V])
 {
   int parent[V];
```

```
int Key [V];
bool mstSet [V];
for (int i=0 ; i<V ; i++)
  Key [i] = INT_MAX, mstSet [i] = false;
Key [0] =0;
Parent [0] =-1;
for(int count =0 ; count<V-1 ; count++) {
  int u = minKey (Key, mstSet);
  mstSet [u] = true;
  for (int v=0 ; v<V ; v++)
    if (graph [u] [v] && mstSet [v] == false && graph
      [u] [v] < Key [v])
      Parent [v] = u, Key [v] = graph [u] [v];
  }
  printMST (Parent, graph);
}
int main() {
      int graph [V] [V] = { {0, 2, 0, 6, 0},
                            {2, 0, 3, 8, 5},
                            {0, 3  0 , 0, 7},
                            {6, 8, 0, 0, 9},
                            {0, 5, 7, 9, 0}}
  PrimMST (graph);
  return 0;
}
```

Output :-

Edge	weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

Algorithm:-

> It's a greedy approach used for finding the minimum spanning tree in a weighted connected graph. Starting with a single node, it iteratly add the lowest weight edge connecting tree to a non tree vertex ensuring the tree spans all nodes while minimizing edge weight.

Kruskal's algorithm
C++

```
#include <bits/stdc++.h>
using namespace std;
class DSU {
    int* parent;
    int* rank;
public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];
        for (int i=0; i<n; i++){
            parent[i] = -1;
            rank[i] = 1;
        }
    }
int find(int i){
    if (parent[i] == -1)
        return i;
    return parent[i] = find(parent[i]);
}
void unite(int x, int y){
    int s1 = find(x);
```

```
int s2 = find(y);
if (s1 != s2){
    if (rank[s1] < rank[s2]){
        parent[s1] = s2;
    }
    else if (rank[s1] > rank[s2]){
        parent[s2] = s1;
    }
    else{
        parent[s2] = s1;
        rank[s1] += 1;
    }
  }
 }
};
class Graph{
    vector<vector<int>> edgelist;
    int V;
  public:
    Graph(int V){this -> V=V;}
    void addEdge(int x, int y, int w){
      edgelist.push_back({w, x, y});
    }
    void kruskals_mst(){
      sort(edgelist.begin(), edgelist.end());
```

```
DSU S(V);
int ans = 0;
cout << "Following are the edges in the ".
     " constructed MST" << endl;
for (auto edge : edgelist) {
    int w = edge[0];
    int x = edge[1];
    int y = edge[2];
    if (S.find(x) != S.find(y)) {
      S.unite(x, y);
      ans += w;
      cout << x << "--" << y << "==" << w << endl;
    }
  }
  cout << "Minimum Cost Spanning Tree:" << ans;
}
};
int main() {
    Graph g(4);
    g.addEdge(0, 1, 10);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);
    g.addEdge(2, 0, 6);
    g.addEdge(0, 3, 5);
```

```
g.KruskAls_mst ();
return 0;
}
```

=

Algorithm:-

> Khusikal's algorithm, a minimum spaning tree algorithm, grows a forest by adding the smallest edge from the graph while avoiding cycles. Sort edges incrementally, ensuring no cycles. This process continue until all vertices are connected, forming a minimum spanning tree. The algorithm's efficiency stems from its greedy nature and ability to hand disconnected graphs.

Output :-

Following are the edges in the constructed

2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10

Minimum cost spanning Tree : 19

pioneerpaper.in
PAGE NO. 17

Travelling Salesman Problem

C++

```
#include <iostream>
using namespace std;
const int n=4;
const int MAX=1000000;
int dist[n+1][n+1]={
{0,0,0,0,0},{0,0,10,15,20},{0,10,0,25,25},
{0,15,25,0,3},{0,20,25,30,0}};
int memo[n+1][1<<(n+1)];
int fun(int i, int mask){
    if (mask==((1<<i)|3))
      return dist[1][i];
    if (memo[i][mask]!=0)
      return memo[i][mask];

    int res=MAX;
    for (int j=1; j<=n; j++)
      if ((mask&(1<<j))&& j!=i && j!=1)
        res=std::min(res, fun(j, mask&(~(1<<i)))
            + dist[j][i]);
    return memo[i][mask]=res;
}
```

```
int main () {
    int ans = MAX;
    for (int i=1 ; i<=n; i++)
        ans = std:: min (ans, sum (i,(1 << (n+1)) -1)
                + dist [i] [1] );
    printf ("The cost of most efficent tour = %d",
            ans);
    return 0;
}
```

Algorithm:-

> The algorithm aims to find the sotest root that a set of nodes and return to start. it is a combinational optimization problem with exponential complexity. Various algorithm, including brute-force, dynamic Programming & heurists approaches like the nearest neighbor & genetic algorithms, are used to find near-option solution for TSP, with application in logistic, router planning & circuit design.

Output :-

The cost of most efficient tour = 80

DATE

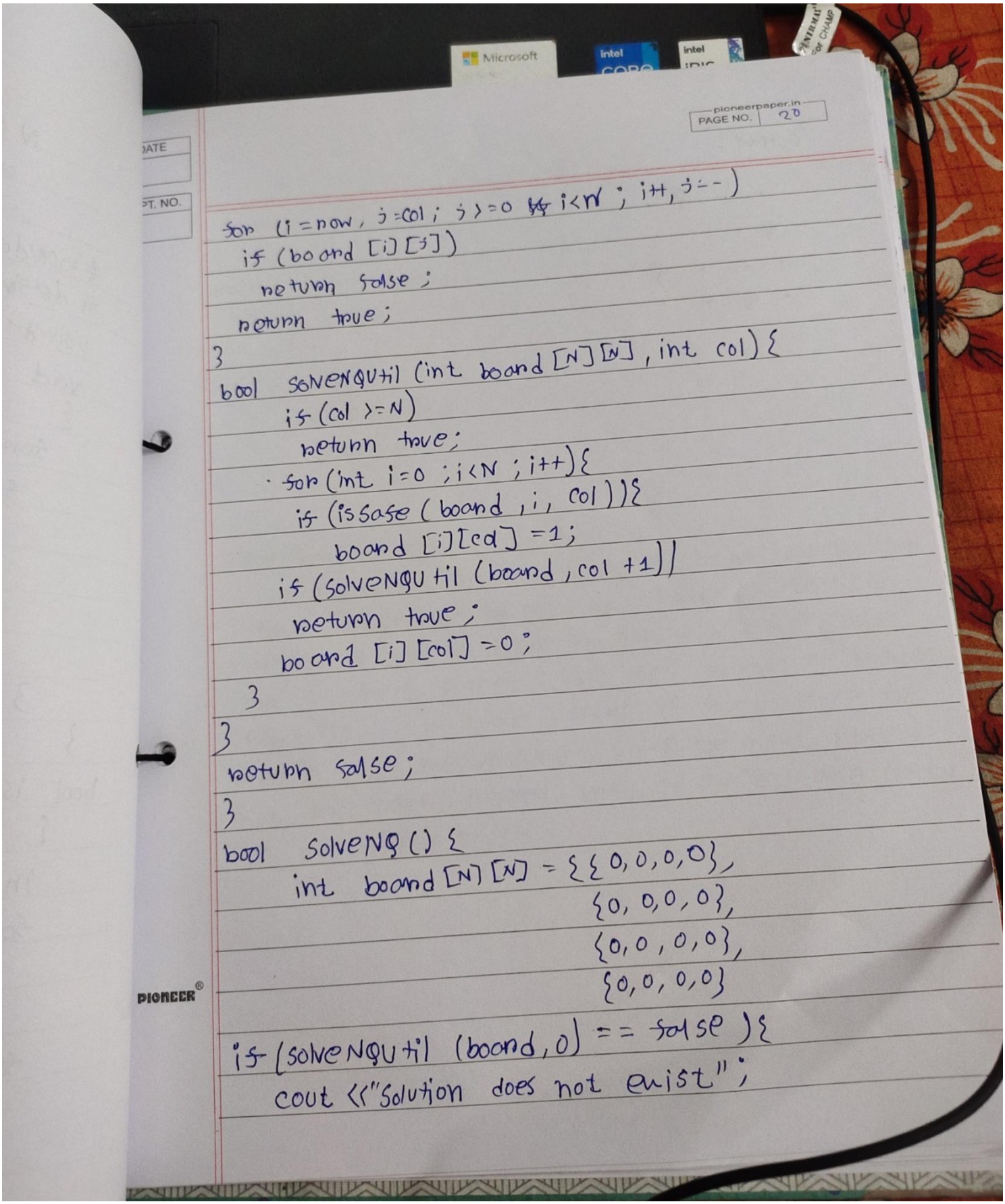
PT. NO.

pioneerpaper.in
PAGE NO. 19

N Queen

C++

```
#include <bits/stdc++.h>
#define N 4
using namespace std;
void printsolution (int board[N][N])
{
  for (int i=0; i<N; i++){
    for(int j=0; j<N; j++)
      if (board[i][j])
        cout << "Q";
        else cout << ".";
        printf("\n");
  }
}
bool issafe (int board[N][N], int row, int col)
  {
  int i, j;
  for (i=0; i<col; i++)
    if (board[row][i])
      return false;
  for (i=row, j=col; i>=0 && j>=0; i--, j--)
    if (board[i][j])
      return false;
```



```
for (i=row, j=col; j>=0 && i<N; i++, j--)
  if (board[i][j])
    return false;
  return true;
}
bool solveNQUtil (int board[N][N], int col){
    if (col >= N)
      return true;
   for (int i=0; i<N; i++){
     if (isSafe (board, i, col)){
        board[i][col] = 1;
     if (solveNQUtil (board, col+1))
      return true;
      board[i][col] = 0;
   }
}
return false;
}
bool solveNQ (){
    int board[N][N] = {{0,0,0,0},
                       {0,0,0,0},
                       {0,0,0,0},
                       {0,0,0,0}};
if (solveNQUtil (board,0) == false){
   cout <<"Solution does not exist";
```

```
    return false;
}
printSolution(board);
    return true;
}
int main() {
        solveNQ();
        return 0;
}
```

=

Algorithm:

> The N queens problem is solved using backtracking. Place queens one by one, backtrack if conflict arises. Define recursive function with base case for N queens placed. Use loops to try all posible positions. If a safe spot is found place the queen and recursively solve the next row. If not, backtrack. Repeat until all queens are placed or no solution exists.

OUTPUT :-

. . ♀ .
♀ . . .
. . . ♀
. ♀ . .

DATE

EXPT. NO.

pioneerpaper.in
PAGE NO. 22

Bellman ford
c++

```
#include <bits/stdc++.h>
using namespace std;
struct Edge {
    int src, dest, weight;
};
struct Graph {
    int V, E;
    struct Edge* edge;
};
struct Graph* createGraph (int V, int E) {
    struct Graph* graph = new Graph;
    graph -> V = V;
    graph -> E = E;
    graph -> edge = new Edge [E];
    return graph;
}
void printArr (int dist[], int n) {
    printf ("Vertex Distance from Source \n");
    for (int i = 0; i < n; ++i)
        printf ("%d \t\t %d \n", i, dist[i]);
}
```

PIONEER®

```
void BellmonFord (struct Graph* graph , int src) {
    int V = graph -> V;
    int E = graph -> E;
    int dist [V];
    for (int i=0 ; i<V; i++)
      dist [i] = INT_MAX ;
      dist [src] = 0;
    for (int i=1; i<= V -1 ;i++){
      for (int j=0 ; j<E ; j++) {
        int u = graph -> edge [j] .src ;
        int v = graph -> edge [j] . dest;
        int weight = graph -> edge [j] . weight;
        if (dist[u] != INT_MAX && dist [u] + weight
            <dist [v])
          dist[v] = dist[u] + weight ;
      }
    }
  for (int i=0 ; i<E ;i++) {
   int u = graph -> edge [i] . src;
   int v = graph -> edge [i] . dest;
   int weight = graph -> edge [i]. weight ;
   if (dist [u] != INT_MAX && dist[u] + weight < dist[v]){
     printf ("Graph contains negative weight cycle");
     return;
   }
}
```

```
    printArr (dist, V);
    return ;
}
int main ()
{
  int V = 5 ;
  int E = 8 ;
  struct Graph* graph = createGraph (V, E);

    graph -> edge [0] . src = 0;
    graph -> edge [0] . dest = 1;
    graph -> edge [0] . weight = -1;

    graph -> edge [1] . src = 0;
    graph -> edge [1] . dest = 2;
    graph -> edge [1] . weight = 4;

    graph -> edge [2] . src = 1 ;
    graph -> edge [2] . dest = 2;
    graph -> edge [2] . weight = 3;

    graph -> edge [3] . src = 1 ;
    graph -> edge [3] . dest = 3 ;
    graph -> edge [3] . weight = 2;
```

```
graph -> edge[4].src = 1;
graph -> edge[4].dest = 4;
graph -> edge[4].weight = 2;

graph -> edge[5].src = 3;
graph -> edge[5].dest = 2;
graph -> edge[5].weight = 5;

graph -> edge[6].src = 3;
graph -> edge[6].dest = 1;
graph -> edge[6].weight = 1;

graph -> edge[7].src = 4;
graph -> edge[7].dest = 3;
graph -> edge[7].weight = -3;

BellmanFord(graph, 0);
return 0;
}
```

Vertex	Distance from Source
0	0
1	-1
2	2
3	-2
4	1

DATE

PT. NO.

pioneerpaper.in
PAGE NO. 28

Algorithm :-

Bellman Ford is a single-source shortest path algorithm that handles negative edge weights. It iterates through all edges, relaxing them by updating distance estimates. The process repeats V-1 times, where V is the number of vertices. If a shorter path is found, the algorithm updates the distance. Negative cycles can be detected if further updates occur in the Vth iteration, indicating no shortest path exists.

Quick Sort

c++

```
#include <bits/stdc++.h>
using namespace std;
int Partition (int arr[], int low, int high){
    int pivot = arr[high];
    int i = (low-);
    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);
    }
}
```

```
int main(){
    int arr[]={10,7,8,9,1,5};
    int n=sizeof(arr)/sizeof(arr[0]);
    quickSort(arr,0,n-1);
    cout<<"Sorted Array\n";
    for(int i=0;i<n;i++)
    {
      cout<<arr[i]<<" ";
    }
    return;
}
```

Algorithm :-

QuickSort is a divide-and-conquer sorting algorithm. it selects a "pivot" elements, partitions the array into two sub array. elements smaller than the pivot and elements greater. Recursively repeat this process on sub-arrays. Efficient average case performance, O(nlogn); but worst-case is O(n²) if poorly chosen pivots.

Sorted array:

1 5 7 8 9 10

pioneerpaper.in
PAGE NO. 22

BFS

C+

```
#include <bits/stdc++.h>
using namespace std;
class Graph {
    int V;
    vector<list<int>> adj;
public:
    Graph(int V);
    void addEdge(int v, int w);
    void BFS(int s);
};
Graph::Graph(int V){
    this->V = V;
    adj.resize(V);
}
void Graph::addEdge(int v, int w){
    adj[v].push_back(w);
}
void Graph::BFS(int s){
    vector<bool> visited;
    visited.resize(V, false);
    list<int> queue;
    visited[s] = true;
```

```
    queue.push_back(s);
    while (!queue.empty()){
        s=queue.front();
        cout<<s<<" ";
        queue.pop_front();
    for(auto adjacent:adj[s]){
        if(!visited[adjacent]){
            visited[adjacent]=true;
            queue.push_back(adjacent);
        }
    }
}
}
int main(){
    Graph g(4);
    g.addEdge(0,1);
    g.addEdge(0,2);
    g.addEdge(1,2);
    g.addEdge(2,0);
    g.addEdge(2,3);
    g.addEdge(3,3);
    cout<<"Following is Breadth First Traversal"<<
        "(starting from vertex 2)\n";
    g.BFS(2);
    return 0;
}
```

Output:-

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

pioneerpaper.in
PAGE NO. 31

Algorithm:

> choose a pivot from the array.
> Partition the array into 2-sub-array elements smaller than the pivot & elements greater than the pivot.
> Recursively apply quick sort to sub array.
> Contive. sorted array.

> BFS explores a graph level by level, starting from a selected node. it visits neighbors before moving deeper. Utilizing a queue, BFS ensures all nodes at the current depth are processed before moving to the next. This guarantees the shortest path is found in an unweighted graph.

=

DFS

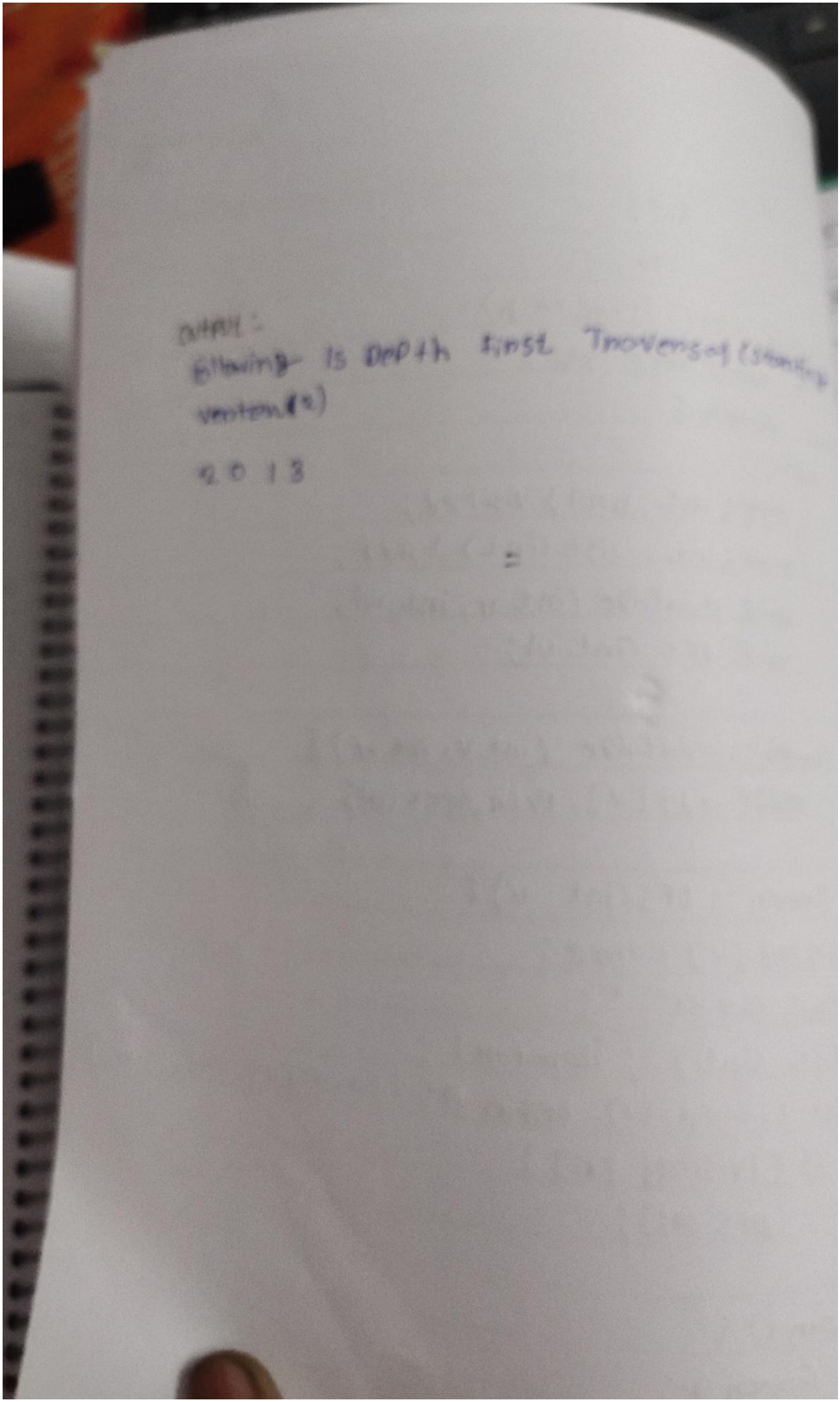
C++

```
#include <bits/stdc++.h>
using namespace std;
class Graph {
  public:
    map<int, bool> visited;
    map<int, list<int>> adj;
    void addEdge (int v, int w);
    void DFS (int v);
};
void Graph::addEdge (int v, int w) {
    adj[v].push_back(w);
}
void Graph::DFS(int v) {
    visited[v] = true;
    cout << v << " ";
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
      if (!visited[*i])
        DFS(*i);
}
int main() {
    Graph g;
    g.addEdge(0, 1);
```

```
g.addEdge(0,2);
g.addEdge(1,2);
g.addEdge(2,0);
g.addEdge(2,3)
g.addEdge(3,3);
Cout<<"Following is Depth First Traversal"
    "(Starting from vertex 2)\n";
g.DFS(2);
return 0;
}
```

Algorithm :-

> Start with initial node and enqueue it.
> Deque a node, visit it & enquing it with nbors.
> Repeat until queue empty.
> Traverse the path in an undirected graph.


OUTPUT:
Following is Depth First Traversal (starting
vertex 2)
2 0 1 3

pioneerpaper.in
PAGE NO. 34

DATE

EXPT. NO.

Dijkstra's Algorithm

Algorithm:-

Dijkstra's algorithm finds the shortest path in weighted graph. It starts from the source node, explores neighbors, and updates their distances. The process continues until the destination is reached. Priority queues manage node priorities. The algorithm ensures the shortest paths are consistently updated.

Code:- C++

```
#include <iostream>
using namespace std;
#include <limits.h>
#define V 9

int minDistance (int dist[], bool sptSet[])
{
int min = INT_MAX, min_index;
  for (int v=0; v<V; v++)
    if (sptSet[v] == false && dist[v] <= min)
      min = dist[v], min_index = v;
```

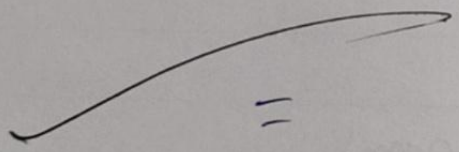
PIONEER®

```
    return min_index;
}

void printSolution(int dist[])
{
  cout << "Vertex \t Distance from Source" << endl;
  for (int i=0; i<V; i++)
     cout << i << " \t\t\t " << dist[i] << endl
}
void dijkstra(int graph[V][V], int src)
{
  int dist[V];
  bool sptSet[V];
  for (int i=0 ;i<V; i++)
     dist[i] = INT_MAX, sptSet[i] = false;
     dist[src] = 0;
     for (int count = 0; count < V-1; count++){
      int u = minDistance(dist, sptSet
       sptSet[u] = true;
       for (int v=0; v<V; v++)
          if (!sptSet[v] && graph[u][v] && dist[u]!=
          INT_MAX && dist[u] + graph[u][v] < dist[v])
          dist[v] = dist[u] + graph[u][v];
}
  printSolution(dist);
}
```

Output :-

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14



```
int main(){
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                       {4, 0, 8, 0, 0, 0, 11, 0},
                       {0, 8, 0, 7, 0, 4, 0, 0, 2},
                       {0, 0, 7, 0, 9, 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}};
    dijkstra(graph, 0);
    return 0;
}
```

27/11/23