**GURU NANAK DEV ENGINEERING COLLEGE, LUDHIANA**

**DEPARTMENT OF IT**

Logo

Description automatically generated

**DESIGN AND ANALYSIS PF ALGORITHMS LABORATORY**

**LPCIT - 113**

**Submitted to: - Submitted by:-**

**Pf. Sachin Bagga Palakpreet Kaur**

**IT B2**

**U.R.N: - 2004836**

**C.R.N: - 2021144**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.NO.** | **CONTENT** | **PAGE NO.** | **SIGN** |
| 1 | Implement binary search algorithm and compute its time complexity. | 3-5 |  |
| 2 | Implement merge sort algorithm and demonstrate divide and conquer technique. | 6-9 |  |
| 3 | Analyse the time complexity of Quick-sort algorithm. | 10-12 |  |
| 4 | Solve minimum-cost spanning tree problem using greedy method. [Prims Algorithm], [Kruskal’s Algorithm] | 13-20 |  |
| 5 | Implement greedy algorithm to solve single-source shortest path problem. | 21-24 |  |
| 6 | Use dynamic programming to solve Knapsack problem. | 25-27 |  |
| 7 | Solve all pairs shortest path problem using dynamic programming. [Floyd Warshall Algorithm] | 28-29 |  |
| 8 | Use backtracking to solve 8-queens’ problem. | 30-32 |  |
| 9 | Solve sum of subsets problem using backtracking. | 33-36 |  |
| 10 | Implement Boyer-Moore algorithm. | 37-40 |  |

**Practical no. – 1**

**Aim -** Implement binary search algorithm and compute its time complexity.

**Binary Search:**

Binary search is a very fast and efficient searching technique. It requires the list to be in sorted order. In this method, to search an element you can compare it with the present element at the centre of the list. If it matches, then the search is successful otherwise the list is divided into two halves: one from the 0th element to the middle element which is the centre element (first half) another from the centre element to the last element (which is the 2nd half) where all values are greater than the centre element.

The searching mechanism proceeds from either of the two halves depending upon whether the target element is greater or smaller than the central element. If the element is smaller than the central element, then searching is done in the first half, otherwise searching is done in the second half.

**Algorithm:**

def binary\_search(arr, target):

low = 0

high = len(arr) - 1

while low <= high:

mid = (low + high) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

low = mid + 1

else:

high = mid - 1

return -1 # target not found

**Program :-**

#include <iostream>

using namespace std;

int binarySearch(int arr[], int left, int right, int x)

{

    while (left <= right)

    {

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

        if (arr[mid] == x)

        {

            return mid;

        }

        else if (arr[mid] < x)

        {

            left = mid + 1;

        }

        else

        {

            right = mid - 1;

        }

    }

    return -1;

}

int main()

{

    int myarr[10];

    int num;

    int output;

 cout << "Palakpreet kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

cout << " Enter 10 elements ASCENDING order" << endl;

    for (int i = 0; i < 10; i++)

    {

        cin >> myarr[i];

    }

    cout << "Enter an element to search" << endl;

    cin >> num;

    output = binarySearch(myarr, 0, 9, num);

    if (output == -1)

    {

        cout << "No Match Found" << endl;

    }

    else

    {

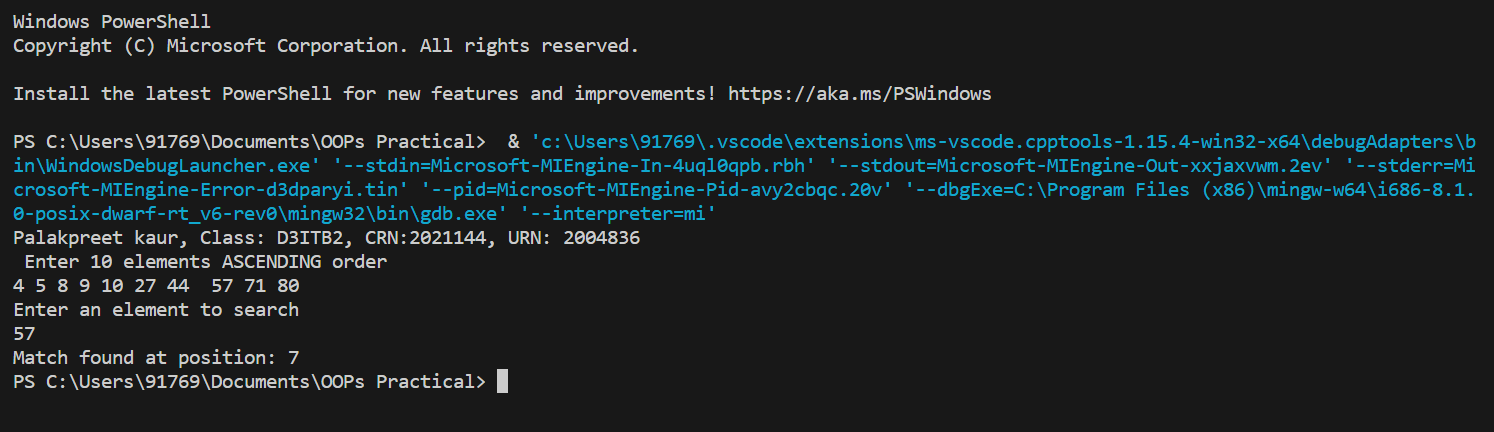
        cout << "Match found at position: " << output << endl;

    }

    return 0;

}

**Output –**

****

**Analysis of Binary Search –**

At Iteration 2:

Length of array = n/2

At Iteration 3:

Length of array = (n/2)/2 = n/22

Therefore, after Iteration k:

Length of array = n/2k

Also, we know that after k iterations, the length of the array becomes 1 Therefore, the Length of the array

n/2k = 1

=> n = 2k

Applying log function on both sides:

=> log2n = log2 2k

=> log2n = k \* log2 2

As (loga (a) = 1) Therefore, k = log2 (n)

**Practical no. – 2**

**Aim -** Implement merge sort algorithm and demonstrate divide and conquer technique.

**Merge Sort:**

Merge Sort is a divide-and-conquer sorting algorithm that recursively divides the input list into smaller sub-lists, sorts them, and merges them back into a sorted list. The time complexity of Merge Sort is O(n log n), making it efficient for large data sets.

**Algorithm:**

MergeSort(arr, left, right):

if left > right

return

mid = (left+right)/2

mergeSort(arr, left, mid)

mergeSort(arr, mid+1, right)

merge(arr, left, mid, right)

end

**Program -**-

#include<iostream>

using namespace std;

void merge(int arr[], int l, int m, int r)

{

    int i = l;

    int j = m + 1;

    int k = l;

/\* create temp array \*/

    int temp[5];

    while (i <= m && j <= r)

    {

        if (arr[i] <= arr[j])

        {

            temp[k] = arr[i];

            i++;

            k++;

        }

        else

        {

            temp[k] = arr[j];

            j++;

            k++;

        }

    }

    /\* Copy the remaining elements of first half, if there are any \*/

    while (i <= m)

    {

        temp[k] = arr[i];

        i++;

        k++;

    }

    /\* Copy the remaining elements of second half, if there are any \*/

    while (j <= r)

    {

        temp[k] = arr[j];

        j++;

        k++;

    }

    /\* Copy the temp array to original array \*/

    for (int p = l; p <= r; p++)

    {

        arr[p] = temp[p];

    }

}

/\* l is for left index and r is right index of the

   sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

    if (l < r)

    {

        // find midpoint

        int m = (l + r) / 2;

        // recurcive mergesort first and second halves

        mergeSort(arr, l, m);

        mergeSort(arr, m + 1, r);

        // merge

        merge(arr, l, m, r);

    }

}

int main()

{

    int myarray[5];

    // int arr\_size = sizeof(myarray)/sizeof(myarray[0]);

    int arr\_size = 5;

cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

 cout << "Enter 5 integers in any order: " << endl;

    for (int i = 0; i < 5; i++)

    {

        cin >> myarray[i];

    }

    cout << "Before Sorting" << endl;

    for (int i = 0; i < 5; i++)

    {

        cout << myarray[i] << " ";

    }

    cout << endl;

    mergeSort(myarray, 0, (arr\_size - 1)); // mergesort(arr,left,right) called

    cout << "After Sorting" << endl;

    for (int i = 0; i < 5; i++)

    {

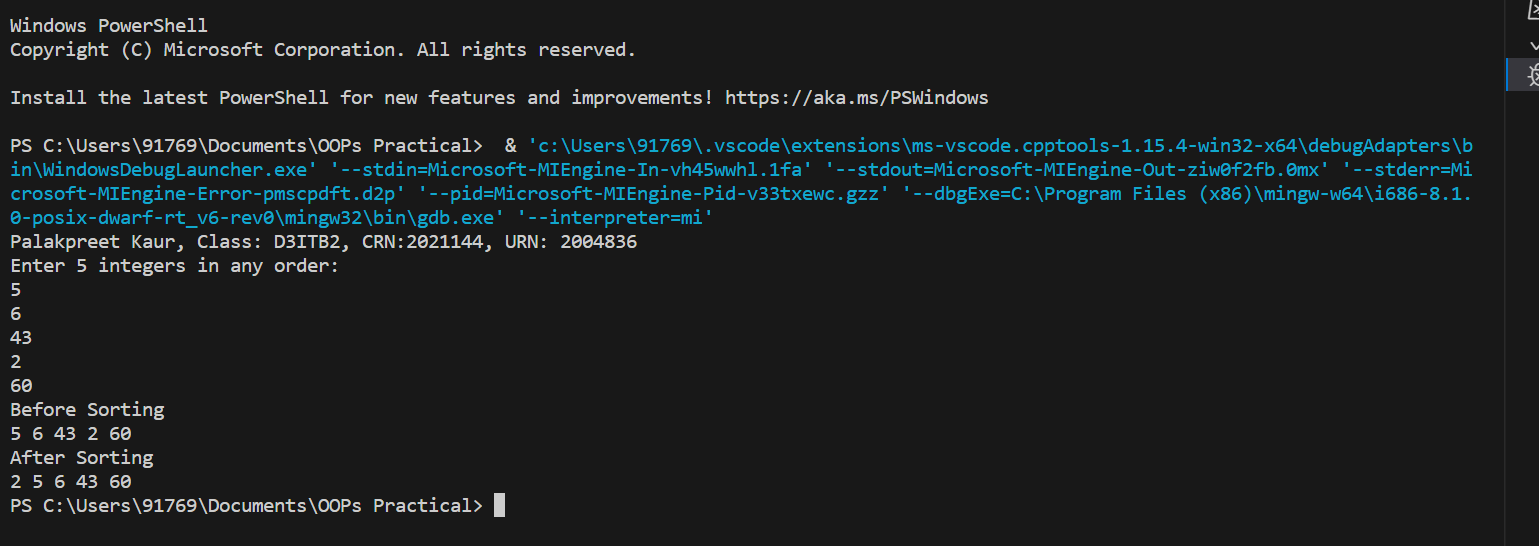
        cout << myarray[i] << " ";

    }

    return 0;

}

**Output –**

****

**Analysis of Merge Sort:**

Let us consider, the running time of Merge-Sort as T(n). Hence,

T(n)= { c if n⩽1

2xT(n/2) + dxn otherwise where c and d are constants

Therefore, using this recurrence relation,

T(n)=2iT(n2i) + i.d.n

As, i=logn,T(n)=2lognT(n2logn) + logn.d.n = c.n + d.n.logn

Therefore, T(n)=O(nlogn)

**Practical no. – 3**

Aim - Analyse the time complexity of Quick-sort algorithm.

**Quick Sort** -

Quick Sort is a divide-and-conquer sorting algorithm that operates by partitioning an array into two sub-arrays, one containing element smaller than a chosen pivot element, and one containing element larger than the pivot. The two sub-arrays are then recursively sorted until the entire array is sorted. Its time complexity is O(n log n) on average, but can degrade to O(n^2) in the worst case, and its space complexity is O(log n) due to the recursion stack.

**Algorithm -**

Quick-Sort (A, p, r)

if p < r then

q Partition (A, p, r)

Quick-Sort (A, p, q)

Quick-Sort (A, q + r, r)

**Program –**

#include <iostream>

using namespace std;

// quick sort sorting algorithm

int Partition(int arr[], int s, int e)

{

    int pivot = arr[e];

    int pIndex = s;

    for (int i = s; i < e; i++)

    {

        if (arr[i] < pivot)

        {

            int temp = arr[i];

            arr[i] = arr[pIndex];

            arr[pIndex] = temp;

            pIndex++;

        }

    }

    int temp = arr[e];

    arr[e] = arr[pIndex];

    arr[pIndex] = temp;

    return pIndex;

}

void QuickSort(int arr[], int s, int e)

{

    if (s < e)

    {

        int p = Partition(arr, s, e);

        QuickSort(arr, s, (p - 1)); // recursive QS call for left partition

        QuickSort(arr, (p + 1), e); // recursive QS call for right partition

    }

}

int main()

{

cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;    int size = 0;

    cout << "Enter Size of array: " << endl;

    cin >> size;

    int myarray[size];

    cout << "Enter " << size << " integers in any order: " << endl;

    for (int i = 0; i < size; i++)

    {

        cin >> myarray[i];

    }

    cout << "Before Sorting" << endl;

    for (int i = 0; i < size; i++)

    {

        cout << myarray[i] << " ";

    }

    cout << endl;

    QuickSort(myarray, 0, (size - 1)); // quick sort called

    cout << "After Sorting" << endl;

    for (int i = 0; i < size; i++)

    {

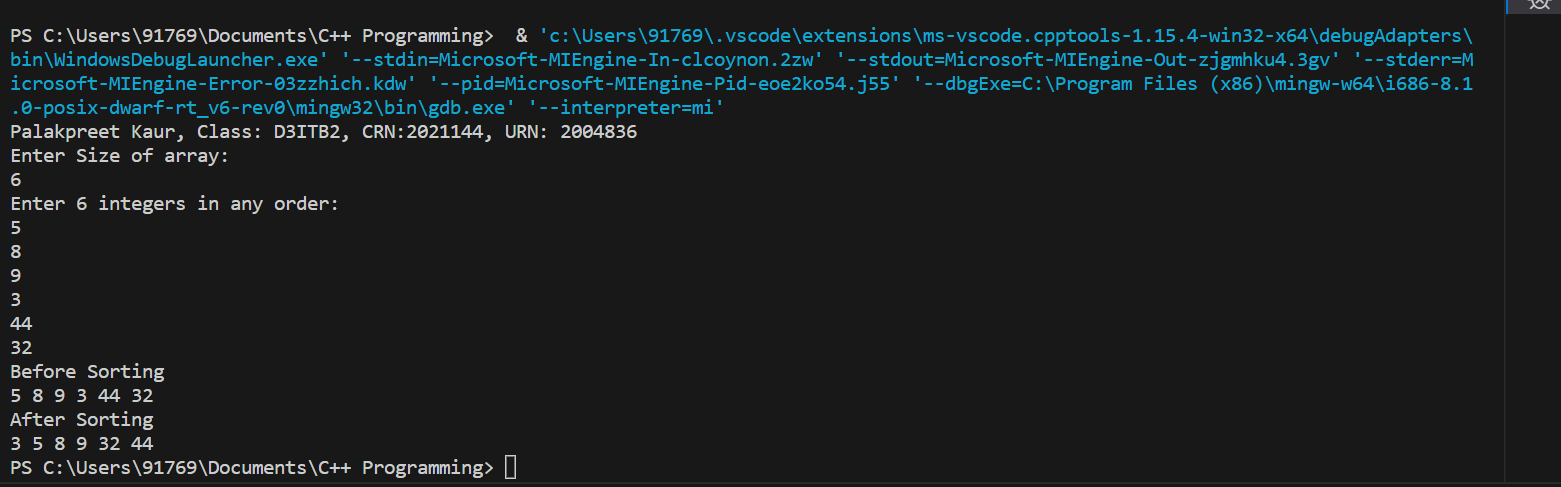
        cout << myarray[i] << " ";

    }

    return 0;

}

**Output:-**

****

**Analysis Of Quick Sort:**

Time taken by Quicksort, in general, can be written as follows.

T(n) = T(k) + T(n-k-1) + θ (n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.

The time taken by Quicksort depends upon the input array and partition strategy. Following are three cases.

**Worst Case:**

The worst case occurs when the partition process always picks the greatest or smallest element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for the worst case.

T(n) = T(0) + T(n-1) + θ (n) which is equivalent to T(n) = T(n-1) + θ (n)

**The solution to the above recurrence is  (n2).**

**Best Case:**

The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

T(n) = 2T(n/2) + θ (n)

**The solution for the above recurrence is (n log n).**

**Average Case:**

Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + θ (n)

**The solution of above recurrence is also O(n log n).**

**Practical no. – 4(a)**

**Aim -** Solve minimum-cost spanning tree problem using greedy method. [Prims Algorithm]

**Prim’s Algorithm:**

Prim's algorithm is widely used as a Greedy algorithm that helps discover the most miniature spanning tree for a weighted undirected graph. This algorithm tends to search the subgroup of the edges that can construct a tree, and the complete poundage of all the edges in the tree should be minimal.

Prim's algorithm has the property that the edges in the set T, which contains the edges of the minimum spanning tree when the algorithm proceeds step-by-step, always form a single tree. That is, at each step, there is only one connected component.

**Prim's Algorithm Steps**

The following steps are as follows:

* Begin with one starting vertex (say v) of a given graph G(V, E).
* Then, choose a minimum weight edge (u, v) connecting vertex v in set A to the vertices in the set (V-A) in each iteration. We always find an edge (u, v) of minimum weight, such as v ∈ A and u ∈ V-A. Then we modify the set A by adding u, A ← A ∪ {u}.
* The process is repeated until ≠V, that is, until all the vertices are not in the set A.

**Algorithm:**

**MST-PRIM (G, w, r)**

for each u ∈ V [G]

do key [u] ← ∞

π [u] ← NIL

key [r] ← 0

Q ← V [G]

While Q ? ∅

do u ← EXTRACT - MIN (Q)

for each v ∈ Adj [u]

do if v ∈ Q and w (u, v) < key [v]

then π [v] ← u

key [v] ← w (u, v)

**Program:**

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 5

// A utility function to find the vertex with

// minimum key value, from the set of vertices

// not yet included in MST

int minKey(int key[], bool mstSet[])

{

    // Initialize min value

    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;

}

// A utility function to print the

// constructed MST stored in parent[]

void printMST(int parent[], int graph[V][V])

{

    cout << "Edge \tWeight\n";

    for (int i = 1; i < V; i++)

        cout << parent[i] << " - " << i << " \t"

            << graph[i][parent[i]] << " \n";}

// Function to construct and print MST for

// a graph represented using adjacency

// matrix representation

void primMST(int graph[V][V])

{

    // Array to store constructed MST

    int parent[V];

    // Key values used to pick minimum weight edge in cut

    int key[V];

    // To represent set of vertices included in MST

    bool mstSet[V];

    // Initialize all keys as INFINITE

    for (int i = 0; i < V; i++)

        key[i] = INT\_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.

    // Make key 0 so that this vertex is picked as first

    // vertex.

    key[0] = 0;

    // First node is always root of MST

    parent[0] = -1;

    for (int count = 0; count < V - 1; count++)

{   // Pick the minimum key vertex from the

        // set of vertices not yet included in MST

        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set

        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];

    }

    // Print the constructed MST

    printMST(parent, graph);

}

// Driver's code

int main()

{

    cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

    int graph[V][V] = { {0, 9, 75, 0, 0},

                        {9, 0, 95, 19, 42},

                        {75, 95, 0, 51, 66},

                        {0, 19, 51, 0, 31},

                        {0, 42, 66, 31, 0 } };

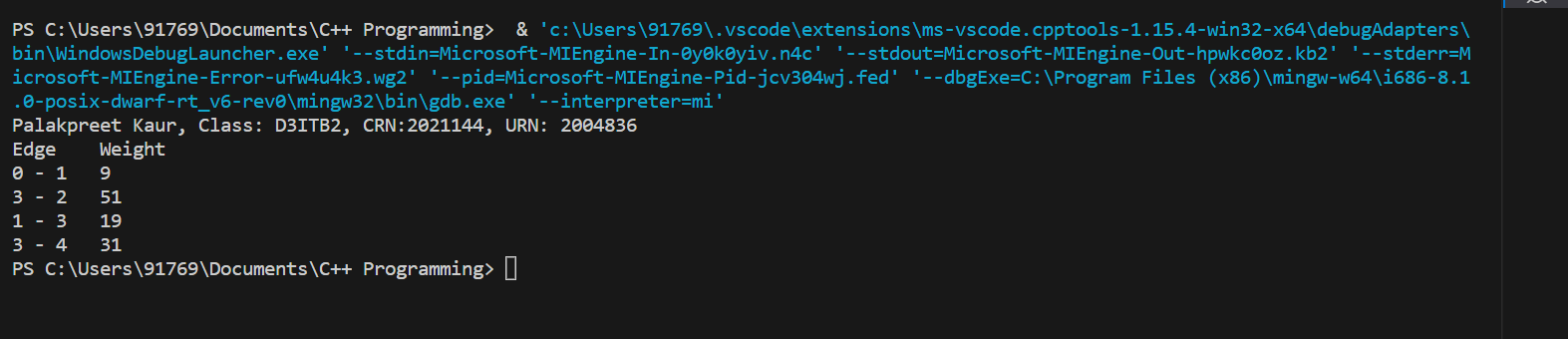
    // Print the solution

    primMST(graph);

    return 0;

}

**Output:**

****

**Analysis of Prim’s Algorithm:**

**The analysis of prim's algorithm by using min-heap is as follows.**

For constructing heap: O(V)

The loop executes |V| times, and extracting the minimum element from the min-heap takes log V time, so while loop takes VlogV.

Total 'E' decrease key operations. Hence takes E log V

**(Vlog V+E log V) = (V+ E) log V ≈ Elog V**

The prim's algorithm time complexity using the Fibonacci heap is **O(E + V log V).**

The prim's algorithm time complexity using the binomial heap is **O(V + E).**

**Practical no. – 4(b)**

**Aim -** Solve minimum-cost spanning tree problem using greedy method. [Kruskal’s Algorithm]

**Kruskal’s Algorithm:**

Kruskal Algorithm generates the minimum spanning tree, initiating from the smallest weighted edge. Kruskal algorithm is developed for discovering the minimum spanning tree of a graph. The Kruskal algorithms are popular and follow different steps to solve the same kind of problem. This algorithm does not consider the larger problem as a whole, but the optimal solution for each smaller instance will provide an immediate output.

**Working:**

* First, examine the edges of G in the order of increasing weight.
* Then select an edge (u, v) є E of minimum weight and check whether its endpoints belong to the same or different connected components.
* If u and v belong to different connected components, we add them to set A. Otherwise, it is rejected because it can create a cycle.
* The algorithm terminates when only one connected component remains (that is, all the vertices of G have been reached).

**Algorithm:**

**MST- KRUSKAL (G, w)**

A ← ∅

for each vertex v ∈ V [G]

do MAKE - SET (v)

sort the edges of E into non decreasing order by weight w

for each edge (u, v) ∈ E, taken in non-decreasing order by weight

do if FIND-SET (μ) ≠ if FIND-SET (v)

then A ← A ∪ {(u, v)}

UNION (u, v)

return A

**Program:**

// Kruskal's algorithm in C++

#include <algorithm>

#include <iostream>

#include <vector>

using namespace std;

#define edge pair<int, int>

class Graph {

   private:

  vector<pair<int, edge> > G;  // graph

  vector<pair<int, edge> > T;  // mst

  int \*parent;

  int V;  // number of vertices/nodes in graph

   public:

  Graph(int V);

  void AddWeightedEdge(int u, int v, int w);

  int find\_set(int i);

  void union\_set(int u, int v);

  void kruskal();

  void print();

};

Graph::Graph(int V) {

  parent = new int[V];

  //i 0 1 2 3 4 5

  //parent[i] 0 1 2 3 4 5

  for (int i = 0; i < V; i++)

    parent[i] = i;

  G.clear();

  T.clear();

}

void Graph::AddWeightedEdge(int u, int v, int w) {

  G.push\_back(make\_pair(w, edge(u, v)));

}

int Graph::find\_set(int i) {

  // If i is the parent of itself

  if (i == parent[i])

    return i;

  else

    // Else if i is not the parent of itself

    // Then i is not the representative of his set,

    // so we recursively call Find on its parent

    return find\_set(parent[i]);

}

void Graph::union\_set(int u, int v) {

  parent[u] = parent[v];

}

void Graph::kruskal() {

  int i, uRep, vRep;

  sort(G.begin(), G.end());  // increasing weight

  for (i = 0; i < G.size(); i++) {

    uRep = find\_set(G[i].second.first);

    vRep = find\_set(G[i].second.second);

    if (uRep != vRep) {

      T.push\_back(G[i]);  // add to tree

      union\_set(uRep, vRep);

    }

  }

}

void Graph::print() {

  cout << "Edge :"

     << " Weight" << endl;

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

    cout << T[i].second.first << " - " << T[i].second.second << " : "

       << T[i].first;

    cout << endl;

  }

}

int main() {

     cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

  Graph g(6);

  g.AddWeightedEdge(0, 1, 4);

  g.AddWeightedEdge(0, 2, 4);

  g.AddWeightedEdge(1, 2, 2);

  g.AddWeightedEdge(1, 0, 4);

  g.AddWeightedEdge(2, 0, 4);

  g.AddWeightedEdge(2, 1, 2);

  g.AddWeightedEdge(2, 3, 3);

  g.AddWeightedEdge(2, 5, 2);

  g.AddWeightedEdge(2, 4, 4);

  g.AddWeightedEdge(3, 2, 3);

  g.AddWeightedEdge(3, 4, 3);

  g.AddWeightedEdge(4, 2, 4);

  g.AddWeightedEdge(4, 3, 3);

  g.AddWeightedEdge(5, 2, 2);

  g.AddWeightedEdge(5, 4, 3);

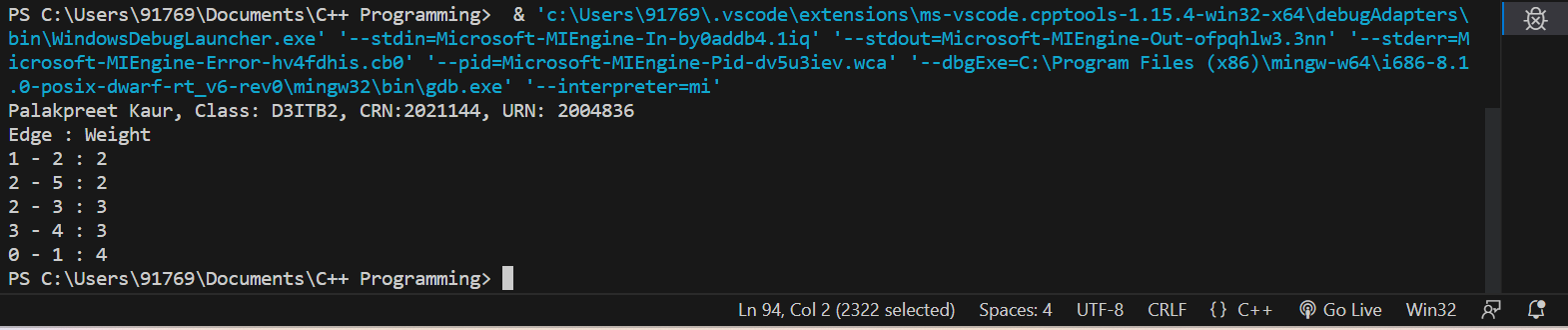
  g.kruskal();

  g.print();

  return 0;

}

**Output:**

****

**Analysis Of Kruskal’s Algorithm:**

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

* E is at most V2 and log V2= 2 x log V is O (log V).
* If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus, the total time is **O (E log E) = O (E log V)**

**Practical no. – 5**

**Aim:** Implement greedy algorithm to solve single-source shortest path problem.

**Dijkstra’s Algorithm:**

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

Dijkstra’s algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.

The graph has the following−

* vertices, or nodes, denoted in the algorithm by v or u.
* weighted edges that connect two nodes: (u,v) denotes an edge, and w(u,v)denotes its weight. In the diagram on the right, the weight for each edge is written in gray.

**Algorithm:**

function dijkstra(G, S)

for each vertex V in G

distance[V] <- infinite

previous[V] <- NULL

If V != S, add V to Priority Queue Q

distance[S] <- 0

while Q IS NOT EMPTY

U <- Extract MIN from Q

for each unvisited neighbour V of U

tempDistance <- distance[U] + edge\_weight(U, V)

if tempDistance < distance[V]

distance[V] <- tempDistance

previous[V] <- U

return distance[], previous[]

**Program:**

#include <iostream>

using namespace std;

int miniDist(int distance[], bool Tset[]) // finding minimum distance

{

    int minimum=INT\_MAX,ind;

    for(int k=0;k<6;k++)

    {

        if(Tset[k]==false && distance[k]<=minimum)

        {

            minimum=distance[k];

            ind=k;

        }

    }

     return ind;

}

void DijkstraAlgo(int graph[6][6],int src) // adjacency matrix

{

    int distance[6]; // // array to calculate the minimum distance for each node

    bool Tset[6];// boolean array to mark visited and unvisited for each node

    for(int k = 0; k<6; k++)

    {

        distance[k] = INT\_MAX;

        Tset[k] = false;

    }

    distance[src] = 0;   // Source vertex distance is set 0

    for(int k = 0; k<6; k++)

    {

        int m=miniDist(distance,Tset);

        Tset[m]=true;

        for(int k = 0; k<6; k++)

        {

            if(!Tset[k] && graph[m][k] && distance[m]!=INT\_MAX &&   distance[m]+graph[m][k]<distance[k])

                distance[k]=distance[m]+graph[m][k];

        }

    }

    cout<<"Vertex\t\tDistance from source vertex"<<endl;

    for(int k = 0; k<6; k++)

    {

      char str=65+k;

        cout<<str<<"\t\t\t"<<distance[k]<<endl;

    }

}

int main()

{   cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

        int graph[6][6]={

        {0, 1, 2, 0, 0, 0},

        {1, 0, 0, 5, 1, 0},

        {2, 0, 0, 2, 3, 0},

        {0, 5, 2, 0, 2, 2},

        {0, 1, 3, 2, 0, 1},

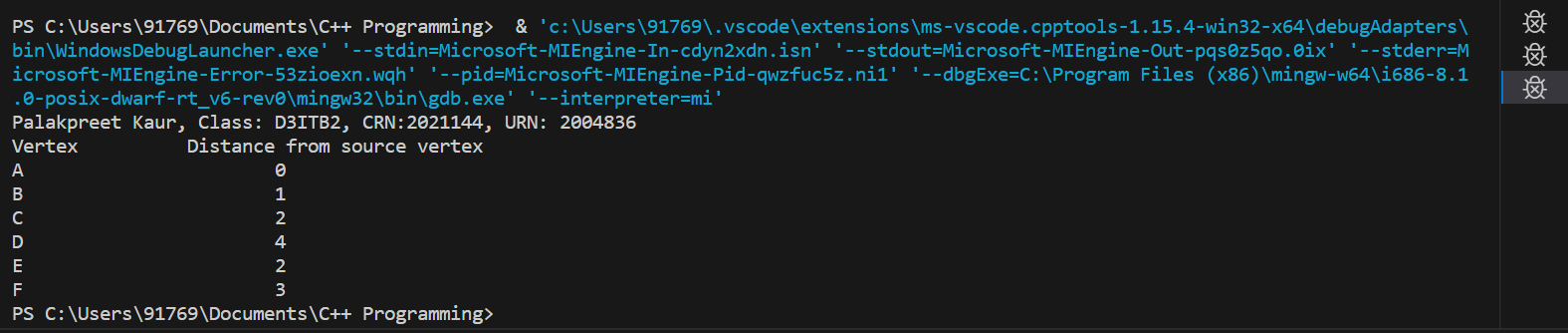
        {0, 0, 0, 2, 1, 0}};

    DijkstraAlgo(graph,0);

    return 0;

}

**Output:**

****

**Analysis Of Dijkstra’s Algorithm:**

Time Complexity: O(E Log V)

where, E is the number of edges and V is the number of vertices.

Complexity analysis for dijkstra's algorithm with adjacency matrix representation of graph.

Time complexity of Dijkstra's algorithm is O(V 2) where V is the number of vertices in the graph.

It can be explained as below:

* First thing we need to do is find the unvisited vertex with the smallest path. For that we require O(V) time as we need check all the vertices.
* Now for each vertex selected as above, we need to relax its neighbours which means to update each neighbours path to the smaller value between its current path or to the newly found. The time required to relax one neighbour comes out to be of order of O(1) (constant time).
* For each vertex we need to relax all of its neighbours, and a vertex can have at most V-1 neighbours, so the time required to update all neighbours of a vertex comes out to be [O(V) \* O(1)] = O(V)

So now following the above conditions, we get:

Time for visiting all vertices =O(V)

Time required for processing one vertex=O(V)

Time required for visiting and processing all the vertices =O(V)∗O(V)=**O(V 2)**

So the time complexity of Dijkstra’s algorithm using adjacency matrix representation comes out to be O(V 2).

The time complexity of Dijkstra’s algorithm can be reduced to **O((V+E)logV)** using adjacency list representation of the graph and a min-heap to store the unvisited vertices, where E is the number of edges in the graph and V is the number of vertices in the graph.

**Practical no. – 6**

**Aim :** Use dynamic programming to solve Knapsack problem.

**0/1 Knapsack Problem:**

knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

**Algorithm:**

Begin

Input set of items each with a weight and a value

Set knapsack capacity

Create a function that returns maximum of two integers.

Create a function which returns the maximum value that can be put in a knapsack of capacity W.

int knapSack(int W, int w[], int v[], int n)

int i, wt;

int K[n + 1][W + 1]

for i = 0 to n

for wt = 0 to W

if (i == 0 or wt == 0)

Do K[i][wt] = 0

else if (w[i - 1] <= wt)

Compute: K[i][wt] = max(v[i - 1] + K[i - 1][wt - w[i - 1]], K[i -1][wt])

else

K[i][wt] = K[i - 1][wt]

return K[n][W]

Call the function and print.

End

**Program:**

#include <bits/stdc++.h>

using namespace std;

// A utility function that returns

// maximum of two integers

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

// Returns the maximum value that

// can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

    int i, w;

    vector<vector<int> > K(n + 1, vector<int>(W + 1));

    // Build table K[][] in bottom up manner

    for (i = 0; i <= n; i++) {

        for (w = 0; w <= W; w++) {

            if (i == 0 || w == 0)

                K[i][w] = 0;

            else if (wt[i - 1] <= w)

                K[i][w] = max(val[i - 1]

                                + K[i - 1][w - wt[i - 1]],

                            K[i - 1][w]);

            else

                K[i][w] = K[i - 1][w];

        }

    }

    return K[n][W];

}

// Driver Code

int main()

{

    cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

    int profit[] = { 60, 100, 120 };

    int weight[] = { 10, 20, 30 };

    int W = 50;

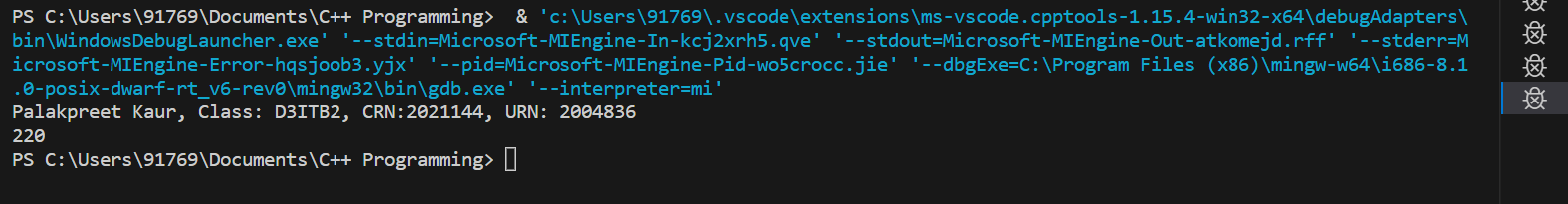
    int n = sizeof(profit) / sizeof(profit[0]);

    cout << knapSack(W, weight, profit, n);

    return 0;

}

**Output:**

****

**Analysis Of 0/1 Knapsack Problem:**

* Each entry of the table requires constant time θ(1) for its computation.
* It takes θ(nw) time to fill (n+1)(w+1) table entries.
* It takes θ(n) time for tracing the solution since tracing process traces the n rows.
* Thus, overall θ(nw) time is taken to solve 0/1 knapsack problem using dynamic programming.

**Practical no. – 7**

**Aim-** Solve all pairs shortest path problem using dynamic programming. [Floyd Warshall Algorithm]

**Floyd-Warshall Algorithm:**

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). This algorithm follows the dynamic programming approach to find the shortest paths.

**Algorithm:**

FLOYD - WARSHALL (W)

n ← rows [W].

D0 ← W

for k ← 1 to n

do for i ← 1 to n

do for j ← 1 to n

do dij(k) ← min (dij(k-1),dik(k-1)+dkj(k-1) )

return D(n)

**Program:**

#include <iostream>

using namespace std;

// defining the number of vertices

#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

// Implementing floyd warshall algorithm

void floydWarshall(int graph[][nV]) {

  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)

    for (j = 0; j < nV; j++)

      matrix[i][j] = graph[i][j];

  // Adding vertices individually

  for (k = 0; k < nV; k++) {

    for (i = 0; i < nV; i++) {

      for (j = 0; j < nV; j++) {

        if (matrix[i][k] + matrix[k][j] < matrix[i][j])

          matrix[i][j] = matrix[i][k] + matrix[k][j];

      }

    }

  }

  printMatrix(matrix);

}

void printMatrix(int matrix[][nV]) {

  for (int i = 0; i < nV; i++) {

    for (int j = 0; j < nV; j++) {

      if (matrix[i][j] == INF)

        printf("%4s", "INF");

      else

        printf("%4d", matrix[i][j]);

    }

    printf("\n");

  }}

int main() {

cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

    int graph[nV][nV] = {{0, 3, INF, 5},

             {2, 0, INF, 4},

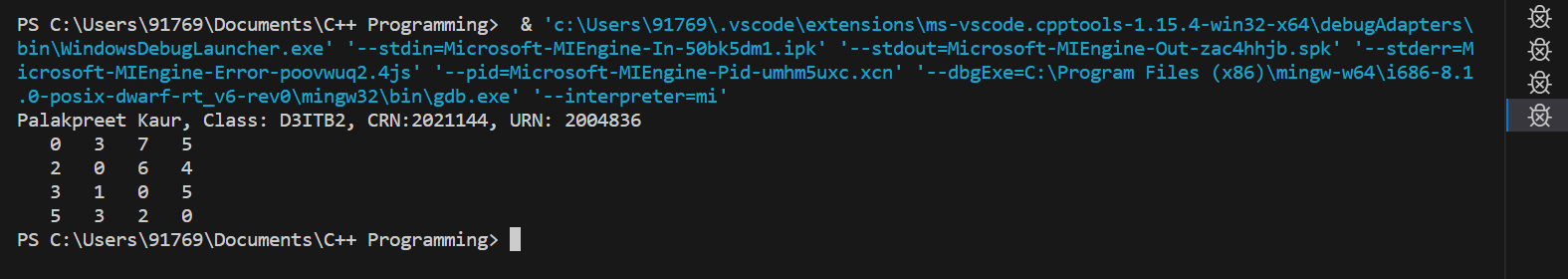
             {INF, 1, 0, INF},

             {INF, INF, 2, 0}};

    floydWarshall(graph);

}

**Output:**

****

**Analysis Of Floyd-Warshall Algorithm:**

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is **O(n 3).**

**Practical no. – 8**

**Aim :** Use backtracking to solve 8-queens’ problem.

**8 – Queen’s Problem:**

N- queen’s is to place n-queens in such a manner on an n\*n chessboard that no queens attack each other by being in the same row, column, or diagonal. It can be seen such that for n=1, the problem has a trivial solution, and no solution exists for n=2 and n=3. So, the 8-queens problem is the problem of placing 8-queens on an 8\*8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal).

**Algorithm:**

START

1. begin from the leftmost column

2. if all the queens are placed,

return true/ print configuration

3. check for all rows in the current column

a) if queen placed safely, mark row and column; and

recursively check if we approach in the current

configuration, do we obtain a solution or not

b) if placing yields a solution, return true

c) if placing does not yield a solution, unmark and

try other rows

4. if all rows tried and solution not obtained, return

false and backtrack

END

**Program:**

#include <bits/stdc++.h>

using namespace std;

int countt=0;

// A function to print a solution

void print(int board[][8]){

    for(int i=0;i<8;i++){

        for(int j=0;j<8;j++){

            cout<<board[i][j]<<" ";

        }

        cout<<endl;

    }

    cout<<"-----------------\n";

}

//Function to check whether a position is valid or not

bool isValid(int board[][8],int row,int col){

    //loop to check horizontal positions

    for(int i=col;i>=0;i--){

        if(board[row][i])

        return false;

    }

    int i=row,j=col;

    //loop to check the upper left diagonal

    while(i>=0&&j>=0){

        if(board[i][j])

        return false;

        i--;

        j--;

    }

    i=row;

    j=col;

    //loop to check the lower left diagonal

    while(i<8&&j>=0){

        if(board[i][j])

        return false;

        i++;

        j--;

    }

    return true;

}

//function to check all the possible solutions

void Queens(int board[][8],int currentColumn){

    if(currentColumn>=8)

    return;

    //loop to cover all the columns

    for(int i=0;i<8;i++){

        if(isValid(board,i,currentColumn)){

            board[i][currentColumn]=1;

            if(currentColumn==7){

                print(board);

                countt++;

            }

            //recursively calling the function

            Queens(board,currentColumn+1);

            //backtracking

            board[i][currentColumn]=0;

        }

    }

}

int main() {

cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

    int board[8][8]={{0,0,0,0,0,0,0,0},

                     {0,0,0,0,0,0,0,0},

                     {0,0,0,0,0,0,0,0},

                     {0,0,0,0,0,0,0,0},

                     {0,0,0,0,0,0,0,0},

                     {0,0,0,0,0,0,0,0},

                     {0,0,0,0,0,0,0,0},

                     {0,0,0,0,0,0,0,0}};

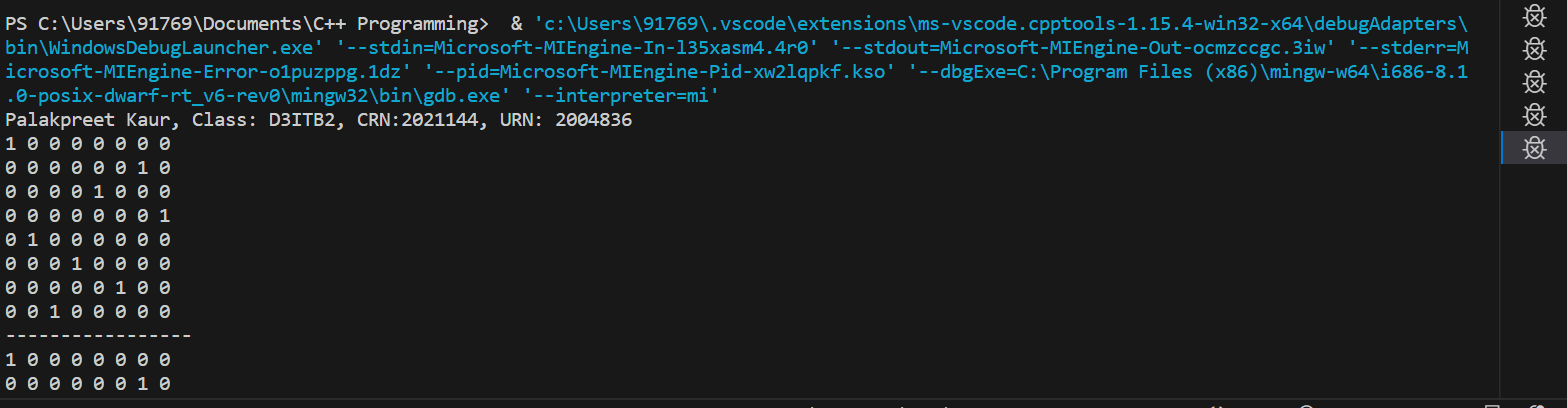
    Queens(board,0);

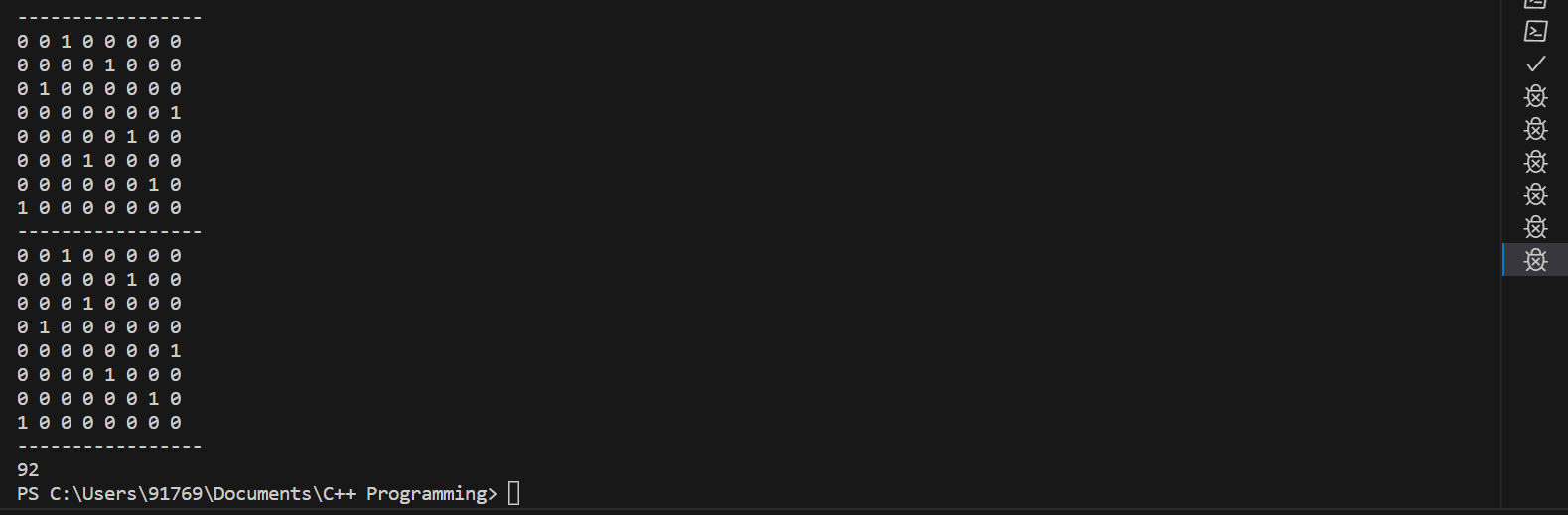
    cout<<countt<<endl;

    return 0;

}

**Output:**

****

****

**Analysis of 8-Queen’s Problem:**

O(N!), For the first column, we will have N choices, then for the next column, we will have N-1 choices, and so on. Therefore the total time taken will be N\*(N-1) \* (N-2)...., which makes the time complexity to be O(N!).

**Practical no. – 9**

**Aim -** Solve sum of subsets problem using backtracking.

**Sum Of Subsets Problem :**

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

**Algorithm:**

void subset\_sum(int list[], int sum, int starting\_index, int target\_sum)

{

if( target\_sum == sum )

{

subset\_count++;

if(starting\_index < list.length)

subset\_sum(list, sum - list[starting\_index-1], starting\_index, target\_sum);

}

else

{

for( int i = starting\_index; i < list.length; i++ )

{

subset\_sum(list, sum + list[i], i + 1, target\_sum);

}

}

}

**Program:**

#include <iostream>

#include <stack>

using namespace std;

int set[] = {10, 5, 18, 20, 2, 15};

int numberOfElements = 6, sum = 25;

class SubSet{

public:

  stack<int> solutionSet;

  bool hasSolution;

  void solve(int s, int idx){

    //return if s exceed sum

    if(s>sum)

        return;

    //check if stack has the right subsets of numbers

    if(s==sum){

        hasSolution = true;

        //display stack contents

        displaySolutionSet();

        //Though found a solution but deliberately

        //returning to find more

        return;

    }

    for(int i=idx; i<numberOfElements; i++){

        //Adding element to the stack

        solutionSet.push(set[i]);

        //add set[i] to the 's' and recusively start from next number

        solve(s+set[i],i+1);

        //Removing element from stack i.e Backtracking

        solutionSet.pop();

    }

  }

  //Function to display stack content

  void displaySolutionSet(){

        stack<int> temp;

        while (!solutionSet.empty())

        {

            cout <<  solutionSet.top() << " ";

            temp.push(solutionSet.top());

            solutionSet.pop();

        }

        cout << '\n';

        while (!temp.empty())

        {

            solutionSet.push(temp.top());

            temp.pop();

        }

    }

};

int main()

{   cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

    SubSet ss;

    ss.solve(0,0);

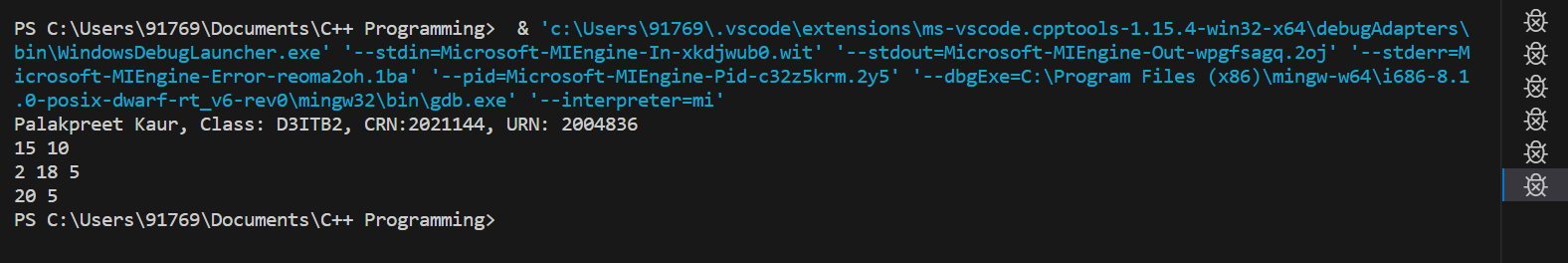
    if(ss.hasSolution == false)

        cout << "No Solution";

    return 0;

}

**Output:**

****

**Analysis Of Sum of Subsets Problem:**

Worst case time complexity for sum of subsets problem is **O(2^n).**

**Practical no. – 10**

**Aim -** Implement Boyer-Moore algorithm.

**Boyer-Moore Algorithm:**

The Boyer-Moore algorithm is a string searching algorithm that is used to find occurrences of a pattern within a larger string. It is considered to be one of the most efficient string searching algorithms. The algorithm works by first pre-processing the pattern to determine a set of rules that can be used to skip over large portions of the text during the search. This pre-processing is done using two techniques: the "bad character" rule and the "good suffix" rule.

* The "bad character" rule works by examining each character in the pattern from right to left and determining the distance to the rightmost occurrence of that character in the pattern. During the search, when a mismatch occurs between the pattern and the text at a particular position, the algorithm uses this distance to determine how far to shift the pattern to the right, such that the next character in the text lines up with the rightmost occurrence of that character in the pattern.
* The "good suffix" rule works by examining each suffix of the pattern and determining the length of the longest suffix that is also a prefix of the pattern. This information is used during the search to determine how far to shift the pattern to the right when a mismatch occurs at a particular position.

By combining these two techniques, the Boyer-Moore algorithm is able to skip over large portions of the text during the search, resulting in faster search times than other string searching algorithms.

**Algorithm:**

COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑)

1. for each character a ∈ ∑

2. do λ [a] = 0

3. for j ← 1 to m

4. do λ [P [j]] ← j

5. Return λ

COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

1. Π ← COMPUTE-PREFIX-FUNCTION (P)

2. P'← reverse (P)

3. Π'← COMPUTE-PREFIX-FUNCTION (P')

4. for j ← 0 to m

5. do ɣ [j] ← m - Π [m]

6. for l ← 1 to m

7. do j ← m - Π' [L]

8. If ɣ [j] > l - Π' [L]

9. then ɣ [j] ← 1 - Π'[L]

10. Return ɣ

BOYER-MOORE-MATCHER (T, P, ∑)

1. n ←length [T]

2. m ←length [P]

3. λ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑)

4. ɣ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

5. s ←0

6. While s ≤ n - m

7. do j ← m

8. While j > 0 and P [j] = T [s + j]

9. do j ←j-1

10. If j = 0

11. then print "Pattern occurs at shift" s

12. s ← s + ɣ [0]

13. else s ← s + max (ɣ [j], j - λ[T[s+j]])

**Program:**

#include <bits/stdc++.h>

using namespace std;

# define NO\_OF\_CHARS 256

void badCharHeuristic( string str, int size, int badchar[NO\_OF\_CHARS])

{   int i;

    // Initialize all occurrences as -1

    for (i = 0; i < NO\_OF\_CHARS; i++)

        badchar[i] = -1;

    // Fill the actual value of last occurrence of a character

    for (i = 0; i < size; i++)

        badchar[(int) str[i]] = i;

}

/\* A pattern searching function that uses Bad

Character Heuristic of Boyer Moore Algorithm \*/

void search( string txt, string pat)

{   int m = pat.size();

    int n = txt.size();

    int badchar[NO\_OF\_CHARS];

    /\* Fill the bad character array by calling

    the preprocessing function badCharHeuristic()

    for given pattern \*/

    badCharHeuristic(pat, m, badchar);

    int s = 0; // s is shift of the pattern with respect to text

    while(s <= (n - m))

    { int j = m - 1;

                   /\* Keep reducing index j of pattern while

        characters of pattern and text are

        matching at this shift s \*/

        while(j >= 0 && pat[j] == txt[s + j])

            j--;

        /\* If the pattern is present at current

        shift, then index j will become -1 after

        the above loop \*/

        if (j < 0)

        {

            cout << "pattern found at position : " << s << endl;

                                     s += (s + m < n)? m-badchar[txt[s + m]] : 1;

        }

        else

            s += max(1, j - badchar[txt[s + j]]);

    }

}

/\* Driver code \*/

int main()

{   cout << "Palakpreet Kaur, Class: D3ITB2, CRN:2021144, URN: 2004836" << endl;

    string txt= "ABAAABCDAABCBBCACCABCDDCBABCD";

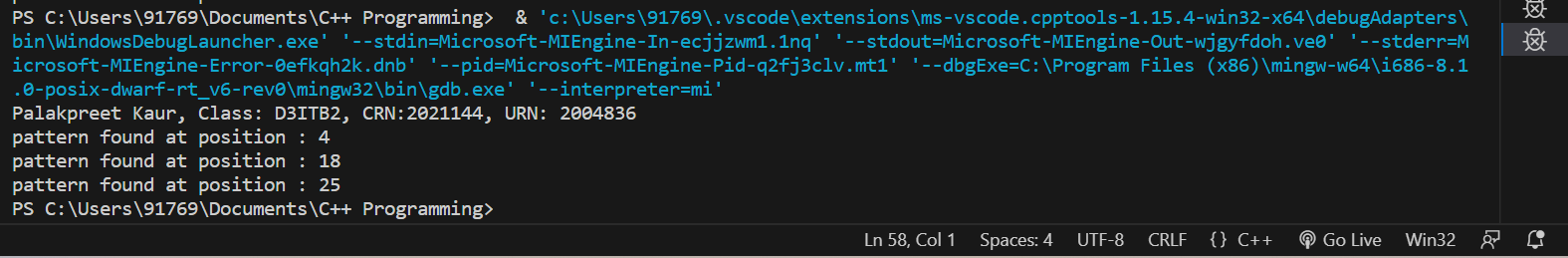
    string pat = "ABCD";

    search(txt, pat);

    return 0;

}

**Output:**

****

**Analysis Of Boyer-Moore Algorithm:**

* The Bad Character Heuristic may take **O(n x m)** time in worst case. The worst case occurs when all characters of the text and pattern are same.
* The Bad Character Heuristic may take **O(n/m)** in the best case. The best case occurs when all the characters of the text and pattern are different.