MAHARISHI DAYANAND UNIVERSITY

Delhi Global Institute of Technology

Design and Analysis of Algorithms Lab using C++

Submitted By: Bazgha Razi

Subject Code : LC-CSE-325G

Subject Name: Design and Analysis of Algorithms Lab

Registration Number :*191380214*

Roll Number :

Index

S.No.	Topic	Date	Teacher's Sign
1.	WAP in C++ for iterative and recursive Binary Search.	12/10/2021	
2.	WAP in C++ to sort a given set of elements using the Quick Sort/ Merge Sort/ Selection Sort and determine the time required to sort the elements.	26/10/2021	
3.	WAP in C++ for implementation of fractional knapsack problem using greedy method and 0/1 knapsack problem using dynamic programming.	02/11/2021	
4.	WAP to find the shortest path from a given vertex to other vertices in a weighted connected graph using Dijkstra's algorithm.	16/11/2021	
5.	WAP in C++ to find the minimum spanning tree of a given undirected graph using Kruskal's algorithm/Prim's algorithm.	23/11/2021	
6.	WAP in C++ to implement N-queens problem.	30/11/2021	
7.	WAP in C++ to check whether a given graph is connected or not using DFS method.	6/12/2021	
8.	WAP to implement the Travelling Salesman Problem(TSP).	13/12/2021	
9.	WAP in C++ to check whether undirected graph is connected using DFS.	20/12/2021	
10.	WAP in C++ to implement the subset sum problem.	27/12/2021	

Program 1

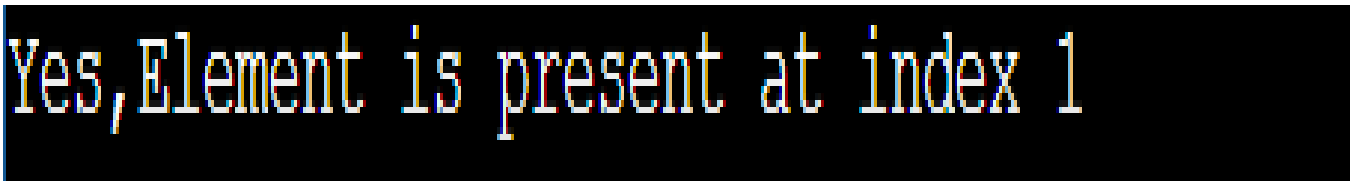
Aim: WAP in C++ for iterative and recursive Binary Search.

CODE
a) Iterative Binary Search

```
#include <bits/stdc++.h>
using namespace std;
struct Node {
  int data;
  struct Node *left, *right;
};
bool iterativeSearch(struct Node* root, int key)
{
  while (root != NULL) {
        if (key > root->data)
              root = root->right;
        else if (key < root->data)
              root = root->left;
        else
              return true;
  }
  return false;
}
struct Node* newNode(int item)
{
  struct Node* temp = new Node;
  temp->data = item;
  temp->left = temp->right = NULL;
  return temp;
}
struct Node* insert(struct Node* Node, int data)
{
  if (Node == NULL)
        return newNode(data);
  if (data < Node->data)
        Node->left = insert(Node->left, data);
  else if (data > Node->data)
        Node->right = insert(Node->right, data);
  return Node;
}
int main()
{
  struct Node* root = NULL;
```

```
  root = insert(root, 50);
  insert(root, 30);
  insert(root, 20);
  insert(root, 40);
  insert(root, 70);
  insert(root, 60);
  insert(root, 80);
  if (iterativeSearch(root, 15))
        cout << "Yes, Element is present at index ";
  else
        cout << "No";
  return 0;
}
```

OUTPUT



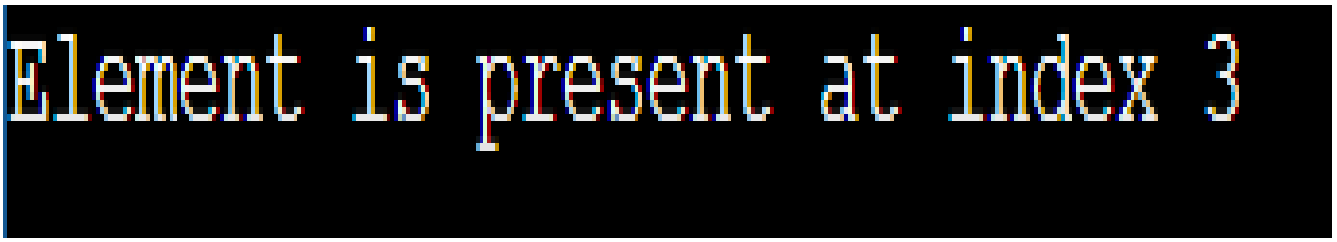
CODE

b) Recursive Binary Search

```
#include <bits/stdc++.h>
using namespace std;
int binarySearch(int arr[], int l, int r, int x)
{
  if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
              return mid;
        if (arr[mid] > x)
              return binarySearch(arr, l, mid - 1, x);
        return binarySearch(arr, mid + 1, r, x);
  }
  return -1;
}
int main(void)
```

```
{ int arr[] = { 2, 3, 4, 10, 40 };
  int x = 10;
  int n = sizeof(arr) / sizeof(arr[0]);
  int result = binarySearch(arr, 0, n - 1, x);
  (result == -1)
            ? cout << "Element is not present in array"
            : cout << "Element is present at index " << result;
  return 0;
}
```

OUTPUT



Program 2

Aim: WAP in C++ to sort a given set of elements using the Quick Sort/ Merge Sort/ Selection Sort and determine the time required to sort the elements.

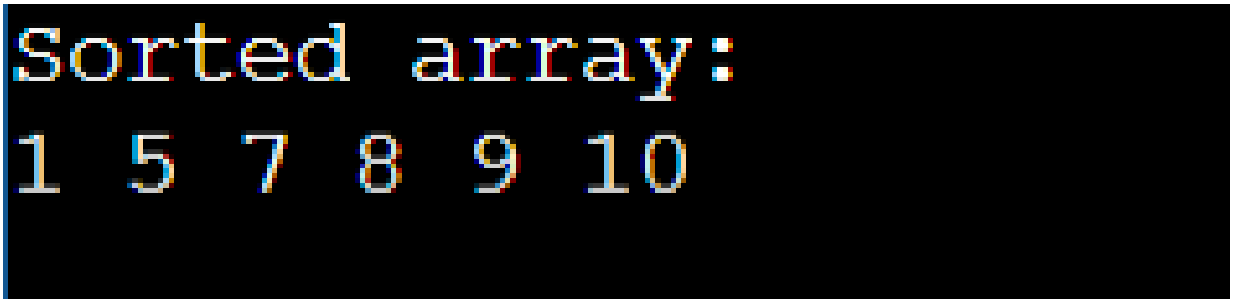
CODE

a) Quick Sort

```
#include <bits/stdc++.h>
using namespace std;

void swap(int* a, int* b)
{
  int t = *a;
  *a = *b;
  *b = t;
}
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++;
              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)
{
  int i;
  for (i = 0; i < size; i++)
        cout << arr[i] << " ";
  cout << endl;
```

```
}
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";
  printArray(arr, n);
  return 0;
}
```

OUTPUT

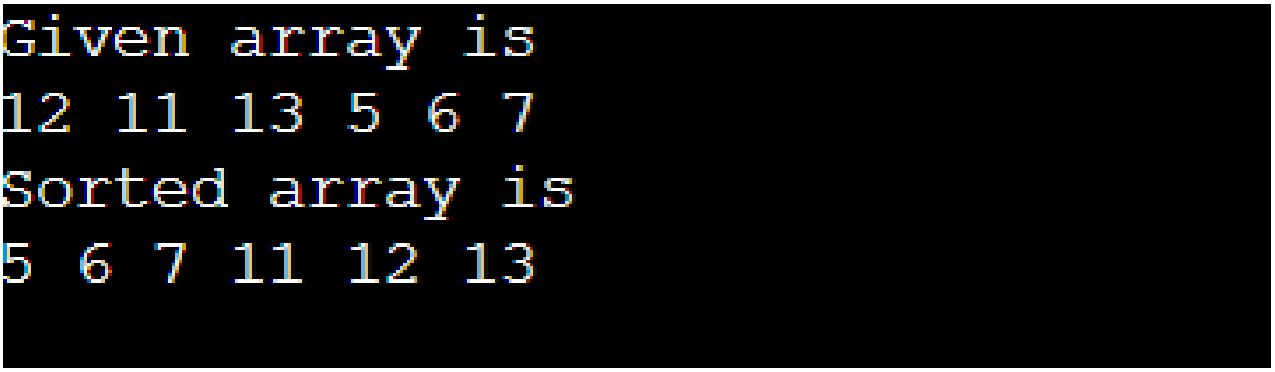


b) Merge Sort

```
#include <iostream>
using namespace std;

void merge(int array[], int const left, int const mid, int const right)
{
  auto const subArrayOne = mid - left + 1;
  auto const subArrayTwo = right - mid;
  auto *leftArray = new int[subArrayOne],
          *rightArray = new int[subArrayTwo];
  for (auto i = 0; i < subArrayOne; i++)
          leftArray[i] = array[left + i];
  for (auto j = 0; j < subArrayTwo; j++)
          rightArray[j] = array[mid + 1 + j];
  auto indexOfSubArrayOne = 0,
          indexOfSubArrayTwo = 0;
  int indexOfMergedArray = left;
  while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {
          if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {
                  array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
                  indexOfSubArrayOne++;
          }
          else {
                  array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
                  indexOfSubArrayTwo++;  }
          indexOfMergedArray++;  }
  while (indexOfSubArrayOne < subArrayOne) {
          array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
```

```
            indexOfSubArrayOne++;
            indexOfMergedArray++;  }
  while (indexOfSubArrayTwo < subArrayTwo) {
            array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
            indexOfMergedArray++;
  } }
void mergeSort(int array[], int const begin, int const end)
{
  if (begin >= end)
            return;
  auto mid = begin + (end - begin) / 2;
  mergeSort(array, begin, mid);
  mergeSort(array, mid + 1, end);
  merge(array, begin, mid, end);
}
void printArray(int A[], int size)
{ for (auto i = 0; i < size; i++)
            cout << A[i] << " "; }
int main()
{
  int arr[] = { 12, 11, 13, 5, 6, 7 };
  auto arr_size = sizeof(arr) / sizeof(arr[0]);
  cout << "Given array is \n";
  printArray(arr, arr_size);
  mergeSort(arr, 0, arr_size - 1);
  cout << "\nSorted array is \n";
  printArray(arr, arr_size);
  return 0;
}
```

OUTPUT

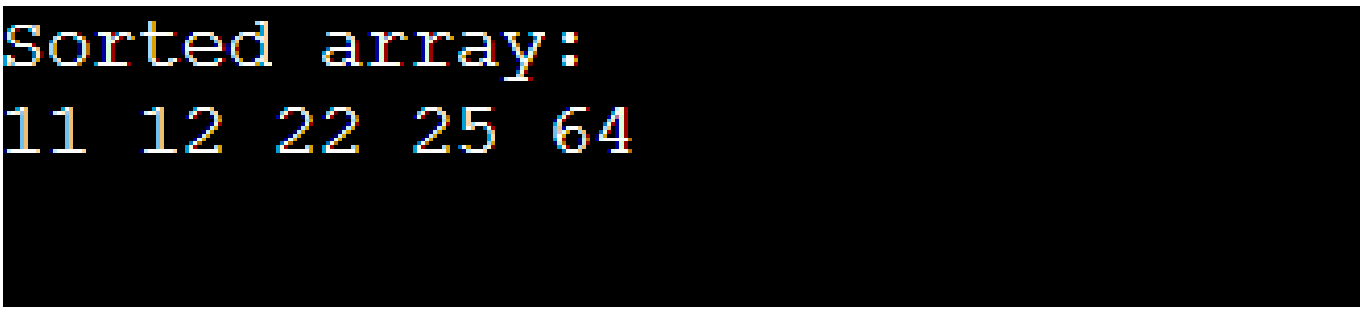


c) Selection Sort

```
#include <bits/stdc++.h>
using namespace std;
void swap(int *xp, int *yp)
{
  int temp = *xp;
```

```
  *xp = *yp;
  *yp = temp;
}
void selectionSort(int arr[], int n)
{
  int i, j, min_idx;
  for (i = 0; i < n-1; i++)
  {
            min_idx = i;
            for (j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                      min_idx = j;
            swap(&arr[min_idx], &arr[i]);
  }
}
void printArray(int arr[], int size)
{
  int i;
  for (i=0; i < size; i++)
            cout << arr[i] << " ";
  cout << endl;
}
int main()
{ int arr[] = {64, 25, 12, 22, 11};
  int n = sizeof(arr)/sizeof(arr[0]);
  selectionSort(arr, n);
  cout << "Sorted array: \n";
  printArray(arr, n);
  return 0;
}
```

OUTPUT



Program 3

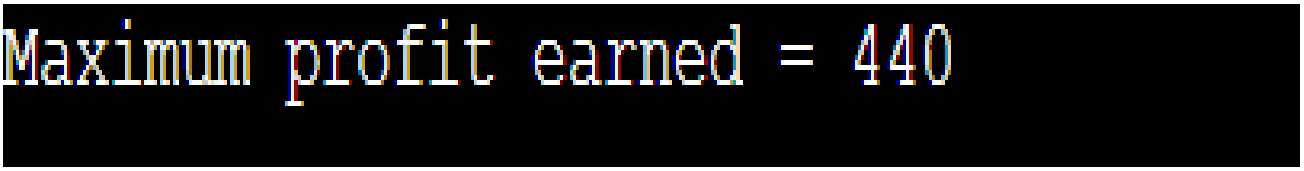
Aim: WAP in C++ for implementation of fractional knapsack problem using greedy method and 0/1 knapsack problem using dynamic programming.

CODE

a) Fractional Knapsack

```
#include <bits/stdc++.h>
using namespace std;
struct Item {
  int value, weight;
  Item(int value, int weight)
        : value(value), weight(weight)
  {
  } };
bool cmp(struct Item a, struct Item b)
{
  double r1 = (double)a.value / a.weight;
  double r2 = (double)b.value / b.weight;
  return r1 > r2;
}
double fractionalKnapsack(struct Item arr[], int N, int size)
{
  sort(arr, arr + size, cmp);
  int curWeight = 0;
  double finalvalue = 0.0;
  for (int i = 0; i < size; i++) {
        if (curWeight + arr[i].weight <= N) {
              curWeight += arr[i].weight;
              finalvalue += arr[i].value; }
        else {   int remain = N - curWeight;
              finalvalue += arr[i].value* ((double)remain/ arr[i].weight);
              break; } }
  return finalvalue;
}
int main()
{ int N = 60;
  Item arr[] = { { 100, 10 }, { 280, 40 }, { 120, 20 }, { 120, 24 } };
  int size = sizeof(arr) / sizeof(arr[0]);
  cout << "Maximum profit earned = "
        << fractionalKnapsack(arr, N, size);
  return 0;
}
```

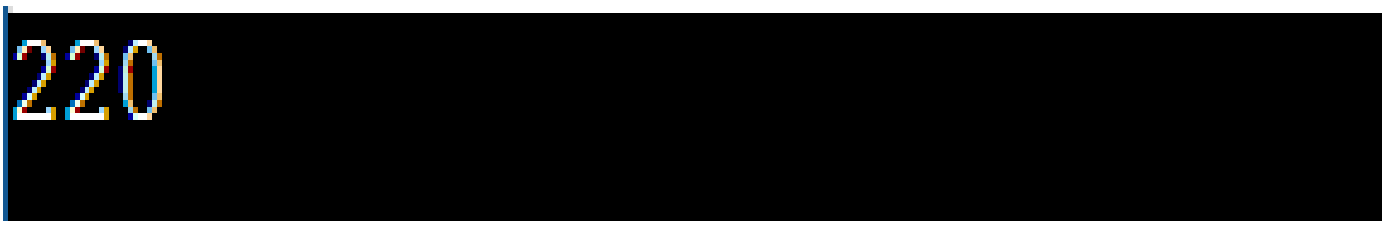
OUTPUT



b) 0/1 Knapsack

```
#include <bits/stdc++.h>
using namespace std;
int max(int a, int b) { return (a > b) ? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
  if (n == 0 || W == 0)
        return 0;
  if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);
  else
        return max(val[n - 1]+ knapSack(W - wt[n - 1],wt, val, n - 1),knapSack(W, wt, val, n - 1));
}
int main()
{
  int val[] = { 60, 100, 120 };
  int wt[] = { 10, 20, 30 };
  int W = 50;
  int n = sizeof(val) / sizeof(val[0]);
  cout << knapSack(W, wt, val, n);
  return 0;
}
```

OUTPUT



Program 4

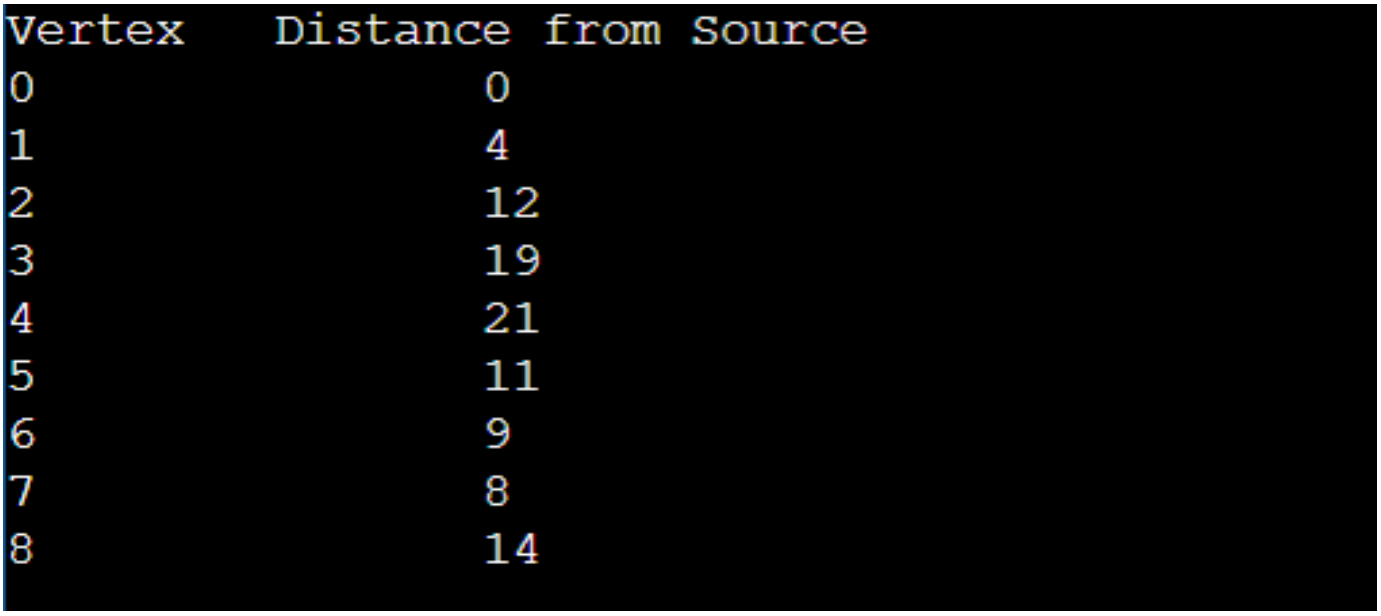
Aim: WAP to find the shortest path from a given vertex to other vertices in a weighted connected graph using Dijkstra's algorithm.

CODE

```
#include <iostream>
using namespace std;
#include <limits.h>
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;
	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"<<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);
}

int main()
{
```

```
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 }, { 4, 0, 8, 0, 0, 0, 0, 11, 0 }, { 0, 8, 0, 7, 0, 4, 0, 0, 2 }, { 0, 0, 7, 0, 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;
}
```

OUTPUT



Program 5

Aim: WAP in C++ to find the minimum spanning tree of a given undirected graph using Kruskal's algorithm/Prim's algorithm.

CODE

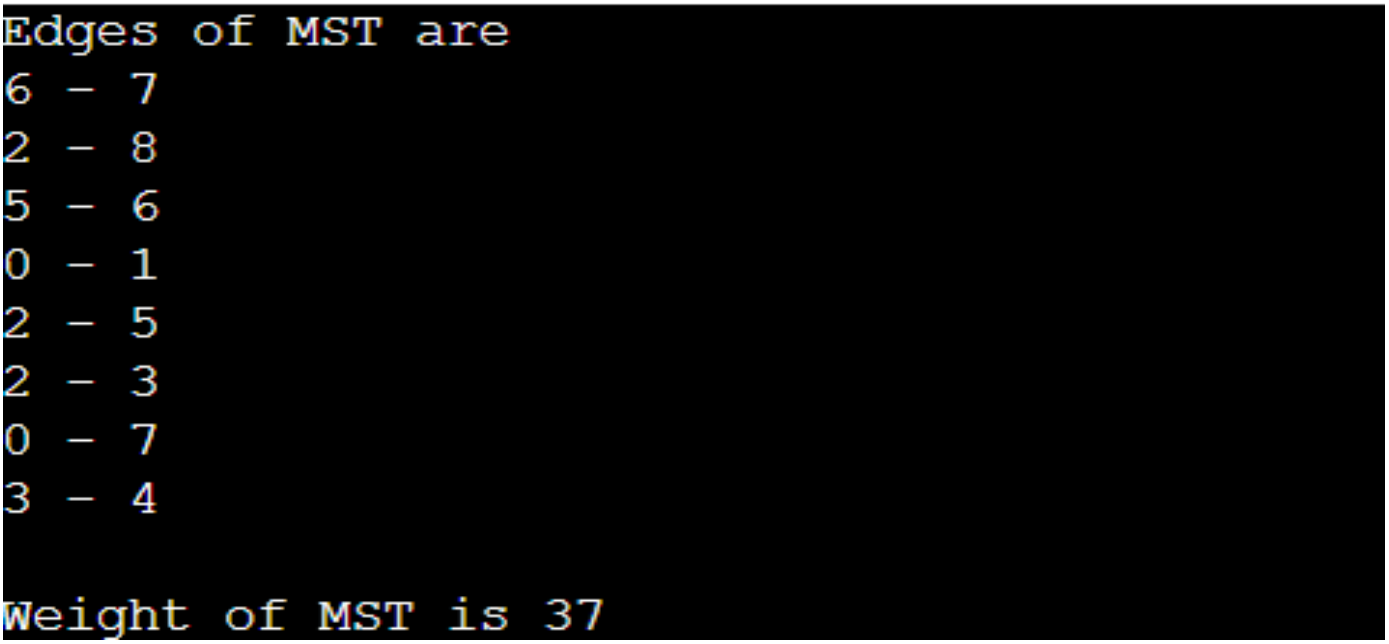
a) Kruskal's Method

```
#include<bits/stdc++.h>
using namespace std;
typedef pair<int, int> iPair;
struct Graph
{
  int V, E;
  vector< pair<int, iPair> > edges;
  Graph(int V, int E)
  {
        this->V = V;
        this->E = E;
  }
  void addEdge(int u, int v, int w)
  {
        edges.push_back({w, {u, v}});
  }
  int kruskalMST();
};
struct DisjointSets
{
  int *parent, *rnk;
  int n;
  DisjointSets(int n)
  {     this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];
        for (int i = 0; i <= n; i++)
        {
              rnk[i] = 0;
              parent[i] = i;
        }
  }
  int find(int u)
  {     if (u != parent[u])
        parent[u] = find(parent[u]);
        return parent[u];
  }

  void merge(int x, int y)
  {
```

```
        x = find(x), y = find(y);
        if (rnk[x] > rnk[y])
                parent[y] = x;
        else
                parent[x] = y;

        if (rnk[x] == rnk[y])
                rnk[y]++;  } };
int Graph::kruskalMST()
{
  int mst_wt = 0;
  sort(edges.begin(), edges.end());
  DisjointSets ds(V);
  vector< pair<int, iPair> >::iterator it;
  for (it=edges.begin(); it!=edges.end(); it++)
  {       int u = it->second.first;
          int v = it->second.second;
          int set_u = ds.find(u);
          int set_v = ds.find(v);
          if (set_u != set_v)
          {       cout << u << " - " << v << endl;
                  mst_wt += it->first;
                  ds.merge(set_u, set_v);
          }
  }
  return mst_wt;
}
int main()
{
  int V = 9, E = 14;
  Graph g(V, E);
  g.addEdge(0, 1, 4);
  g.addEdge(0, 7, 8);
  g.addEdge(1, 2, 8);
  g.addEdge(1, 7, 11);
  g.addEdge(2, 3, 7);
  g.addEdge(2, 8, 2);
  g.addEdge(2, 5, 4);
  g.addEdge(3, 4, 9);
  g.addEdge(3, 5, 14);
  g.addEdge(4, 5, 10);
  g.addEdge(5, 6, 2);
  g.addEdge(6, 7, 1);
  g.addEdge(6, 8, 6);
  g.addEdge(7, 8, 7);

  cout << "Edges of MST are \n";
  int mst_wt = g.kruskalMST();
```

```
  cout << "\nWeight of MST is " << mst_wt;

  return 0;
}
```

OUTPUT

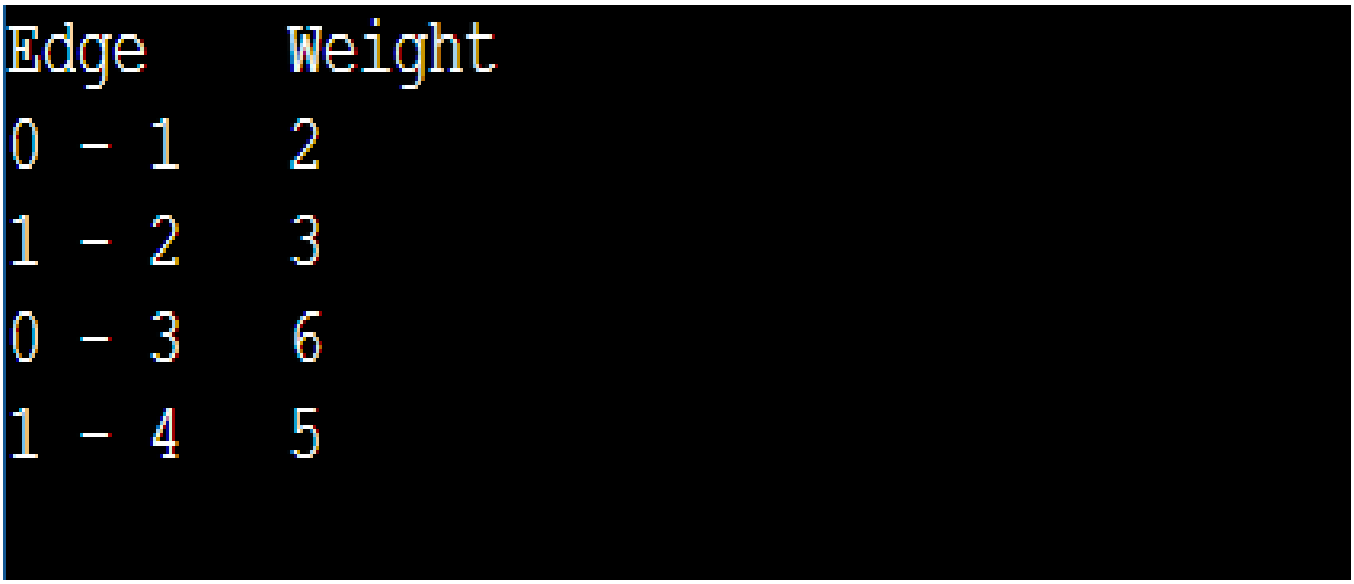


b) Prim's Method

```
#include <bits/stdc++.h>
using namespace std;
#define V 5
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])
{
  cout<<"Edge \tWeight\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



Program 6

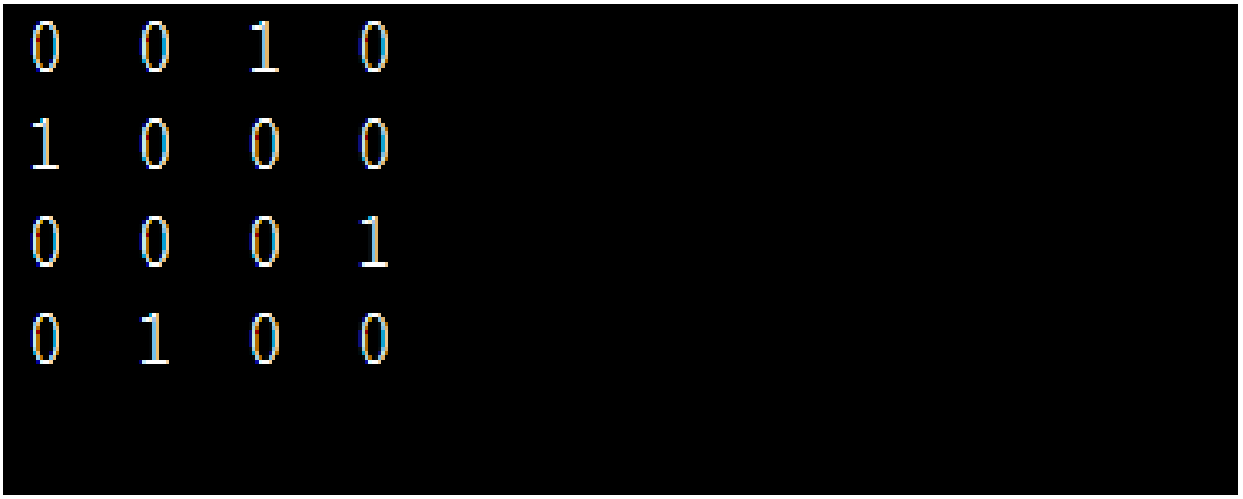
Aim: WAP in C++ to implement N-queens problem using backtracking.

CODE

```
#include <stdbool.h>
#include <stdio.h>
void printSolution(int board[N][N])
{	for (int i = 0; i < N; i++) {
		for (int j = 0; j < N; j++)
			printf(" %d ", board[i][j]);
		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) {
		printf("Solution does not exist");
		return false;
```

```
        }

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

OUTPUT



Program 7

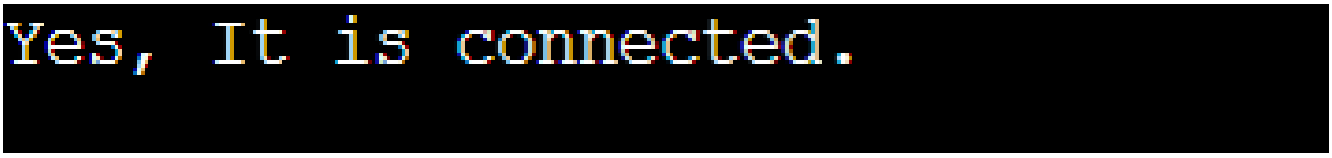
Aim: WAP in C++ to check whether a given graph is connected or not using DFS method.

CODE

```
#include <bits/stdc++.h>
using namespace std;
#define N 100000
vector<int> gr1[N], gr2[N];
bool vis1[N], vis2[N];
void Add_edge(int u, int v)
{
	gr1[u].push_back(v);
	gr2[v].push_back(u);
}
void dfs1(int x)
{
	vis1[x] = true;
	for (auto i : gr1[x])
		if (!vis1[i])
			dfs1(i);
}

// DFS function
void dfs2(int x)
{
	vis2[x] = true;
	for (auto i : gr2[x])
		if (!vis2[i])
			dfs2(i);
}
bool Is_Connected(int n)
{
	memset(vis1, false, sizeof vis1);
	dfs1(1);
	memset(vis2, false, sizeof vis2);
	dfs2(1);
	for (int i = 1; i <= n; i++) {
		if (!vis1[i] and !vis2[i])
			return false; }
	return true;
}
```

```
int main()
{
	int n = 4;
	Add_edge(1, 2);
	Add_edge(1, 3);
	Add_edge(2, 3);
	Add_edge(3, 4);
	if (Is_Connected(n))
		cout << "Yes, It is connected.";
	else
		cout << "No";

	return 0;
}
```

OUTPUT



Program 8

Aim: WAP to implement the Travelling Salesman Problem(TSP).

CODE

```
#include <bits/stdc++.h>
using namespace std;
int travllingSalesmanProblem(int graph[][V], int s)
{
	vector<int> vertex;
	for (int i = 0; i < V; i++)
		if (i != s)
			vertex.push_back(i);
	int min_path = INT_MAX;
	do {
		int current_pathweight = 0;
		int k = s;
		for (int i = 0; i < vertex.size(); i++) {
			current_pathweight += graph[k][vertex[i]];
			k = vertex[i];
		}
		current_pathweight += graph[k][s];
		min_path = min(min_path, current_pathweight);}
 while (
		next_permutation(vertex.begin(), vertex.end()));
	return min_path;
}
int main()
{
	int graph[][V] = { { 0, 10, 15, 20 },{ 10, 0, 35, 25 },{ 15, 35, 0, 30 },{ 20, 25, 30, 0 } };
	int s = 0;
	cout << travllingSalesmanProblem(graph, s) << endl;
	return 0;
}
```

OUTPUT

Program 9

Aim: WAP in C++ to check whether undirected graph is connected using DFS.

CODE

```
#include <iostream>
#include <list>
#include <stack>
using namespace std;
class Graph
{
   private:
      int V;
      list<int> *adj;
      void DFSUtil(int v, bool visited[]);
   public:
      Graph(int V)
      {
         this->V = V;
         adj = new list<int>[V];
      }
      ~Graph()
      {
         delete [] adj;
      }
      void addEdge(int v, int w);
      bool isConnected();
      Graph getTranspose();
};
void Graph::DFSUtil(int v, bool visited[])
{
   visited[v] = true;
   list<int>::iterator i;
   for (i = adj[v].begin(); i != adj[v].end(); ++i)
      if (!visited[*i])
         DFSUtil(*i, visited);
}
Graph Graph::getTranspose()
{
   Graph g(V);
   for (int v = 0; v < V; v++)
   {
      list<int>::iterator i;
      for(i = adj[v].begin(); i != adj[v].end(); ++i)
      {
         g.adj[*i].push_back(v);
      }
   }
   return g;
}
void Graph::addEdge(int v, int w)
{
   adj[v].push_back(w);
   adj[w].push_back(v);
}
```

```
bool Graph::isConnected()
{
   bool visited[V];
   for (int i = 0; i < V; i++)
      visited[i] = false;
   DFSUtil(0, visited);
   for (int i = 0; i < V; i++)
      if (visited[i] == false)
          return false;
   Graph gr = getTranspose();
   for(int i = 0; i < V; i++)
      visited[i] = false;
   gr.DFSUtil(0, visited);
   for (int i = 0; i < V; i++)
      if (visited[i] == false)
          return false;
   return true;
}
int main()
{
   Graph g1(5);
   g1.addEdge(0, 1);
   g1.addEdge(1, 2);
   g1.addEdge(2, 3);
   g1.addEdge(3, 0);
   g1.addEdge(2, 4);
   g1.addEdge(4, 2);
   if (g1.isConnected())
      cout<<"The Graph is Conneted"<<endl;
   else
      cout<<"The Graph is not Connected"<<endl;

   Graph g2(4);
   g2.addEdge(0, 1);
   g2.addEdge(1, 2);
   g2.addEdge(2, 3);
   if (g2.isConnected())
      cout<<"The Graph is Connected"<<endl;
   else
      cout<<"The Graph is not Connected"<<endl;

   return 0;
}
```

OUTPUT

```
The Graph is Conneted
The Graph is Connected
```

Program 10

Aim: WAP in C++ to implement the subset sum problem.

CODE

```
#include<iostream>
using namespace std;
int main()
{
  int n,sum;
  cin>>n>>sum;
  int a[n];
  for(int i=0;i<n;i++)
     cin>>a[i];
  int table[n+1][sum+1];
  for(int i=0;i<sum+1;i++)
     table[0][i]=false;
  for(int i=0;i<n+1;i++)
     table[i][0]=true;
  for(int i=1;i<n+1;i++)
   {
     for(int j=1;j<sum+1;j++)
     {
        if(j<a[i-1])
           table[i][j]=table[i-1][j];
        else
           table[i][j]=table[i-1][j]||table[i-1][j-a[i-1]];
     }
   }
   if(table[n][sum])
     cout<<"Yes";
  else
     cout<<"No";
}
```

OUTPUT

```
5 15
2 3 4 5 1
Yes
```