**EXPERIMENT NO. 1**

Write a program to find out a roll number from college database using binary search algorithm.

**PROGRAM**

1. **Iterative (Non-recursive approach)**

#include <iostream>

#include <stdio.h>

using namespace std;

int main()

{

//clrscr();

int n, i, search, first, last, middle;

int arr[] = {1715372, 1715373, 1715374, 1715375, 1715376};

int x;

cout<<”The list of roll numbers saved is as follows:”<<endl;

for (int i=0; i<5; i++) {

cout<<arr[i]<<"\t";

}

cout<<"Enter a number to find :";

cin>>search;

first = 0;

last = n-1;

middle = (first+last)/2;

while (first <= last)

{

if(arr[middle] < search)

{

first = middle + 1;

}

else if(arr[middle] == search)

{

cout<<search<<" found at location "<<middle+1<<"\n";

break;

}

else

{

last = middle - 1;

}

middle = (first + last)/2;

}

if(first > last)

{

cout<<"Not found! "<<search<<" is not present in the list.";

}

return 0;

getch();

}

1. **Recursive approach**

// C++ program to implement recursive Binary Search

#include <iostream>

using namespace std;

// A recursive binary search function. It returns

// location of x in given array arr[l..r] is present,

// otherwise -1

int binarySearch(int arr[], int l, int r, int x)

{

if (r >= l) {

int mid = l + (r - l) / 2;

// If the element is present at the middle

// itself

if (arr[mid] == x)

return mid;

// If element is smaller than mid, then

// it can only be present in left subarray

if (arr[mid] > x)

return binarySearch(arr, l, mid - 1, x);

// Else the element can only be present

// in right subarray

return binarySearch(arr, mid + 1, r, x); }

// We reach here when element is not

// present in array

return -1;

}

int main(void){

int arr[] = {1715372, 1715373, 1715374, 1715375, 1715376};

int x;

cout<<”The list of roll numbers saved is as follows:”<<endl;

for (int i=0; i<5; i++) {

cout<<arr[i]<<"\t";

}

cout<<"Enter the element to search: "<<endl;

cin>>x;

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

int result = binarySearch(arr, 0, n - 1, x);

(result == -1) ? cout << "Element is not present in array"

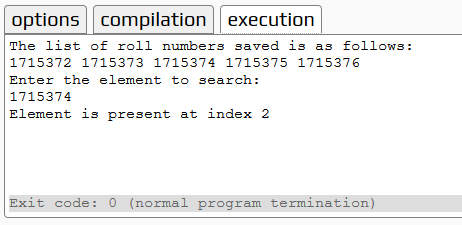
: cout << "Element is present at index " << result;

return 0;

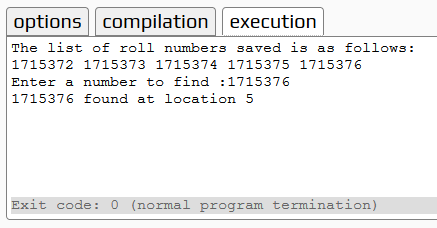
}

**Output**

1. **Iterative (Non-recursive approach)**



1. **Recursive approach**



**EXPERIMENT NO. 2**

Write a program to sort the roll numbers of your class using merge sort algorithm and determine the time required to sort the elements.

**PROGRAM**

#include<iostream>

using namespace std;

void swapping(int &a, int &b) { //swap the content of a and b

int temp;

temp = a;

a = b;

b = temp;

}

void display(int \*array, int size) {

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

cout << array[i] << " ";

cout << endl;

}

void merge(int \*array, int l, int m, int r) {

int i, j, k, nl, nr;

//size of left and right sub-arrays

nl = m-l+1; nr = r-m;

int larr[nl], rarr[nr];

//fill left and right sub-arrays

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

larr[i] = array[l+i];

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

rarr[j] = array[m+1+j];

i = 0; j = 0; k = l;

//marge temp arrays to real array

while(i < nl && j<nr) {

if(larr[i] <= rarr[j]) {

array[k] = larr[i];

i++;

}else{

array[k] = rarr[j];

j++;

}

k++;

}

while(i<nl) { //extra element in left array

array[k] = larr[i];

i++; k++;

}

while(j<nr) { //extra element in right array

array[k] = rarr[j];

j++; k++;

}

}

void mergeSort(int \*array, int l, int r) {

int m;

if(l < r) {

int m = l+(r-l)/2;

// Sort first and second arrays

mergeSort(array, l, m);

mergeSort(array, m+1, r);

merge(array, l, m, r);

}

}

int main() {

int n;

cout << "Enter the number of elements: ";

cin >> n;

int arr[n]; //create an array with given number of elements

cout << "Enter elements:" << endl;

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

cin >> arr[i];

}

cout << "Array before Sorting: ";

display(arr, n);

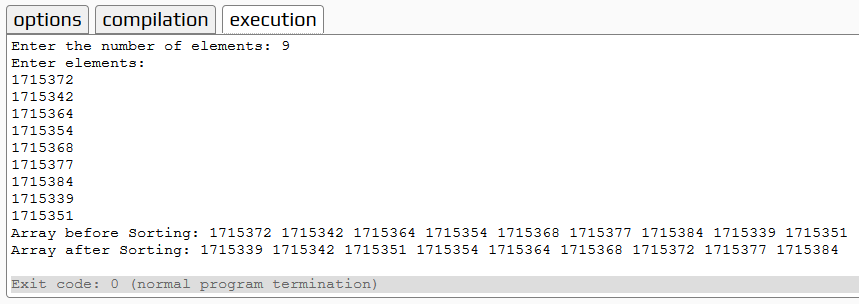
mergeSort(arr, 0, n-1); //(n-1) for last index

cout << "Array after Sorting: ";

display(arr, n);

}

**Output**



**EXPERIMENT NO. 3**

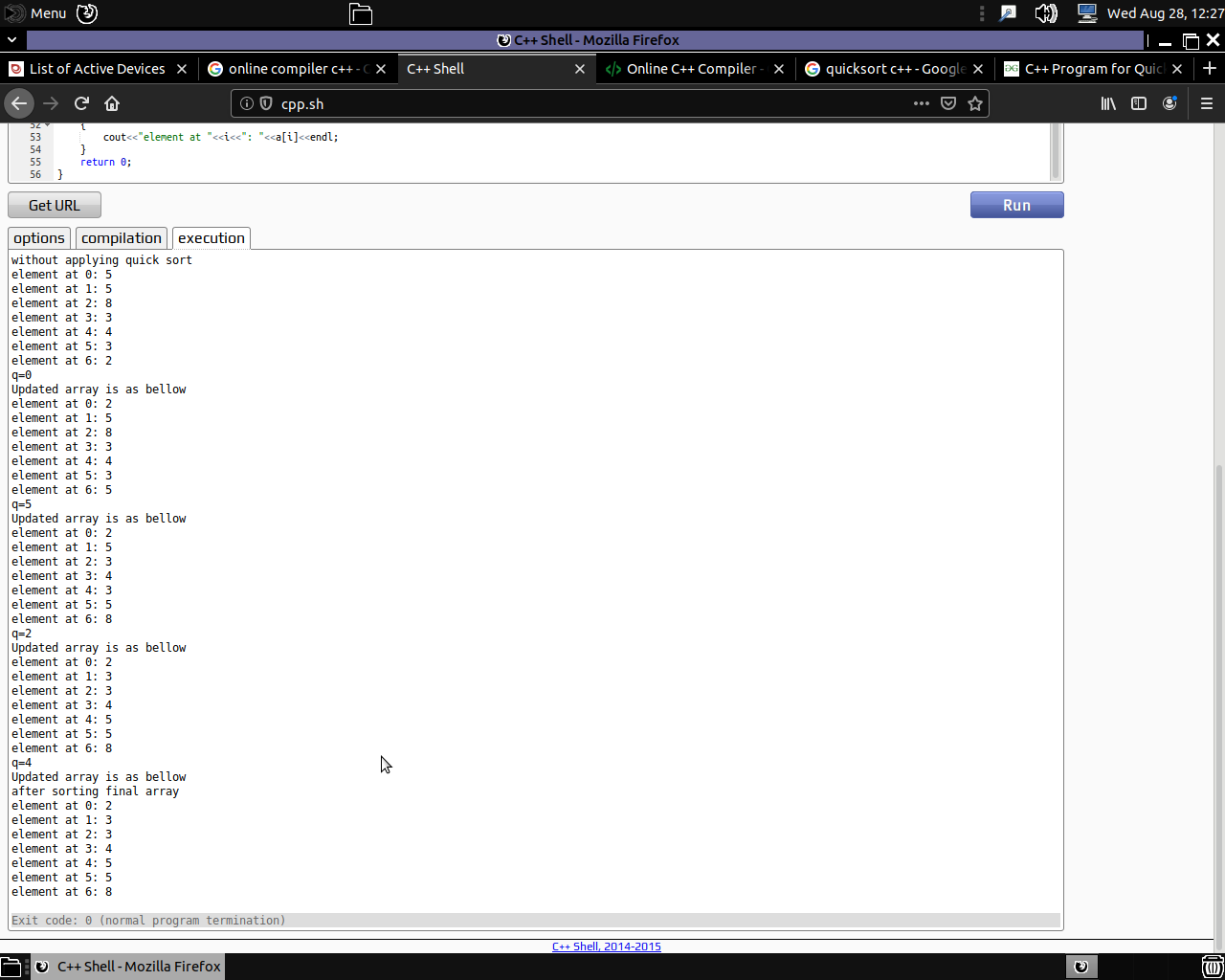
Write a program to sort the university roll numbers of your class using Quick sort method and determine the time required to sort the element.

**PROGRAM**

#include <iostream>  
using namespace std;  
int partition(int a[],int p,int r) {  
    int x = a[r];  
    int temp;  
    int i = p-1;  
    for (int j = p; j <= r-1; j++) {  
        if (a[j] <= x) {  
            i = i+1;

// swapping the element less than the pivot element in the list  
            temp = a[i];   
            a[i] = a[j];  
            a[j] = temp;  
        }  
    }  
    temp = a[i+1]; //swapping pivot element  
    a[i+1] = a[r];  
    a[r] = temp;  
    return (i+1);  
}  
void quicksort(int a[],int p,int r) {  
    if (p<r) {  
        for (int i=0; i<=6; i++) {  
            cout<<"element at "<<i<<": "<<a[i]<<endl;  
        }  
        int q = partition(a,p,r);  
        cout<<"q="<<q<<endl;  
        cout<<"Updated array is as bellow"<<endl;  
        quicksort(a,p,q-1);  
        quicksort(a,q+1,r);    
    }  
}  
int main()  
{  
    int a[7] = {5,5,8,3,4,3,2};  
    int p,r;  
    p=0;  
    r=6;  
    cout<<"without applying quick sort"<<endl;  
    quicksort(a,p,r);  
    cout<<"after sorting final array"<<endl;  
    for (int i=0; i<7; i++)  
    {  
        cout<<"element at "<<i<<": "<<a[i]<<endl;  
    }  
    return 0;  
}

**Output**



**EXPERIMENT NO. 4**

Write a program to solve 0/1 knapsack using Greedy algorithm

**PROGRAM**

1. **Iterative (Non-recursive approach)**

#include<iostream>

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)

{

cout<<"inside function"<<endl;

int i, w;

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

cout<<"Variables intialized"<<endl;

// 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];

}

}

cout<<"for loop checking whether the elements to be considered or not"<<K[n][W]<<endl;

return K[n][W];

}

int main()

{

cout<<"Iterative Method"<<endl;

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

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

cout<<"Initial variables are as folows: \n"<<endl;

cout<<"Size of array is: "<<n<<endl;

cout<<"Weight of Knapsack is: "<<W<<endl;

cout<<"Values are: "<<endl;

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

{

cout<<val[i]<<endl;

}

cout<<"Weights are: "<<endl;

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

{

cout<<wt[i]<<endl;

}

cout<<"calling function with values \nW- weight of knapsack,\nwt- array of weight of items,\nval- value of various items in an array form,\nn- total number of elements in the array"<<endl;

cout<< "Value returned from the function is: "<<knapSack(W, wt, val, n);

return 0;

}

1. **Recursive approach**

/\* A Naive recursive implementation of 0-1 Knapsack problem \*/

#include <iostream>

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)

{

// Base Case

if (n == 0 || W == 0)

return 0;

// If weight of the nth item is more

// than Knapsack capacity W, then

// this item cannot be included

// in the optimal solution

if (wt[n-1] > W)

{

cout<<"if weight is greather then the next value is considered"<<endl;

return knapSack(W, wt, val, n-1);

}

// Return the maximum of two cases:

// (1) nth item included

// (2) not included

else

{

cout<<"Max function called to check the max value between 2 knapsack return values"<<endl;

return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),

knapSack(W, wt, val, n-1) );

}

}

// Driver code

int main()

{

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

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

cout<<"Initial variables are as folows: \n"<<endl;

cout<<"Size of array is: "<<n<<endl;

cout<<"Weight of Knapsack is: "<<W<<endl;

cout<<"Values are: "<<endl;

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

{

cout<<val[i]<<endl;

}

cout<<"Weights are: "<<endl;

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

{

cout<<wt[i]<<endl;

}

cout<<"calling function with values \nW- weight of knapsack,\nwt- array of weight of items,\nval- value of various items in an array form,\nn- total number of elements in the array"<<endl;

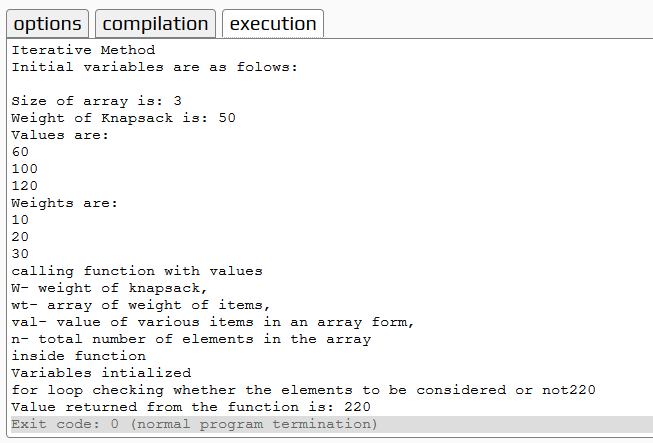
cout<< "Value returned from the function is: "<<knapSack(W, wt, val, n);

return 0;

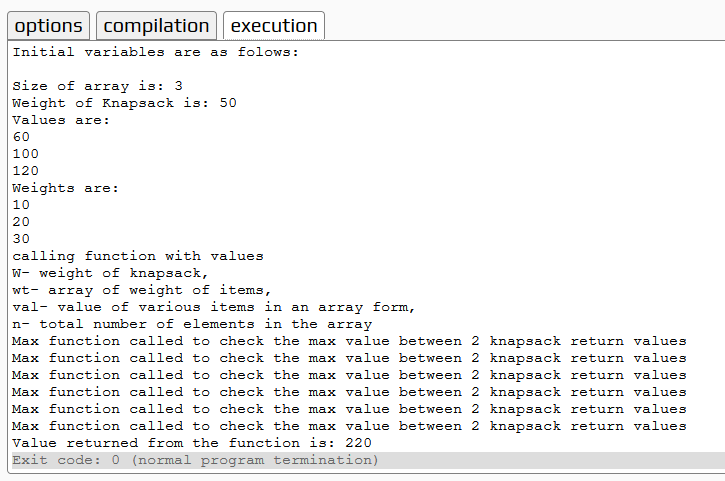
}

**Output**

1. **Iterative (Non-recursive approach)**

****

1. **Recursive approach**

****

**EXPERIMENT NO. 5**

Write a program to find minimum cost to set the phone lines to connect all the cities of your state using Prim’s algorithm.

**PROGRAM**

#include <iostream>

#include <vector>

#include <queue>

#include <functional>

#include <utility>

using namespace std;

const int MAX = 1e4 + 5;

typedef pair<long long, int> PII;

bool marked[MAX];

vector <PII> adj[MAX];

long long prim(int x)

{

priority\_queue<PII, vector<PII>, greater<PII> > Q;

int y;

long long minimumCost = 0;

PII p;

Q.push(make\_pair(0, x));

while(!Q.empty())

{

// Select the edge with minimum weight

p = Q.top();

Q.pop();

x = p.second;

// Checking for cycle

if(marked[x] == true)

continue;

minimumCost += p.first;

marked[x] = true;

for(int i = 0;i < adj[x].size();++i)

{

y = adj[x][i].second;

if(marked[y] == false)

Q.push(adj[x][i]);

}

}

return minimumCost;

}

int main()

{

int nodes, edges, x, y;

long long weight, minimumCost;

cout<<"Enter the number of nodes and edges: "<<endl;

cin >> nodes >> edges;

for(int i = 0;i < edges;++i)

{

cout<<"Starting vertex";

cin >> x;

cout<<"Ending vertex";

cin>> y ;

cout<<"Weight";

cin>> weight;

adj[x].push\_back(make\_pair(weight, y));

adj[y].push\_back(make\_pair(weight, x));

}

// Selecting 1 as the starting node

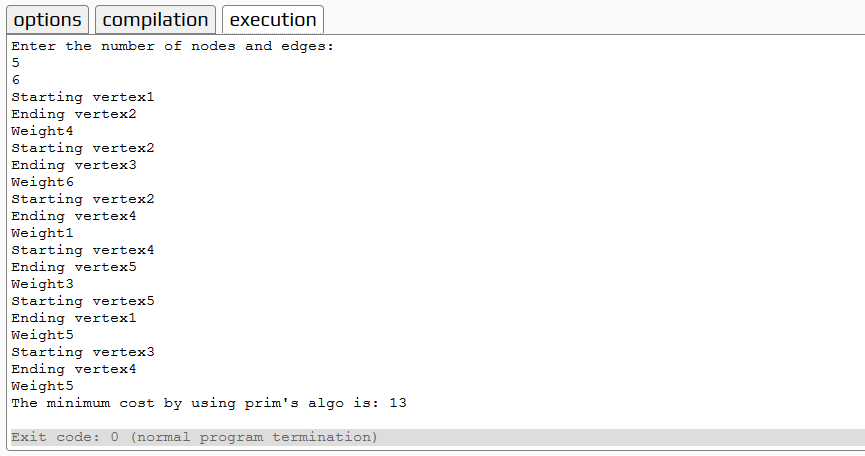
minimumCost = prim(1);

cout <<"The minimum cost by using prim's algo is: "<< minimumCost << endl;

return 0;

}

**Output**



**EXPERIMENT NO. 6**

Write a program to find minimum cost of connecting all the engineering colleges in your state using Kruskal’s algorithm.

**PROGRAM**

#include <iostream>

#include <vector>

#include <utility>

#include <algorithm>

using namespace std;

const int MAX = 1e4 + 5;

int id[MAX], nodes, edges;

pair <long long, pair<int, int> > p[MAX];

void initialize()

{

for(int i = 0;i < MAX;++i)

id[i] = i;

}

int root(int x)

{

while(id[x] != x)

{

id[x] = id[id[x]];

x = id[x];

}

return x;

}

void union1(int x, int y)

{

int p = root(x);

int q = root(y);

id[p] = id[q];

}

long long kruskal(pair<long long, pair<int, int> > p[])

{

int x, y;

long long cost, minimumCost = 0;

for(int i = 0;i < edges;++i)

{

// Selecting edges one by one in increasing order from the beginning

x = p[i].second.first;

y = p[i].second.second;

cost = p[i].first;

// Check if the selected edge is creating a cycle or not

if(root(x) != root(y))

{

minimumCost += cost;

union1(x, y);

}

}

return minimumCost;

}

int main()

{

int x, y;

long long weight, cost, minimumCost;

initialize();

cout<<"Enter the number of nodes: ";

cin >> nodes;

cout<<"Enter the number of edges: ";

cin>>edges;

cout<<"Enter the values in the form as given (starting vertex, ending vertex and weight)";

for(int i = 0;i < edges;++i)

{

cout<<"St veterx: ";

cin >> x;

cout<<"Ed vertex: ";

cin>> y;

cout<<"Weight: ";

cin>> weight;

p[i] = make\_pair(weight, make\_pair(x, y));

}

// Sort the edges in the ascending order

sort(p, p + edges);

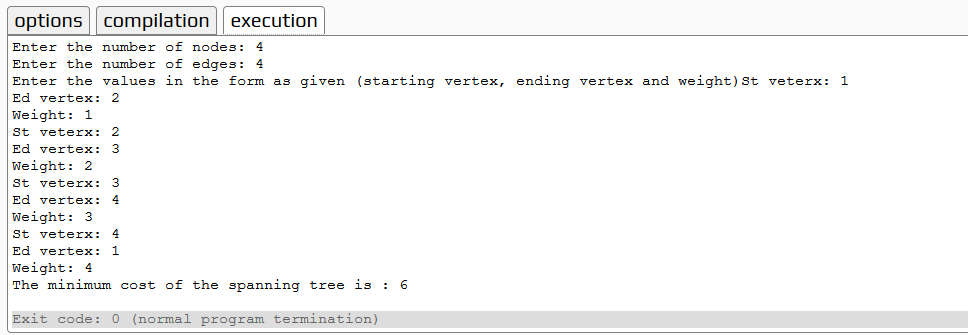
minimumCost = kruskal(p);

cout <<"The minimum cost of the spanning tree is : "<< minimumCost << endl;

return 0;

}

**Output**



**EXPERIMENT NO. 7**

Write a program to find minimum route for a newspaper distributer of your locality using greedy algorithm.

**PROGRAM**

// Example program

#include <iostream>

using namespace std;

int matrix[25][25], visited\_cities[10], limit, cost = 0;

int tsp(int c)

{

int count, nearest\_city = 999;

int minimum = 999, temp;

for(count = 0; count < limit; count++)

{

if((matrix[c][count] != 0) && (visited\_cities[count] == 0))

{

if(matrix[c][count] < minimum)

{

minimum = matrix[count][0] + matrix[c][count];

}

temp = matrix[c][count];

nearest\_city = count;

}}

if(minimum != 999)

{ cost = cost + temp; }

return nearest\_city;

}

void minimum\_cost(int city)

{

int nearest\_city;

visited\_cities[city] = 1;

cout<<city + 1;

nearest\_city = tsp(city);

if(nearest\_city == 999)

{

nearest\_city = 0;

cout<< nearest\_city + 1;

cost = cost + matrix[city][nearest\_city];

return;

}

minimum\_cost(nearest\_city);

}

int main()

{

int i, j;

cout<<"Enter Total Number of Cities:\t";

cin>>limit;

cout<<"\nEnter Cost Matrix\n";

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

{ int t;

t=i+1;

cout<<"\nEnter "<<limit << " Elements in Row[" << t << "]\n";

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

{

cin>>matrix[i][j];

}

visited\_cities[i] = 0; }

cout<<"\nEntered Cost Matrix\n";

for(i = 0; i < limit; i++){

cout<<"\n";

for(j = 0; j < limit; j++) {

cout<<matrix[i][j];

}}

cout<<"\n\nPath:\t";

minimum\_cost(0);

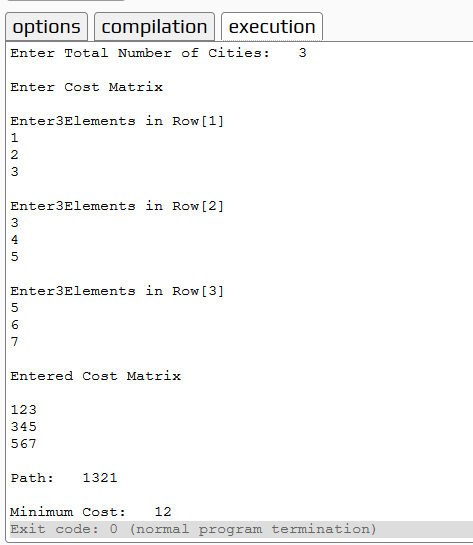
cout<<"\n\nMinimum Cost: \t";

cout<<cost;

return 0;

}

**Output**



**EXPERIMENT NO. 8**

Write a program to find shortest path from home to college using Dijkstra’s Algorithm.

**PROGRAM**

#include<iostream>

#include<stdio.h>

using namespace std;

int shortest(int ,int);

int cost[10][10],dist[20],i,j,n,k,m,S[20],v,totcost,path[20],p;

main()

{

int c;

cout <<"enter no of vertices";

cin >> n;

cout <<"enter no of edges";

cin >>m;

cout <<"\nenter\nEDGE Cost\n";

for(k=1;k<=m;k++)

{

cin >> i >> j >>c;

cost[i][j]=c;

}

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

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

if(cost[i][j]==0)

cost[i][j]=31999;

cout <<"enter initial vertex: ";

cin >>v;

cout << v<<"\n";

shortest(v,n);

}

int shortest(int v,int n)

{

int min;

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

{

S[i]=0;

dist[i]=cost[v][i];

}

path[++p]=v;

S[v]=1;

dist[v]=0;

for(i=2;i<=n-1;i++)

{

k=-1;

min=31999;

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

{

if(dist[j]<min && S[j]!=1)

{

min=dist[j];

k=j;

} }

if(cost[v][k]<=dist[k])

p=1;

path[++p]=k;

for(j=1;j<=p;j++)

cout<<path[j];

cout <<"\n";

//cout <<k;

S[k]=1;

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

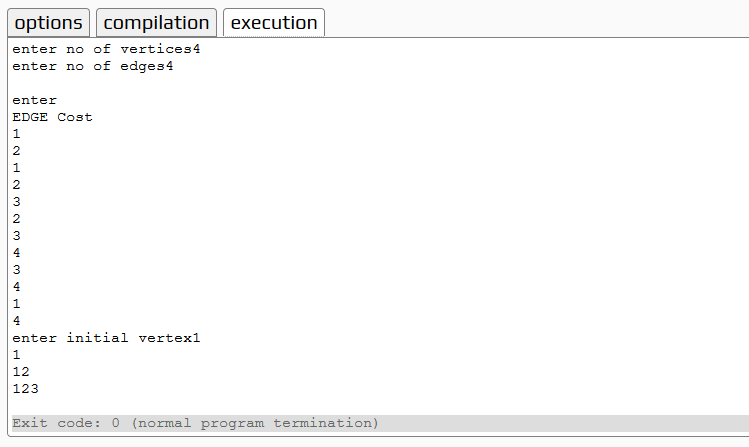
if(cost[k][j]!=31999 && dist[j]>=dist[k]+cost[k][j] && S[j]!=1)

dist[j]=dist[k]+cost[k][j];

}

}

**Output**



**EXPERIMENT NO. 9**

Write a program to find shortest path from home to college using Bellman-Ford Algorithm.

**PROGRAM**

// A C++ program for Bellman-Ford's single source

// shortest path algorithm.

#include <iostream>

#include <limits.h>

using namespace std;

// The main function that finds shortest

// distances from src to all other vertices

// using Bellman-Ford algorithm. The function

// also detects negative weight cycle

// The row graph[i] represents i-th edge with

// three values u, v and w.

void BellmanFord(int graph[][3], int V, int E,

int src)

{

// Initialize distance of all vertices as 0.

int dis[V];

for (int i = 0; i < V; i++)

dis[i] = INT\_MAX;

// initialize distance of source as 0

dis[src] = 0;

// Relax all edges |V| - 1 times. A simple

// shortest path from src to any other

// vertex can have at-most |V| - 1 edges

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

for (int j = 0; j < E; j++) {

if (dis[graph[j][0]] + graph[j][2] <

dis[graph[j][1]])

dis[graph[j][1]] =

dis[graph[j][0]] + graph[j][2];

}

}

// check for negative-weight cycles.

// The above step guarantees shortest

// distances if graph doesn't contain

// negative weight cycle. If we get a

// shorter path, then there is a cycle.

for (int i = 0; i < E; i++) {

int x = graph[i][0];

int y = graph[i][1];

int weight = graph[i][2];

if (dis[x] != INT\_MAX &&

dis[x] + weight < dis[y])

cout << "Graph contains negative"

" weight cycle"

<< endl;

}

cout << "Vertex Distance from Source" << endl;

for (int i = 0; i < V; i++)

cout << i << "\t\t" << dis[i] << endl;

}

// Driver program to test above functions

int main()

{

int V = 5; // Number of vertices in graph

int E = 8; // Number of edges in graph

// Every edge has three values (u, v, w) where

// the edge is from vertex u to v. And weight

// of the edge is w.

int graph[][3] = { { 0, 1, -1 }, { 0, 2, 4 },

{ 1, 2, 3 }, { 1, 3, 2 },

{ 1, 4, 2 }, { 3, 2, 5 },

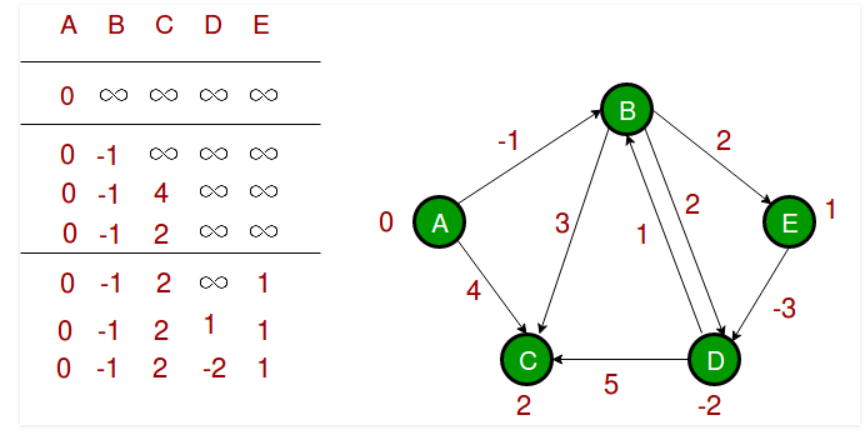
{ 3, 1, 1 }, { 4, 3, -3 } };

BellmanFord(graph, V, E, 0);

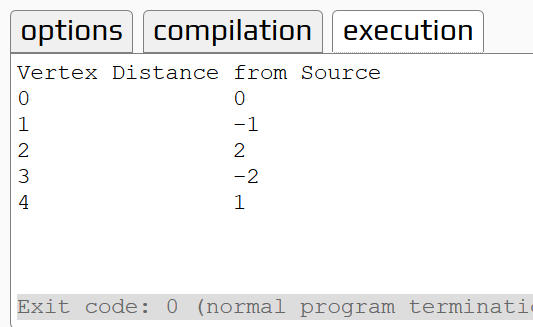
return 0;

}

**Problem**



**Output**



**EXPERIMENT NO. 10**

Write a program to solve 0/1 knapsack using dynamic programming.

**PROGRAM**

#include <iostream>

using namespace std;

int max(int x, int y) {

return (x > y) ? x : y;

}

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

int i, wt;

int kp[n+1];

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

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

for (wt = 0; wt <= W; wt++) {

if (i == 0 || wt == 0)

K[i][wt] = 0;

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

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];

}

int main() {

cout << "Enter the number of items in a Knapsack:";

int n, W;

cin >> n;

int v[n], w[n];

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

cout << "Enter value and weight for item " << i << ":";

cin >> v[i];

cin >> w[i];

}

cout << "Enter the capacity of knapsack";

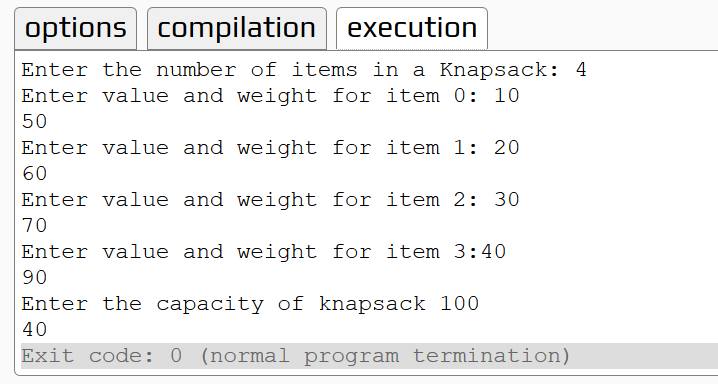
cin >> W;

cout << knapSack(W, w, v, n);

return 0;

}

**Output**



**EXPERIMENT NO. 11**

Write a program to find shortest path of multistage graph using dynamic programming.

**PROGRAM**

#include<iostream>

using namespace std;

struct fwd

{

int l;

int a[20];

};

struct fwd s1[10];

int main()

{

int k,i,j,n,p[10],m,z,cost[50],v,c[20][20];

cout<<"Enter the no. of stages :";

cin>>k;

n=0;

//Get the input for vertices

for(i=1;i<=k;i++)

{

cout<<"Enter no. of vertices in stage "<<i<<" :";

cin>>s1[i].l;

n+=s1[i].l;

for(j=1;j<=s1[i].l;j++){

cout<<"Enter the value of vertex "<<j<<" :";

cin>>s1[i].a[j];

}}

//Get the input for cost matrix

for(i=1;i<k;i++){

for(j=s1[i].a[1];j<=s1[i].a[s1[i].l];j++){

for(z=s1[i+1].a[1];z<=s1[i+1].a[s1[i+1].l];z++){

cout<<"Enter the cost of c["<<j<<"]["<<z<<"] :";

cin>>c[j][z];

}}}

//fwd approach

cost[n]=0;

int min,d[50],t;

for(i=k-1;i>=1;i--){

for(j=s1[i].a[1];j<=s1[i].a[s1[i].l];j++){

min=999;

for(z=s1[i+1].a[1];z<=s1[i+1].a[s1[i].l];z++){

if(cost[z]+c[j][z]<min)

min=cost[z]+c[j][z];

t=z;

}

cost[j]=min;

d[j]=t;

}}

//To display the path

p[1]=1;

p[k]=n;

for(i=2;i<k;i++){

p[i]=d[p[i-1]];

}

for(i=1;i<k;i++){

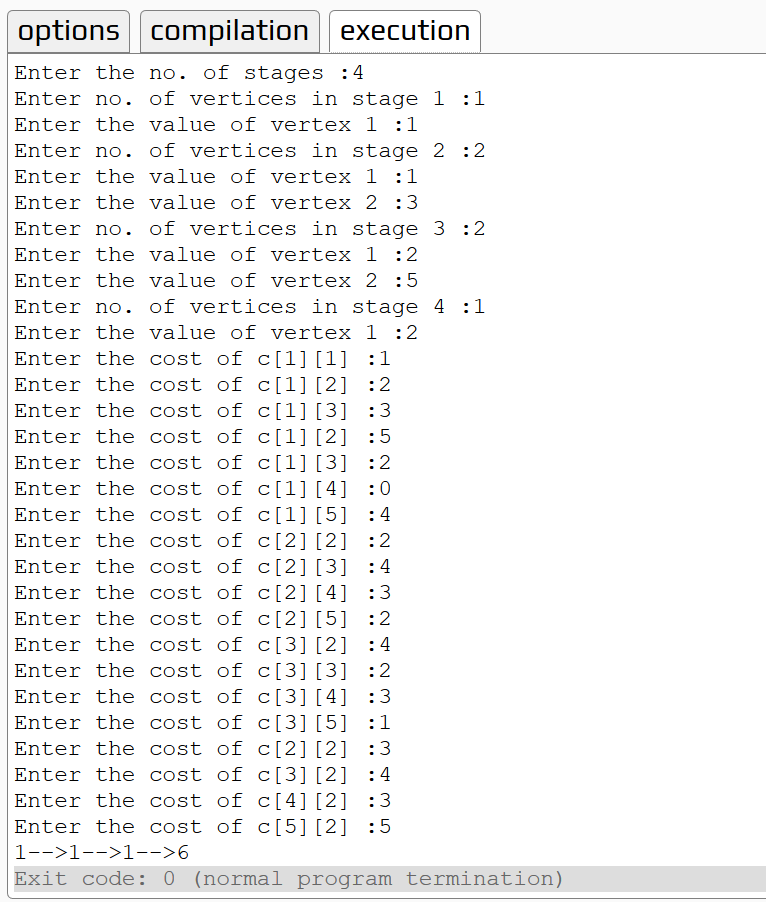
cout<<p[i]<<"-->";}

cout<<p[k];

return 0;

}

**Output**



**EXPERIMENT NO. 12**

Write a program to find minimum distance between different cities of your state using Floyd-War shall algorithm.

**PROGRAM**

// C++ Program for Floyd Warshall Algorithm

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph

#define V 4

/\* Define Infinite as a large enough

value.This value will be used for

vertices not connected to each other \*/

#define INF 99999

// A function to print the solution matrix

void printSolution(int dist[][V]);

// Solves the all-pairs shortest path

// problem using Floyd Warshall algorithm

void floydWarshall (int graph[][V])

{

/\* dist[][] will be the output matrix

that will finally have the shortest

distances between every pair of vertices \*/

int dist[V][V], i, j, k;

/\* Initialize the solution matrix same

as input graph matrix. Or we can say

the initial values of shortest distances

are based on shortest paths considering

no intermediate vertex. \*/

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

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

dist[i][j] = graph[i][j];

/\* Add all vertices one by one to

the set of intermediate vertices.

---> Before start of an iteration,

we have shortest distances between all

pairs of vertices such that the

shortest distances consider only the

vertices in set {0, 1, 2, .. k-1} as

intermediate vertices.

----> After the end of an iteration,

vertex no. k is added to the set of

intermediate vertices and the set becomes {0, 1, 2, .. k} \*/

for (k = 0; k < V; k++)

{

// Pick all vertices as source one by one

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

{

// Pick all vertices as destination for the

// above picked source

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

{

// If vertex k is on the shortest path from

// i to j, then update the value of dist[i][j]

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

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

}

}

}

// Print the shortest distance matrix

printSolution(dist);

}

/\* A utility function to print solution \*/

void printSolution(int dist[][V])

{

cout<<"The following matrix shows the shortest distances"

" between every pair of vertices \n";

for (int i = 0; i < V; i++)

{

for (int j = 0; j < V; j++)

{

if (dist[i][j] == INF)

cout<<"INF"<<" ";

else

cout<<dist[i][j]<<" ";

}

cout<<endl;

}

}

// Driver code

int main()

{ /\* Let us create the following weighted graph

10

(0)------->(3)

| /|\

5 | |

| | 1

\|/ |

(1)------->(2)

3 \*/

int graph[V][V]= { {0, 5, INF, 10},

{INF, 0, 3, INF},

{INF, INF, 0, 1},

{INF, INF, INF, 0}

};

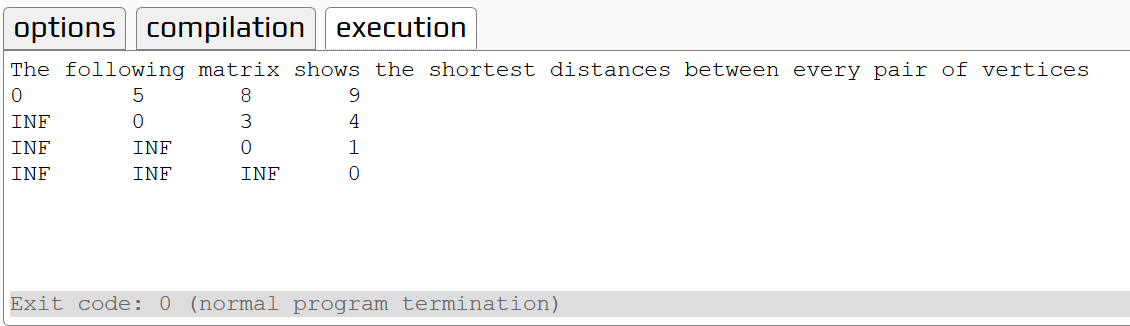
// Print the solution

floydWarshall(graph);

return 0;

}

**Output**



**EXPERIMENT NO. 13**

Write a program to find the solution to the 8-queen problem using backtracking.

**PROGRAM**

#include<iostream>

using namespace std;

#define N 8

void printBoard(int board[N][N]) {

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

cout << board[i][j] << " ";

cout << endl;

}

}

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

for (int i = 0; i < col; i++) //check whether there is queen in the left or not

if (board[row][i])

return false;

for (int i=row, j=col; i>=0 && j>=0; i--, j--)

if (board[i][j]) //check whether there is queen in the left upper diagonal or not

return false;

for (int i=row, j=col; j>=0 && i<N; i++, j--)

if (board[i][j]) //check whether there is queen in the left lower diagonal or not

return false;

return true;

}

bool solveNQueen(int board[N][N], int col) {

if (col >= N) //when N queens are placed successfully

return true;

for (int i = 0; i < N; i++) { //for each row, check placing of queen is possible or not

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

board[i][col] = 1; //if validate, place the queen at place (i, col)

if ( solveNQueen(board, col + 1)) //Go for the other columns recursively

return true;

board[i][col] = 0; //When no place is vacant remove that queen

}

}

return false; //when no possible order is found

}

bool checkSolution() {

int board[N][N];

for(int i = 0; i<N; i++)

for(int j = 0; j<N; j++)

board[i][j] = 0; //set all elements to 0

if ( solveNQueen(board, 0) == false ) { //starting from 0th column

cout << "Solution does not exist";

return false;

}

printBoard(board);

return true;

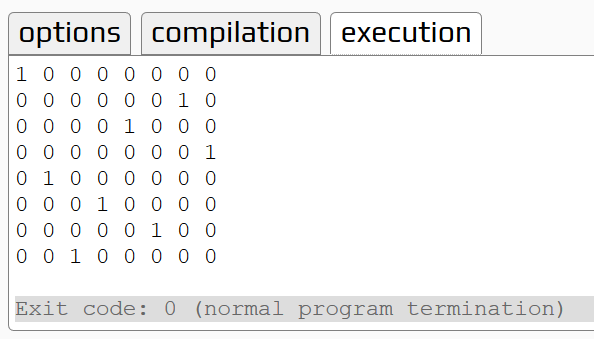
}

int main() {

checkSolution();

}

**Output**



**EXPERIMENT NO. 14**

Write a program to solve subset problem using Backtracking.

**PROGRAM**

#include <iostream>

using namespace std;

void displaySubset(int subSet[], int size) {

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

cout << subSet[i] << " ";}

cout << endl;}

void subsetSum(int set[], int subSet[], int n, int subSize, int total, int nodeCount ,int sum) {

if( total == sum) {

displaySubset(subSet, subSize); //print the subset

subsetSum(set,subSet,n,subSize-1,total-set[nodeCount],nodeCount+1,sum); //for other subsets

return;

}else {

for( int i = nodeCount; i < n; i++ ) { //find node along breadth

subSet[subSize] = set[i];

subsetSum(set,subSet,n,subSize+1,total+set[i],i+1,sum); //do for next node in depth

}}}

void findSubset(int set[], int size, int sum) {

int \*subSet = new int[size]; //create subset array to pass parameter of subsetSum

subsetSum(set, subSet, size, 0, 0, 0, sum);

delete[] subSet;

}

int main() {

int size;

cout<<"Enter the size of the problem"<<endl;

cin>>size;

int weights[size];

//= {10, 7, 5, 18, 12, 20, 15};

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

cout<<"enter the weight"<<i<<": ";

cin>>weights[i];

}

int sum;

cout<<"Enter the sum of the problem: ";

cin>>sum;

//= 7;

findSubset(weights, size, sum);

}

**Output**

