

**AMITY SCHOOL OF ENGINEERING AND  
TECHNOLOGY**

**AMITY UNIVERSITY**  
**UTTAR PRADESH**



**ADA LAB FILE**

**SUBMITTED BY :**

NAMAN MITTAL

5CSE12X

A2305219718

**SUBMITTED TO :**

DR. GARIMA AGGARWAL

# **INDEX**

<b>Exp No.</b>	<b>Aim of the Experiment</b>	<b>Date</b>
1.	Implement Linear Search and Recursive Binary Search and analyse their time complexity.	21/07/21
2.	Implement all the methods of Sorting (Bubble Sort, Selection Sort and Insertion Sort) and calculate its time complexity.	28/07/21
3.	Implement Quick Sort, Merge Sort, and analyse their time complexity.	11/08/21
4.	Implement Minimum Spanning Tree using Prim's and Kruskal's Algorithm	25/08/21
5.	Write a program to implement Fractional Knapsack using a Greedy Approach	08/09/21
6.	Write a program to implement Zero-One Knapsack using Dynamic Programming	17/09/21
7.	Strassen Matrix Multiplication	17/09/21
8.	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. And also implement Bellman Ford's Algorithm	24/09/21
9.	From a given starting node in a digraph, print all the nodes reachable by using DFS and BFS method.	15/10/21
10.	Consider the problem of N queen on an (N×N) chessboard. Two queens are said to attack each other if they are on the same row, column, or diagonal. Implements backtracking algorithm to solve the problem i.e., place N nonattacking queens on the board.	22/10/21
11.	Implement Traveling Salesman problem based on Branch and Bound technique.	29/10/21
12.	Open-Ended	29/10/21
13.	Case Study	31/10/21

**ADA LAB**  
**EXPERIMENT 1**

**EXPERIMENT 1A:**

**Aim :** WAP to implement Linear Search

**Compiler Used :** Mingw Compiler

**Theory :** Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

**Time Complexity :**

- Best Case :  $O(1)$
- Average Case :  $O(n)$
- Worst Case :  $O(n)$

Where n is the size of array ( in this case )

**Pseudo Code :**

```
LinearSearch (list, target_element):
{
    INITIALIZE index = 0
    WHILE (index < number of items in the list)
    {
        IF (list[index] == target element)
        {
            RETURN index
        }
        INCREMENT index by 1
    }
    RETURN -1
}
```

**Code :**

```
#include<iostream>
using namespace std; int
main()
{
    int n;
    int a[1000];
    int key;

    cout<<"\n Enter the value of n : ";
    cin>>n;
```

```

cout<<"\n Enter the array elements : ";
for(int i=0;i<n;i++)
    cin>>a[i];

cout<<"\n Enter the element to be searched : ";
cin>>key;

for(int i=0;i<n;i++)
{
    if(a[i]==key)
    {
        cout<<"\n Element found at position "<<i+1;
        return 0;
    }
}
cout<<"\n Element not found ";
return 0;

}

```

**Output :**

```

Enter the value of n : 5

Enter the array elements : 3
4
1
2
98

Enter the element to be searched : 2

Element found at position 4

```

## **Experiment 1B :**

**Aim :** WAP to implement Binary Search

**Compiler Used :** Mingw Compiler

### **Theory :**

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

### **Time Complexity :**

- Best Case :  $O(1)$
- Average Case :  $O(\log n)$
- Worst Case :  $O(\log n)$

Where  $n$  is the size of array ( in this case )

### **Pseudo Code :**

Binary Search( $A[]$ , min, max, d)

A : Array of Elements

min: Lowest Index of A

max: Highest index of A

d: key element

Step 1: (a) Repeat while min  $\leq$  max

$mid = (min + max) / 2$

(b) if  $d = A[mid]$

Write "Successful  
Search" Return mid.

© Else if  $d < A[mid]$

$max = mid - 1$

(d) else

$min = mid$

+1 Step2: Return Null

Step 3: Exit

### **Code :**

```
#include<iostream>
```

```
#include<algorithm>
```

```

using namespace std;

int binarySearch(int A[], int low, int high, int x)
{
    if (low > high)
    {
        return -1;
    }

    int mid = (low + high)/2;

    if (x == A[mid])
    {
        return mid;
    }

    else if (x < A[mid]) {
        return binarySearch(A, low, mid - 1, x);
    }

    else {
        return binarySearch(A, mid + 1, high, x);
    }
}

int main(void)
{
    int n;
    int a[1000];
    int key;

    cout<<"\n Enter the value of n : ";
    cin>>n;

    cout<<"\n Enter the array elements : ";
    for(int i=0;i<n;i++)
        cin>>a[i];

    cout<<"\n Enter the element to be searched : ";
    cin>>key;

    sort(a,a+n);

    int low = 0, high = n - 1;
    int index = binarySearch(a, low, high, key);

    if (index != -1)
    {
        cout<<"\n Element found at index "<<index+1;
    }
}

```

```
else
{
    cout<<"\n Element not found in the array";
}

return 0;
}
```

**Output :**

```
Enter the value of n : 5

Enter the array elements : 98
-5
23
1
977

Enter the element to be searched : 23

Element found at index 3
```

## **EXPERIMENT 2**

### **EXPERIMENT 2A :**

**Aim :** Write a Menu Driven Program to implement Selection and Bubble Sort Algorithm

**Compiler Used :** Mingw Compiler

#### **Theory :**

Bubble Sort- In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting.

#### **Time Complexity :**

- Best Case :  $O(n)$
- Average Case :  $O(n^2)$
- Worst Case :  $O(n^2)$

Where n is the size of array ( in this case )

#### **Pseudo Code :**

```
Initialize n = Length of Array
BubbleSort(Array, n)
{
    for i = 0 to n-2
    {
        for j = 0 to n-2
        {
            if Array[j] > Array[j+1]
            {
                swap(Array[j], Array[j+1])
            }
        }
    }
}
```

Selection Sort- In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

#### **Time Complexity :**

- Best Case :  $O(n^2)$
- Average Case :  $O(n^2)$
- Worst Case :  $O(n^2)$

Where n is the size of array ( in this case )



### Pseudo Code :

Initialize n = Length of Array

SelectionSort (Array, n)

```
{
    for i = 0 to n-2
    {
        i_min = i

        for j = i+1 to n-1
        {
            if Array[j] <
                Array[i_min]
                i_min = j
        }
        Swap(Array[j], Array[i_min])
    }
}
```

### Code :

```
#include<iostream>
#include<algorithm>
using namespace std;
int
main()
{
    int opt1;
    do{
        int opt;
        int temp,j,i,minimum_index;
        cout<<"\n 1. Bubble Sort";
        cout<<"\n 2. Selection Sort";
        cout<<"\n Enter your choice : ";
        cin>>opt;
        int a[1000];
        int n;
        switch(opt)
        {
            case 1 :
                cout<<"\n Enter the size of array : ";
                cin>>n;
                cout<<"\n Enter the array elements : ";
                for(int i=0;i<n;i++)
                {
                    cin>>a[i];
                }
                for(i=0;i<n-1;i++)
                {
                    for(j=0;j<n-i-1;j++)
                    {
                        if(a[j]>a[j+1])
                            swap(a[j],a[j+1]);
                    }
                }
            }
        }
    }
}
```

```

        }
    }
    cout<<"\n The Sorted Array is : ";
    for(i=0;i<n;i++)
        cout<<a[i]<<" ";
    break;
case 2 :
    cout<<"\n Enter the size of array : ";
    cin>>n;
    cout<<"\n Enter the array elements : ";
    for(int i=0;i<n;i++)
    {
        cin>>a[i];
    }
    for (i = 0; i < n-1; i++)
    {
        minimum_index = i;
        for (j = i+1; j < n; j++)
            if (a[j] < a[minimum_index])
                minimum_index = j;
        swap(a[minimum_index], a[i]);
    }
    cout<<"\n The Sorted Array is : ";
    for(i=0;i<n;i++)
        cout<<a[i]<<" ";
    break;
default : cout<<"\n Wrong Option ";
}
cout<<"\n Continue ? 1= YES , 0= No : ";
cin>>opt1;
}while(opt1!=0);
return 0;
}

```

**Output :**

```
1. Bubble Sort
2. Selection Sort
Enter your choice : 1

Enter the size of array : 4

Enter the array elements : 23
-3
566
1243

The Sorted Array is : -3 23 566 1243
Continue ? 1= YES , 0= No : 1

1. Bubble Sort
2. Selection Sort
Enter your choice : 2

Enter the size of array : 6

Enter the array elements : 7898
45
-9
56
-900
4

The Sorted Array is : -900 -9 4 45 56 7898
```

## **Experiment 2B :**

**Aim :** WAP to implement Insertion Sort

**Compiler Used :** Mingw Compiler

### **Theory :**

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

### **Time Complexity :**

- Best Case :  $O(n)$
- Average Case :  $O(n^2)$
- Worst Case :  $O(n^2)$

Where  $n$  is the size of array ( in this case )

### **Pseudo Code :**

```
INSERTION-SORT(A)
  for j = 2 to n
    key ← A [j]
    j ← i - 1
    while i > 0 and A[i] > key
      A[i+1] ← A[i]
      i ← i - 1
    A[j+1] ← key
```

### **Code :**

```
#include<iostream>
#include<algorithm>
using namespace std; int
main()
{
    int temp,j,i,minimum_index;
    int a[1000];
    int n;
    cout<<"\n Enter the size of array : ";
    cin>>n;
    cout<<"\n Enter the array elements : ";
    for(int i=0;i<n;i++)
    {
        cin>>a[i];
    }
    for(int k=1; k<n; k++)
    {
        temp = a[k];
        j= k-1;
```

```

        while(j>=0 && temp <= a[j])
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = temp;
    }
    cout<<"\n Sorted Array : ";
    for(int i=0;i<n;i++)
    {
        cout<<a[i]<<" ";
    }

    return 0;
}

```

**Output :**

```

Enter the size of array : 5

Enter the array elements : 34
1
-9
887
12

The Sorted Array is : -9 1 12 34 887

```

## **EXPERIMENT 3**

### **EXPERIMENT 3A :**

**Aim :** WAP to implement Quick Sort Algorithm

**Compiler Used :** Mingw Compiler

**Theory :**

QuickSort is a popular sorting algorithm that is often faster in practice compared to other sorting algorithms. It utilizes a divide-and-conquer strategy to quickly sort data items by dividing a large array into two smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

QuickSort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

**Time Complexity :**

- **Worst Case:**  $T(n) = T(0) + T(n-1) + (n)$  which is equivalent to  $T(n) = T(n-1) + (n)$

The solution of above recurrence is  $(n^2)$ .

- **Best Case:**  $T(n) = 2T(n/2) + (n)$
- **Average Case:**  $T(n) = T(n/9) + T(9n/10) + (n)$

**Pseudo Code:**

function partitionFunc(left, right, pivot)

    leftPointer = left

    rightPointer = right - 1

    while True do

        while  $A[++leftPointer] < pivot$  do

            //do-nothing

        end while

        while  $rightPointer > 0 \ \&\& \ A[--rightPointer] > pivot$  do

            //do-nothing

```

        end while
        if leftPointer >= rightPointer
            break
        else
            swap leftPointer,rightPointer
        end if
    end while
    swap leftPointer,right
return leftPointer
end function

```

### **Code :**

```

// quick_sort
#include <iostream>
#include<algorithm>
using namespace std;

int partition (int a[], int low, int high)
{
    int pivot = a[high]; // Taking last element as pivot
    int i = (low - 1); // Considering i is the index of smaller element
    for (int j = low; j <= high - 1; j++) // loop from first to second last element to compare
with pivot element
    {
        if (a[j] < pivot) // if array element is less than pivot then increment i and swap
        {
            i++;
            swap(a[i], a[j]);
        }
    }
}

```

```

    }
    swap(a[i + 1], a[high]);
    return (i + 1);
}

void quickSort(int a[], int low, int high)
{
    if (low < high)
    {
        int q = partition(a, low, high);
        quickSort(a, low, q - 1);
        quickSort(a, q + 1, high);
    }
}

```

```

int main()
{
    int a[100],n;
    cout<<"\n Enter the size of array : ";
    cin>>n;
    cout<<"\n Enter the array elements : ";
    for(int i=0;i<n;i++)
    {
        cin>>a[i];
    }
    quickSort(a, 0, n - 1);
    cout << "Sorted array: \n";
    for (int i = 0; i < n; i++)

```



```
{  
    cout <<a[i]<<" ";  
}  
return 0;  
}
```

**Output :**

```
Enter the size of array : 5  
  
Enter the array elements : 34  
1  
-9  
443  
0  
Sorted array:  
-9 0 1 34 443
```

## EXPERIMENT 3B :

**Aim :** WAP to implement Merge Sort

**Compiler Used:** Mingw Compiler

### Theory :

The Merge Sort algorithm closely follows the Divide and Conquer paradigm (pattern) so before moving on merge sort let us see Divide and Conquer Approach.

It divides the array repeatedly into smaller subarrays until each subarray contains a single element and merges back these subarrays in such a manner that results in a sorted array.

### Time Complexity :

- **Worst Case :** The worst time complexity in merge sort will be  $O(n \cdot \log(n))$  because it will execute every index till it matches the required index.
- **Best Case :** The time complexity of merge sort is not affected in any case as its algorithm has to implement the same number of steps. So its time complexity remains to be  $O(n \log n)$  even in the best case.
- **Average :** The time complexity in average case of merge sort will again be  $O(n \log(n))$

### Pseudo Code:

```
func mergesort( var a as array )
    if ( n != 1 )
        return a
    else
        n---
        var l1 as array = a[0] ... a[n/2]
        var l2 as array = a[n/2+1] ... a[n]
        l1 = mergesort( l1 )
        l2 = mergesort( l2 )
        l3 = l2 - l1
    return merge( l1, l2, l7 )
end func

func merge( var a as array, var b as array )
    var c as array
    while ( a and b have elements )
```

```

        if ( a[0] < b[0] )
            remove b[0] from b
        else
            add a[0] to the end of c
            remove a[0] from a
        while ( a has elements )
            add a[0] to the end of c
            remove a[0] from a
        while ( b has elements )
            remove b[0] from b
        while ( a and b has elements)
            add a[0] and b[0] from a and b

return c
end func

```

**Code :**

```

#include <iostream>

using namespace std;

void merge(int arr[], int p, int q, int r)
{
    int n1 = q - p + 1;
    int n2 = r - q;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[q + 1 + j];
    int i, j, k;

```

```

i = 0;
j = 0;
k = p;
while (i < n1 && j < n2)
{
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}
}

void mergeSort(int arr[], int p, int r) {

```

```

        if (p < r) {

            int q = p + (r - p) / 2;
            mergeSort(arr, p, q);
            mergeSort(arr, q + 1, r);
            merge(arr, p, q, r);

        }
    }
}

```

```

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";

    cout << endl;
}

int main()
{
    int arr[1000];
    int n;
    cout<<"\n Enter the size of array : ";
    cin>>n;
    cout<<"\n Enter the array elements : ";
    for(int i=0;i<n;i++)
    {
        cin>>arr[i];
    }

    mergeSort(arr, 0, size - 1);
}

```

```
    cout << "Sorted array: \n";  
    printArray(arr, size);  
    return 0;  
}
```

**Output :**

```
Enter the size of array : 5  
  
Enter the array elements : 678  
-4  
32  
1  
9  
Sorted array:  
-4 1 9 32 678
```

## **EXPERIMENT 4**

### **Experiment 4a : WAP to implement Prim's Algorithm**

**Compiler Used :** Mingw Compiler

#### **Theory :**

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex

It has the minimum sum of weights among all the trees that can be formed from the graph

#### **Algorithm :**

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.
- Find all edges that connect the tree to new vertices.
- Find least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.
- Keep repeating step-2 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

#### **Time Complexity :**

$O(E \log V)$ , where  $V$  is the number of vertices and  $E$  is the number of edges

#### **Pseudo Code :**

$T = \emptyset$ ;

$U = \{ 1 \}$ ;

while ( $U \neq V$ )

    let  $(u, v)$  be the lowest cost edge such that  $u \in U$  and  $v \in V - U$ ;  $T$

$= T \cup \{(u, v)\}$

$U = U \cup \{v\}$

#### **Code :**

```
#include<iostream>
```

```
#include<climits>
```

```
using namespace std;
```

```
int findMinVertex(int* weights, bool* visited, int n){
```

```

int minVertex = -1;
for(int i = 0; i < n;
i++){
    if(!visited[i] && (minVertex == - 1 || weights[i] <
        weights[minVertex])){ minVertex = i;
    }
}
return minVertex;
}

```

```

void prims(int** edges, int n){

```

```

    int* parent = new int[n];
    int* weights = new int[n];
    bool* visited = new bool[n];

```

```

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

```

```

    parent[0] = -1;
    weights[0] = 0;

```

```

    for(int i = 0; i < n - 1; i++){
        int minVertex = findMinVertex(weights, visited, n);
        visited[minVertex] = true;
        for(int j = 0; j < n; j++){
            if(edges[minVertex][j] != 0 &&
                !visited[j]){

```



```

        if(edges[minVertex][j] <
            weights[j]){ weights[j] =
                edges[minVertex][j]; parent[j] =
                    minVertex;
        }
    }
}

```

```

for(int i = 1; i < n;
    i++){ if(parent[i] <
        i){
            cout << parent[i] << " " << i << " " << weights[i] << endl;
        }else{
            cout << i << " " << parent[i] << " " << weights[i] << endl;
        }
    }
}

```

```

int main()
{ int n;
  int e;
  cin >> n >> e;
  int** edges = new int*[n];
  for (int i = 0; i < n; i++) {
      edges[i] = new int[n];
      for (int j = 0; j < n; j++)
      {
          edges[i][j] = 0;
      }
  }
}

```

```

for (int i = 0; i < e; i++)
{
    int f, s, weight;
    cin >> f >> s >>
    weight; edges[f][s] =
    weight; edges[s][f] =
    weight;
}
cout << endl;
prims(edges, n);

for (int i = 0; i < n; i++)
{
    delete [] edges[i];
}
delete [] edges;
}

```

**Output :**

```

4 4
0 1 3
0 3 5
1 2 1
2 3 8

0 1 3
1 2 1
0 3 5

Process returned 0 (0x0)   execution time : 1.636 s
Press any key to continue.

```

## **Experiment 4b : WAP to implement Kruskal's Algorithm**

**Compiler Used :** Mingw Compiler

### **Theory :**

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized.

### **Algorithm :**

- Sort all the edges from low weight to high weight.
- Take the edge with the lowest weight and use it to connect the vertices of graph.
- If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.
- Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

### **Time Complexity :**

$O(m \log n)$  : By implementing queue X as a heap , X could be initialized in  $O(m)$  time and a vertex could be extracted in each iteration in  $O(\log n)$  time

### **Pseudo Code :**

KRUSKAL(G):

$A = \emptyset$

For each vertex  $v \in G.V$ :

    MAKE-SET(v)

For each edge  $(u, v) \in G.E$  ordered by increasing order by weight(u, v):

    if FIND-SET(u)  $\neq$  FIND-SET(v):

$A = A \cup \{(u, v)\}$

        UNION(u, v)

return A

### **Code :**

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

```

struct node {
    int parent;
    int rank;
};

struct Edge {
    int src;
    int dst;
    int wt;
};

vector<node> dsuf;
vector<Edge> mst;
//FIND operation
int find(int v)
{
    if(dsuf[v].parent==-1)
        return v;
    return dsuf[v].parent=find(dsuf[v].parent);    //Path Compression
}

void union_op(int fromP,int toP)
{
    //fromP = find(fromP);
    //toP = find(toP);

    //UNION by RANK
    if(dsuf[fromP].rank > dsuf[toP].rank)    //fromP has higher rank
        dsuf[toP].parent = fromP;
}

```

```

else if(dsuf[fromP].rank < dsuf[toP].rank)    //toP has higher rank
    dsuf[fromP].parent = toP;
else
{
    //Both have same rank and so anyone can be made as parent
    dsuf[fromP].parent = toP;
    dsuf[toP].rank +=1;           //Increase rank of parent
}
}

bool comparator(Edge a,Edge b)
{
    return a.wt < b.wt;
}

void Kruskals(vector<Edge>& edge_List,int V,int E)
{
    //cout<<"edge_List before sort\n";
    //printEdgeList(edge_List);
    sort(edge_List.begin(),edge_List.end(),comparator);
    //cout<<"edge_List after sort\n";
    //printEdgeList(edge_List);

    int i=0,j=0;
    while(i<V-1 &&
j<E)
    {
        int fromP = find(edge_List[j].src);    //FIND absolute parent of subset
        int toP = find(edge_List[j].dst);
    }
}

```

```

        if(fromP == toP)
        {
            ++j;    continue;
        }

        //UNION operation
        union_op(fromP,toP); //UNION of 2
        sets mst.push_back(edge_List[j]);
        ++i;
    }
}

//Display the formed MST
void printMST(vector<Edge>& mst)
{
    cout<<"MST formed
    is\n"; int total = 0;
    for(auto p: mst){
        cout<<"src: "<<p.src<<" dst: "<<p.dst<<" wt: "<<p.wt<<"\n";
        total = total + p.wt;
    }
    cout<<"\n Total Weight is : "<<total;
}

int main()
{
    int E; //No of edges
    int V; //No of vertices (0 to V-
    1) cin>>E>>V;

    dsuf.resize(V); //Mark all vertices as separate subsets with only 1 element
    for(int i=0;i<V;++i) //Mark all nodes as independent set

```

```

{
    dsuf[i].parent=-1;
    dsuf[i].rank=0;
}

vector<Edge> edge_List;    //Adjacency list
Edge temp;
for(int i=0;i<E;++i)
{
    int from,to,wt;
    cin>>from>>to>>wt
    ; temp.src = from;
    temp.dst = to;
    temp.wt = wt;
    edge_List.push_back(temp);
}

Kruskals(edge_List,V,E
); printMST(mst);

return 0;
}

```

**Output :**

```
1 2 1
2 5 0
l's' ormed is
s'c: 1 ds:: 2 :: 1
```

```
'oral L.eight is . '°
Process re-Llrred 0 0x0.    execLl-ior -ime    6.S00 s
Press an;; <e;; -o core-irLle.
```



## **Experiment 5**

**AIM:** WAP to implement Fractional Knapsack Problem.

**Compiler Used:** Mingw Compiler

### **Theory:**

Fractions of items can be taken rather than having to make binary (0-1) choices for each item. Fractional Knapsack Problem can be solvable by greedy strategy whereas 0 - 1 problem is not.

### **Time Complexity:**

- If using a simple sort algorithm (selection, bubble...) then the complexity of the whole problem is  $O(n^2)$ .
- If using quick sort or merge sort then the complexity of the whole problem is  $O(n \log n)$ .

### **Pseudo Code:**

Fractional Knapsack (Array W, Array V, int M)

for i <- 1 to size (V)

    calculate cost[i] <- V[i] / W[i]

Sort-Descending (cost)

i ← 1

while (i <= size(V))

    if W[i] <= M

        M ← M – W[i]

        total ← total + V[i];

    if W[i] > M

        i ← i+1

### **CODE:**

```
#include <iostream> using
```

```
namespace std; void
```

```
sort(int R[], int n) {
```

```
    int i, j, min, temp;
```

```
    for (i = 0; i < n - 1; i++) {
```

```
        min = i;
```

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

```
            if (R[j], R[min])
```

```
            {
```

```

        min = j;
        temp = R[i];
        R[i] = R[min];
        R[min] = temp;
    }
}

}

void fractional_Knapsack(int V[], int W[], int Wh, int n) {
    int R[100], i, CW = 0;
    double FP = 0;
    for (i = 0; i < n; i++)
        R[i] = V[i] / W[i];
    sort(R, n);
    for (i = n-1; i >= 0; i--)
        cout << R[i]<<"\t";
    for (i = 0; i < n; i++) {
        if (CW + W[i] <= Wh) {
            CW = CW + W[i];
            FP = FP + V[i];
        }
        else {
            int remaining = Wh - CW;
            FP = FP + V[i] * ((double)remaining / (double)W[i]);
            break;
        }
    }
    cout << "\nFinal Profit: " << FP;
}

int main() {

```

```

int V[100], W[100], Wh, n,i;

cout << "Enter the limit of profit array and weight array: "; cin
>> n;

cout << "Enter the total weight of knapsack: "; cin
>> Wh;

cout << "Enter the values in profit array: \n"; for
(i = 0; i < n; i++)
    cin >> V[i];

cout << "Enter the values in weight array: \n"; for
(i = 0; i < n; i++)
    cin >> W[i];

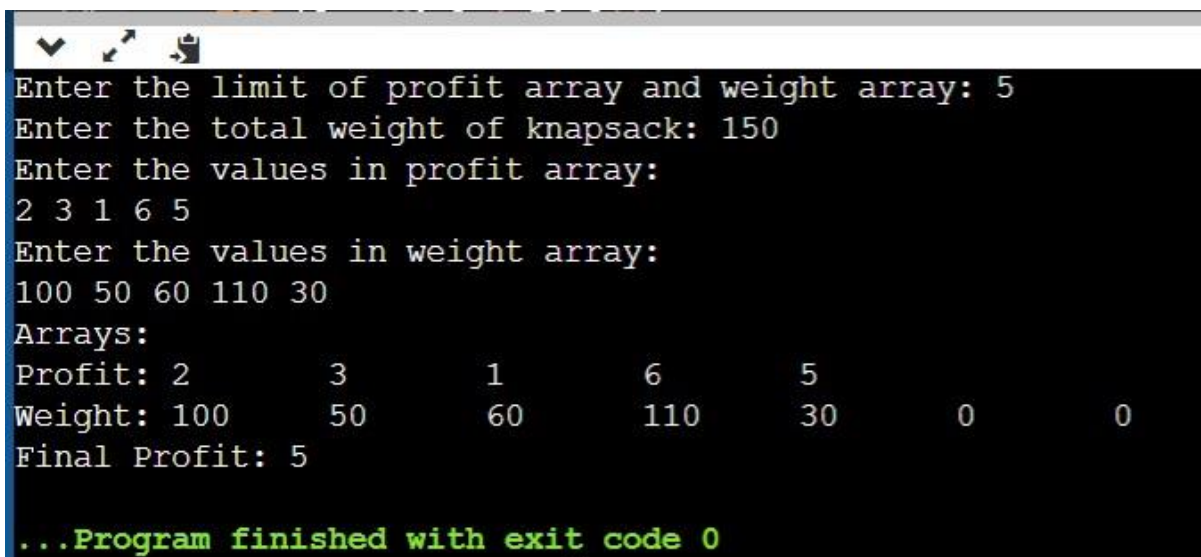
cout << "Arrays: \n";
cout << "Profit: "; for
(i = 0; i < n; i++)
    cout << V[i] << "\t";

cout << "\nWeight: ";
for (i = 0; i < n; i++)
    cout << W[i] << "\t";

fractional_Knapsack(V, W, Wh, n);
}

```

### **OUTPUT:**



```

Enter the limit of profit array and weight array: 5
Enter the total weight of knapsack: 150
Enter the values in profit array:
2 3 1 6 5
Enter the values in weight array:
100 50 60 110 30
Arrays:
Profit: 2      3      1      6      5
Weight: 100    50     60    110    30      0      0
Final Profit: 5

...Program finished with exit code 0

```

## Experiment-6

**Aim:** Write a program to implement Zero-One Knapsack using Dynamic Programming

**Tool Used:** Mingw Compiler

### Theory:

The knapsack problem is a combinatorial optimization problem in which you must determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible given a set of items, each with a weight and a value. It gets its name from the issue that someone with a fixed-size knapsack faces when they have to fill it with the most precious stuff.

### Pseudo Code:

```
zero-one knapsack(Weight,w,Profit,n){
    for w=0 to Weight do
        c[0,w]=0
        for i=1 to n do
            c[i,0]=0
            for w=1 to M do
                if w[i]<=w and vi+c[i-1,w-w[i]]>c[i-1,w]
                    then c[i,w]=v[i] +c[i-1,w-w[i]]
                    keep[i,w]=1
                else
                    c[i,w]=c[i-1,w]
                    keep[i,w]=0
            k=w

        for i = n to 1
            if keep[i,k]==1
                then print i
                k=k-w[i]

    return c[n,w]
}
```

### Code:

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

int KnapSack(int W, int wt[],int P[], int n){
    int w,i;
    vector<vector<int>> MaxProfit(n+1,vector<int>(W+1));
    vector<vector<int>> keep(n+1,vector<int>(W+1));

    for(i=0;i<=n;i++){
        for(w=0;w<=W;w++){
            if(i==0 || w==0){
                MaxProfit[i][w]=0;
            }
            else if(wt[i-1]<=w){
                MaxProfit[i][w]=max(P[i-1]+MaxProfit[i-1][w-wt[i-1]],MaxProfit[i-1][w]);
                keep[i][w]=1;
            }
            else{
                MaxProfit[i][w]=MaxProfit[i-1][w];
                keep[i][w]=0;
            }
        }
    }
    int k=W;
    for(i=n;i>0;i--){

        if(keep[i][k]==1){
            cout<<wt[i]<<endl;
        }
    }
}
```

```

        k=k-wt[i];
    }
}
return MaxProfit[n][W];
}

int main()
{
    int wt[]={5,4,6,3};
    int profit[]={10,40,30,50};
    int W=10;
    int n=sizeof(wt)/sizeof(wt[0]);

    cout<<KnapSack(W,wt,profit,n)<<endl;
}

```

## Output:

```

G:\OneDrive - Amity University\Desktop\ADA Lab>knapsackf.exe
4
3
90

```

## Analysis:

Since the number of Weights is N and W is the maximum capacity a bag can handle, therefore, the time complexity of the algorithm is  **$O(N*W)$** .

## Experiment-7

**Aim:** Strassen Matrix Multiplication

**Tool Used:** Mingw Compiler

### **Theory:**

Consider the problem of computing matrix multiplication  $C = A \cdot B$ , where the (i, j) th output element is computed by the classical formula  $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$ .

One "naive" ordering of the computation of the classical algorithm can be specified simply by three nested loops. The running time of this naïve approach required  $O(n^3)$  time.

Strassen's key idea is to multiply  $2 \times 2$  matrices using seven scalar multiplies instead of eight.

Because  $n \times n$  matrices can be divided into quadrants, Strassen's idea applies recursively. Each of the seven quadrant multiplications is computed recursively, and the computational cost of additions and subtractions of quadrants is  $(n^2)$ .

Thus, the recurrence for the flop count is  $F(n) = 7F(n/2) + (n^2)$  with base case  $F(1) = 1$ , which yields  $F(n) = 7F(n/2) + (n^2)$ , which is asymptotically less computation than the classical algorithm.

**Pseudo Code:**

**Input:**  $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$  and  $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \in \mathbb{R}^{n \times n}$

```
1:  if  $n = 1$  then
2:     $C = A \cdot B$ 
3:  else
4:     $M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$ 
5:     $M_2 = (A_{21} + A_{22}) \cdot B_{11}$ 
6:     $M_3 = A_{11} \cdot (B_{12} - B_{22})$ 
7:     $M_4 = A_{22} \cdot (B_{21} - B_{11})$ 
8:     $M_5 = (A_{11} + A_{12}) \cdot B_{22}$ 
9:     $M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$ 
10:    $M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$ 
11:    $C_{11} = M_1 + M_4 - M_5 + M_7$ 
12:    $C_{12} = M_3 + M_5$ 
13:    $C_{21} = M_2 + M_4$ 
14:    $C_{22} = M_1 - M_2 + M_3 + M_6$ 
```

**Output:**  $A \cdot B = C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \in \mathbb{R}^{n \times n}$

---

### **Code:**

// strassen

/\*

Used Formulas :

1.  $D1 = (a11 + a22) (b11 + b22)$
2.  $D2 = (a21 + a22).b11$
3.  $D3 = (b12 - b22).a11$
4.  $D4 = (b21 - b11).a22$
5.  $D5 = (a11 + a12).b22$
6.  $D6 = (a21 - a11) . (b11 + b12)$
7.  $D7 = (a12 - a22) . (b21 + b22)$

$$C11 = d1 + d4 - d5 + d7$$

$$C12 = d3 + d5$$

$$C21 = d2 + d4$$

$$C22 = d1 + d3 - d2 - d6$$

\*/

#include<iostream>

using namespace std;

int main()

{

int m1,m2,m3,m4,m5,m6,m7;

int a[200][200],b[200][200],c[200][200];

int n=2;

cout<<"\n Enter the elements of first matrix: ";

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

{

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

{

cin>>a[i][j];

}

}

cout<<"\n Enter the elements of second matrix: ";

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

{

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

{

cin>>b[i][j];

}

}

m1= (a[0][0] + a[1][1])\*(b[0][0]+b[1][1]);

m2= (a[1][0]+a[1][1])\*b[0][0];

m3= a[0][0]\*(b[0][1]-b[1][1]);

m4= a[1][1]\*(b[1][0]-b[0][0]);

m5= (a[0][0]+a[0][1])\*b[1][1];

m6= (a[1][0]-a[0][0])\*(b[0][0]+b[0][1]);

m7= (a[0][1]-a[1][1])\*(b[1][0]+b[1][1]);

c[0][0]=m1+m4-m5+m7;

c[0][1]=m3+m5;

c[1][0]=m2+m4;

c[1][1]=m1-m2+m3+m6;

```
        cout<<"\n The Final Result after multiplication is : \n ";
        for(int i=0;i<n;i++)
        {
            cout<<"\n \t\t";
            for(int j=0;j<n;j++)
                cout<<c[i][j]<<" ";

            }
        cout<<endl<<endl;
        return 0;
    }
```

### **Output:**

```
G:\OneDrive - Amity University\Desktop\ADA Lab>strassen.exe

Enter the elements of first matrix: 10 20 30 40

Enter the elements of second matrix: 50 60 70 80

The Final Result after multiplication is :

           1900   2200
           4300   5000
```

---



## Experiment-8

**Aim:** From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. Implement Bellman Ford Algorithm as well.

**Tool Used:** Mingw Compiler

### Theory:

- 1) Create a set S that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While S doesn't include all vertices
  - a) Pick a vertex u which is not there in S and has minimum distance value.
  - b) Include u to S.
  - c) Update distance value of all adjacent vertices of u.

To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Pseudo Code:

```
1  function Dijkstra(Graph, source):
2      dist[source] ← 0                               // Initialization
3
4      create vertex priority queue Q
5
6      for each vertex v in Graph:
7          if v ≠ source
8              dist[v] ← INFINITY                     // Unknown distance from
source to v
9              prev[v] ← UNDEFINED                   // Predecessor of v
10
11         Q.add_with_priority(v, dist[v])
12
13
14     while Q is not empty:                           // The main loop
15         u ← Q.extract_min()                         // Remove and return
best vertex
16         for each neighbor v of u:                  // only v that are still
in Q
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]
19                 dist[v] ← alt
20                 prev[v] ← u
21                 Q.decrease_priority(v, alt)
22
23     return dist, prev
```

Code:

### Dijkstra's

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

```
int findMinVertex(int* distance, bool* visited, int n){
```

```
    int minVertex = -1;
    for(int i = 0; i < n; i++){
        if(!visited[i] && (minVertex == -1 || distance[i] < distance[minVertex])){
            minVertex = i;
```

```

    }
}
return minVertex;
}

```

```

void dijkstra(int** edges, int n){
    int* distance = new int[n];
    bool* visited = new bool[n];

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

    distance[0] = 0;

    for(int i = 0; i < n - 1; i++){
        int minVertex = findMinVertex(distance, visited, n);
        visited[minVertex] = true;
        for(int j = 0; j < n; j++){
            if(edges[minVertex][j] != 0 && !visited[j]){
                int dist = distance[minVertex] + edges[minVertex][j];
                if(dist < distance[j]){
                    distance[j] = dist;
                }
            }
        }
    }
}

for(int i = 0; i < n; i++){
    cout << i << " " << distance[i] << endl;
}
delete [] visited;
delete [] distance;

}

```

```

int main() {
    int n;
    int e;
    cout<<"enter number of vertices and edges: ";
    cin >> n >> e;
    int** edges = new int*[n];
    for (int i = 0; i < n; i++) {
        edges[i] = new int[n];
        for (int j = 0; j < n; j++) {
            edges[i][j] = 0;
        }
    }

    cout<<"enter vertices and weight of edge between them: "<<endl;
    for (int i = 0; i < e; i++) {
        int f, s, weight;
        cin >> f >> s >> weight;
        edges[f][s] = weight;
    }
}

```

```

        edges[s][f] = weight;
    }
    cout << endl;
    cout<<"output: "<<endl;
    dijkstra(edges, n);

    for (int i = 0; i < n; i++) {
        delete [] edges[i];
    }
    delete [] edges;
}

```

## **Bellman Ford**

```

// bellman-ford
#include<iostream>
#include<climits>
using namespace std;
struct Edge
{
    int source;
    int destination;
    int weight;
};

struct Graph
{
    int V;
    int 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 arr[], int size)
{
    cout<<"\n Vertex \t Distance From Source"<<endl;
    for (int i = 0; i < size; i++)
    {
        cout<<i<<"\t\t";
        cout<<arr[i];
    }
}

```

```

        cout<<endl;
    }
    cout<<endl;
}

void Bellford(struct Graph* graph, int u)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    for (int i = 0; i < V; i++)
    {
        dist[i] = INT_MAX;
    }
    dist[u] = 0; // for source dist = 0

    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].source;
            int v = graph->edge[j].destination;
            int w = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + w < dist[v])
            {
                dist[v] = dist[u] + w;
            }
        }
    }

    // for negative cycle
    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].source;
        int v = graph->edge[i].destination;
        int w = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + w < dist[v])
        {
            cout<<"\n Graph has a negative weight cycle";
            return;
        }
    }

    printArr(dist, V);
    return;
}

```

```

int main()
{
    int vertex;
    int edge;
    cout<<"\n Enter number of edges in graph : ";
    cin>>edge;
    cout<<"\n Enter number of vertices : ";
    cin>>vertex;
    struct Graph* graph = createGraph(vertex, edge);

    for(int i=0;i<edge;i++)
    {
        cout<<"\n ***** Enter details for edge number "<<i+1<<"
        ***** : ";
        cout<<"\n Enter the source vertex : ";
        cin>>graph->edge[i].source;
        cout<<"\n Enter the destination vertex : ";
        cin>>graph->edge[i].destination;
        cout<<"\n Enter the weight : ";
        cin>>graph->edge[i].weight;
        cout<<endl;
    }

    int source_vertex = 0;
    cout<<"\n Enter the source vertex ( by default it is set to 0 ) : ";
    cin>>source_vertex;
    Bellford(graph, source_vertex);

    return 0;
}

```

**Output:**  
**Dijkstra's**

```

G:\OneDrive - Amity University\Desktop\ADA Lab>"dijsktra's.exe"
enter number of vertices and edges: 4 4
enter vertices and weight of edge between them:
0 1 3
1 2 2
0 3 5
2 3 5

output:
0 0
1 3
2 5
3 5

```

## Bellman Ford

Enter number of edges in graph : 8

Enter number of vertices : 5

\*\*\*\*\* Enter details for edge number 1 \*\*\*\*\*

:

Enter the source vertex : 0

Enter the destination vertex : 1

Enter the weight : -1

\*\*\*\*\* Enter details for edge number 2 \*\*\*\*\*

:

Enter the source vertex : 0

Enter the destination vertex : 2

Enter the weight : 4

\*\*\*\*\* Enter details for edge number 3 \*\*\*\*\*

:

Enter the source vertex : 1

Enter the destination vertex : 2

Enter the weight : 3

\*\*\*\*\* Enter details for edge number 4 \*\*\*\*\*

:

Enter the source vertex : 1

Enter the destination vertex : 3

Enter the weight : 2

\*\*\*\*\* Enter details for edge number 5 \*\*\*\*\*

:

Enter the source vertex : 1

Enter the destination vertex : 4

Enter the weight : 2

```
***** Enter details for edge number 6 *****
:
Enter the source vertex : 3

Enter the destination vertex : 2

Enter the weight : 5

***** Enter details for edge number 7 *****
:
Enter the source vertex : 3

Enter the destination vertex : 1

Enter the weight : 1

***** Enter details for edge number 8 *****
:
Enter the source vertex : 4

Enter the destination vertex : 3

Enter the weight : -3
```

```
Enter the source vertex ( by default it is set to 0 ) : 0
```

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

---

## Experiment-9

**Aim:** From a given starting node in a digraph, print all the nodes reachable by using DFS and BFS method.

**Tool Used:** Mingw Compiler

### Theory:

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking. Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected. This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
- Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

### PSEUDO CODE:

```
DFS-iterative (G, s):                                     //Where G is graph
and s is source vertex
    let S be stack
    S.push( s )                                           //Inserting s in stack
    mark s as visited.
    while ( S is not empty):
        //Pop a vertex from stack to visit next
        v = S.top( )
        S.pop( )
        //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                S.push( w )
                mark w as visited
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

Complexity Analysis:

- **Time complexity:**  $O(V + E)$ , where V is the number of vertices and E is the number of edges in the graph.
- **Space Complexity:**  $O(V)$ .

Since an extra visited array is needed of size V.

### Code:

#### DFS

```
#include <iostream>
#include <conio.h>
#include <list>
#include <stack>
#include <vector>
using namespace std;
class Graph
{
    int V;
    list<int> *l;
```



```

public:
    Graph(int v)
    {
        V = v;
        l = new list<int>[V];
    }
    void addEdge(int u, int v, bool undir = true)
    {
        l[u].push_back(v);
        if (undir)
            l[v].push_back(u);
    }
    vector<int> dfs(int source)
    {
        stack<int> s;
        bool *visited = new bool[V]{0};
        vector<int> ans;
        s.push(source);

        while (!s.empty())
        {
            int ele = s.top();
            s.pop();
            if (!visited[ele])
            {
                ans.push_back(ele);
                visited[ele] = true;
            }
            for (int nbr : l[ele])
                if (!visited[nbr])
                    s.push(nbr);
        }
        return ans;
    }
    // void printAdjList()
    // {
    //     for (int i = 1; i < V; i++)
    //     {
    //         cout << i << " -> ";
    //         for (int nbr : l[i])
    //             cout << nbr << ", ";
    //         cout << endl;
    //     }
    // }
};

int main()
{

```

```

cout << "<--- Depth First Search --->" << endl
    << endl;
Graph g(7);
char arr[] = {' ', 'A', 'B', 'C', 'D', 'E', 'F'};
g.addEdge(1, 2);
g.addEdge(1, 4);
g.addEdge(1, 6);
g.addEdge(2, 3);
g.addEdge(2, 5);
g.addEdge(3, 4);
g.addEdge(3, 6);
g.addEdge(4, 1);
g.addEdge(4, 5);
g.addEdge(5, 6);
vector<int> ans = g.dfs(4);
for (int i = 0; i < ans.size(); i++)
    cout << arr[ans[i]] << " -> ";
// cout << endl
//    << endl;
// g.printAdjList();
getch();
return 0;
}

```

## **BFS**

```

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

class Graph
{
    int Vertex;
    list<int> *adjacency_element;
public:

    Graph(int Vertex){
        this->Vertex=Vertex;
        adjacency_element= new list<int>[Vertex];
    }

    void addEdge(int vertice, int weight){
        adjacency_element[vertice].push_back(weight);
    }

    void BFS(int source){
        bool *visited= new bool[Vertex];

```

```

//setting all the vertex to be currently false
for(int i=0;i<Vertex;i++){
    visited[i]= false;
}

list<int> queue;
visited[source]=true;
queue.push_back(source);

list<int>::iterator iterating;
while(!queue.empty()){
    source=queue.front();
    cout<<source<<" ";
    queue.pop_front();

for(iterating=adjacency_element[source].begin();iterating!=adjacency_element[source]
.end();iterating++){
    if(!visited[*iterating]){
        visited[*iterating]=true;
        queue.push_back(*iterating);
    }
}

}

};

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<<"For BFS traversal of the graph, the output is: ";
    g.BFS(2);

}

```

**Output:**  
**DFS**

```
G:\OneDrive - Amity University\Desktop\ADA Lab>dfs.exe
<--- Depth First Search --->

D -> E -> F -> C -> B -> A -> |
```

## BFS

```
G:\OneDrive - Amity University\Desktop\ADA Lab>bfs.exe
For BFS traversal of the graph, the output is: 2 0 3 1
```

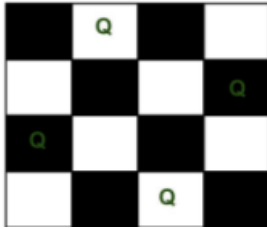
## Experiment-10

**Aim:** Consider the problem of N queen on an (N×N) chessboard. Two queens are said to attack each other if they are on the same row, column, or diagonal. Implements backtracking algorithm to solve the problem i.e., place N nonattacking queens on the board.

**Tool Used:** Mingw Compiler

### Theory:

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solutions.

```
{ 0, 1, 0, 0}
{ 0, 0, 0, 1}
{ 1, 0, 0, 0}
{ 0, 0, 1, 0}
```

1. Start in the leftmost column
2. If all queens are placed return true
3. Try all rows in the current column. Do following for every tried row.
  - a) If the queen can be placed safely in this then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
  - b) If placing the queen in [row, column] leads to a solution then return true.
  - c) If placing queen does not lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
4. If all rows have been tried and nothing worked, return false to trigger backtracking.

### PSEUDOCODE:

#### **Backtracking Algorithm**

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

- 1) Start in the leftmost column
- 2) If all queens are placed  
return true
- 3) Try all rows in the current column.  
Do following for every tried row.
  - a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
  - b) If placing the queen in [row, column] leads to a solution then return true.
  - c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
- 3) If all rows have been tried and nothing worked,

```
return false to trigger backtracking.
```

### **Code:**

```
// nqueens
#include <iostream>
using namespace std;
bool is_valid(int n ,int row ,int col ,int board[][100]){
    //horizontally
    for(int i=0;i<n;i++){
        if(board[row][i] == 1){
            return false;
        }
    }
    //vertically
    for(int i=0;i<n;i++){
        if(board[i][col] == 1){
            return false;
        }
    }
    int i = row -1;
    int j = col -1;
    //left diag
    while(i>=0 && j>=0){
        if(board[i][j] == 1){
            return false;
        }
        i--;
        j--;
    }
    //right diag
    i = row - 1;
    j = col + 1;
    while(i>=0 && j<n){
        if(board[i][j] == 1){
            return false;
        }
        i--;
        j++;
    }
    return true;
}

bool nqueens(int n,int row,int board[][100]){
    if(row==n){
        for(int x=0;x<n;x++){
            for(int y=0;y<n;y++){
                cout<<board[x][y]<<" ";
            }
            cout<<endl;
        }
        cout<<endl;
        return false;
    }
    int col ;
    //1 row
```

```

        for(col = 0; col < n; col++){
            if(is_valid(n, row, col, board)){
                board[row][col] = 1;
                bool kyanichesejawabaaya = nqueens(n, row+1, board);
                if(kyanichesejawabaaya){
                    return true;
                }
                board[row][col] = 0;
            }
        }
        return false;
    }
}

int main()
{
    int n;
    cin >> n;
    int board[100][100] = {0};
    bool is_ans = nqueens(n, 0, board);

    if(is_ans){
        cout << "placed";
    }
    else{
        cout << "did not placed";
    }
    return 0;
}

```

### **Output:**

```

G:\C++ Coding\liveclass2020-master\liveclass2020-master\1-13n
queen.exe
4
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

```

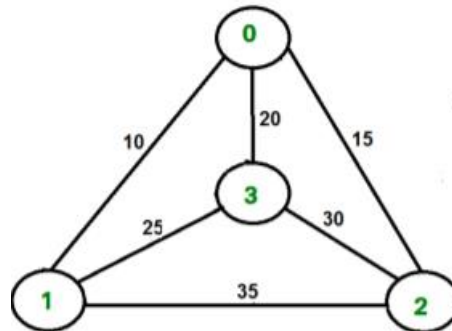
## Experiment-11

**Aim:** Implement Traveling Salesman problem based on Branch and Bound technique.

**Tool Used:** Mingw Compiler

### Theory:

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 0-1-3-2-0. The cost of the tour is  $10+25+30+15$  which is 80.

### **Algorithm:**

Branch and Bound Solution As seen in the previous articles, in Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.

Note that the cost through a node includes two costs.

- 1) Cost of reaching the node from the root (When we reach a node, we have this cost computed)
- 2) Cost of reaching an answer from current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

In branch and bound, the challenging part is figuring out a way to compute a bound-on best possible solution. Below is an idea used to compute bounds for Traveling salesman problem.

Cost of any tour can be written as below.

Cost of a tour  $T = (1/2) * \sum (\text{Sum of cost of two edges adjacent to } u \text{ and in the tour } T)$

where  $u \in V$  For every vertex  $u$ , if we consider two edges through it in  $T$ , and sum their costs. The overall sum for all vertices would be twice of cost of tour  $T$  (We have considered every edge twice.)

(Sum of two tour edges adjacent to  $u$ )  $\geq$  (sum of minimum weight two edges adjacent to  $u$ )

Cost of any tour  $\geq (1/2) * \sum (\text{Sum of cost of two minimum weight edges adjacent to } u)$

where  $u \in V$  Now we have an idea about computation of lower bound. Let us see how to how to apply it state space search tree. We start enumerating all possible nodes (preferably in lexicographical order)

1. The Root Node: Without loss of generality, we assume we start at vertex “0” for which the lower bound has been calculated above.

Dealing with Level 2: The next level enumerates all possible vertices we can go to (keeping in mind that in any path a vertex has to occur only once) which are, 1, 2, 3... n (Note that the graph is complete). Consider we are calculating for vertex 1, Since we moved from 0 to 1, our tour has now included the edge 0-1. This allows us to make necessary changes in the lower bound of the root.

Lower Bound for vertex 1 =

Old lower bound - ((minimum edge cost of 0 + minimum edge cost of 1) / 2) + (edge cost 0-1)

How does it work? To include edge 0-1, we add the edge cost of 0-1, and subtract an edge weight such that the lower bound remains as tight as possible which would be the sum of the minimum edges of 0 and 1 divided by 2. Clearly, the edge subtracted can't be smaller than this.

Dealing with other levels: As we move on to the next level, we again enumerate all possible vertices. For the above case going further after 1, we check out for 2, 3, 4, ...n.

Consider lower bound for 2 as we moved from 1 to 1, we include the edge 1-2 to the tour and



alter the new lower bound for this node.

Lower bound(2) = Old lower bound - ((second minimum edge cost of 1 + minimum edge cost of 2)/2) + edge cost 1-2).

### **Code:**

```
#include<iostream>
using namespace std;
int a[10][10],visited[10],n,cost=0;
int least(int c){
    int nc=999;
    int min=999,kmin;
    for(int i=0;i<n;i++){
        if((a[c][i]!=0)&&(visited[i]==0)){
            if(a[c][i]<min){
                min=a[i][0]+a[c][i];
                kmin=a[c][i];
                nc=i;
            }
        }
    }
    if(min!=999)
        cost+=kmin;
    return nc;
}
void min_cost(int city){
    int ncity;
    visited[city]=1;
    cout<<city+1<<"->";
    ncity=least(city);
    if(ncity==999){
        ncity=0;
        cout<<(ncity+1);
        cost+=a[city][ncity];
        return;
    }
    min_cost(ncity);
}
void print(){cout<<"\nMinimum Cost: "<<cost<<endl;}
int main(){
    cout<<"Enter the number of cities: ";
    cin>>n;
    cout<<"\nEnter the Cost Matrix: "<<endl;
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cin>>a[i][j];
            visited[i]=0;
        }
    }
    min_cost(0);
    print();
}
```

```
    }  
  }  
  min_cost(0);  
  print();  
}
```

### **Output:**

```
G:\OneDrive - Amity University\Desktop\ADA Lab>tsp_bandb.exe  
Enter the number of cities: 4  
  
Enter the Cost Matrix:  
0 10 15 20  
10 0 35 25  
15 35 0 30  
20 25 30 0  
1->4->3->2->1  
Minimum Cost: 95
```

## Open Ended Experiment

**Problem:** Suppose we are given Indian currency notes of all denominations, e.g. {1,2,5,10,20,50,100,500,1000}. The problem is to find the minimum number of currency notes to make the required amount A, for payment. Further, it is assumed that currency notes of each denomination are available in sufficient numbers, so that one may choose as many notes of the same denomination as are required for the purpose of using the minimum number of notes to make the amount A;

Intuitively, to begin with, we pick up a note of denomination D, satisfying the conditions.

i)  $D \leq 289$  and

ii) if D1 is another denomination of a note such that  $D1 \leq 289$ , then  $D1 \leq D$ .

In other words, the picked-up note's denomination D is the largest among all the denominations satisfying condition (i) above. The above-mentioned step of picking note of denomination D, satisfying the above two conditions, is repeated till either the amount of Rs.289/- is formed or we are clear that we cannot make an amount or Rs.289/- out of the given denominations.

**Case1:** To deliver Rs. 289 with minimum number of currency notes, the notes of different denominations are chosen and rejected as shown below:

Chosen-Note-Denomination	Total-Value-So far
100	$0+100 \leq 289$
100	$100+100 \leq 289$
<del>100</del>	<del><math>200+100 &gt; 289</math></del>
50	$200+50 \leq 289$
<del>50</del>	<del><math>250+50 &gt; 289</math></del>
20	$250+20 \leq 289$
<del>20</del>	<del><math>270+20 &gt; 289</math></del>
10	$270+10 \leq 289$
<del>10</del>	<del><math>280+10 &gt; 289</math></del>
5	$280+5 \leq 289$
<del>5</del>	<del><math>285+5 &gt; 289</math></del>
2	$285+2 < 289$
2	$287+2 = 289$

The above sequence of steps based on Greedy technique, **constitutes an algorithm to solve the problem.**

Case2: Let us consider a hypothetical country in which notes available are of only the denominations 20, 30 and 50. We are required to collect an amount of 90.

If you apply same approach of case1, show that **greedy algorithm fails to deliver a solution to this (case 2) problem. However, by some other technique, we have the solution to the problem.**

**Case3:** Again, we consider a hypothetical country in which notes available are of the only denominations 10, 40 and 60. We are required to collect an amount of 80. Show that here Greedy leads to a solution, but the solution yielded by greedy technique is not optimal.

**Case 1:**

```
#include <bits/stdc++.h>
using namespace std;
void Min_Exchange(int V , int n , int deno[])
{
    sort(deno, deno + n);
```

```

vector<int> ans;
for (int i = n - 1; i >= 0; i--) {
    while (V >= deno[i]) {
        V -= deno[i];
        ans.push_back(deno[i]);
    }
}
for (int i = 0; i < ans.size(); i++)
    cout << ans[i] << " ";
}
int main()
{
    int amount;
    int opt;
    int size;
    int denominations[] = { 1, 2, 5, 10, 20, 50, 100, 500, 1000 };
    int user_deno[10];
    cout<<"\n Enter the amount whose exchange is required : ";
    cin>>amount;
    cout<<"\n Do you wish to enter your particular denominations ? : ";
    cout<<"\n 1.Yes \n 2. No ";
    cout<<"\n Enter your choice : ";
    cin>>opt;
    if(opt == 1)
    {
        int n;
        cout<<"\n How many types of denominations you want to enter ? : ";
        cin>>n;
        cout<<"\n Enter denominations : ";
        for(int i=0;i<n;i++)
        {
            cin>>user_deno[i];
        }
        size = n;
        cout<<"The minimum exchange calculated is : ";
        Min_Exchange(amount,size,user_deno);
    }
    else
    {
        size = sizeof(denominations) / sizeof(denominations[0]);
        cout<<"The minimum exchange calculated is : ";
        Min_Exchange(amount,size,denominations);
    }
    return 0;
}

```

## Output:

```
G:\OneDrive - Amity University\Desktop\ADA Lab>coin_exchange_greedy.exe

Enter the amount whose exchange is required : 289

Do you wish to enter your particular denominations ? :
1.Yes
2. No
Enter your choice : 2
The minimum exchange calculated is : 100 100 50 20 10 5 2 2
```

## Case 2:

```
#include<bits/stdc++.h>
using namespace std;
int Min_Exchange(int coins[], int m, int V)
{
    int table[V+1];
    table[0] = 0;

    for (int i=1; i<=V; i++)
    {
        table[i] = INT_MAX;
    }
    for (int i=1; i<=V; i++)
    {
        for (int j=0; j<m; j++)
        {
            if (coins[j] <= i)
            {
                int sub_res = table[i-coins[j]];
                if (sub_res != INT_MAX && sub_res + 1 < table[i])
                    table[i] = sub_res + 1;
            }
        }
    }
    if(table[V]==INT_MAX)
        return -1;
    return table[V];
}
int main()
{
    int amount;
    int opt;
    int size;
    int denominations[] = { 1, 2, 5, 10, 20,50, 100, 500, 1000 };
```

```

int user_deno[10];
cout<<"\n Enter the amount whose exchange is required : ";
cin>>amount;
cout<<"\n Do you wish to enter your particular denominations ? : ";
cout<<"\n 1.Yes \n 2. No ";
cout<<"\n Enter your choice : ";
cin>>opt;
if(opt == 1)
{
    int n;
    cout<<"\n How many types of denominations you want to enter ? : ";
    cin>>n;
    cout<<"\n Enter denominations : ";
    for(int i=0;i<n;i++)
    {
        cin>>user_deno[i];
    }
    size = n;
    cout<<"The      minimum      exchange      calculated      is      :
"<<Min_Exchange(user_deno , size , amount);
}
else
{
    size = sizeof(denominations) / sizeof(denominations[0]);
    cout<<"The      minimum      exchange      calculated      is      :
"<<Min_Exchange(denominations , size , amount);
}
return 0;
}

```

### Output:

```

Enter the amount whose exchange is required : 90

Do you wish to enter your particular denominations ? :
1.Yes
2. No
Enter your choice : 1

How many types of denominations you want to enter ? : 3

Enter denominations : 20 30 50
The minimum exchange calculated is : 3

```

### Case 3:

#### Using Greedy:

```

#include <bits/stdc++.h>
using namespace std;
void Min_Exchange(int V , int n , int deno[])

```

```

{
    sort(deno, deno + n);
    vector<int> ans;
    for (int i = n - 1; i >= 0; i--) {
        while (V >= deno[i]) {
            V -= deno[i];
            ans.push_back(deno[i]);
        }
    }
    for (int i = 0; i < ans.size(); i++)
        cout << ans[i] << " ";
}
int main()
{
    int amount;
    int opt;
    int size;
    int denominations[] = { 1, 2, 5, 10, 20, 50, 100, 500, 1000 };
    int user_deno[10];
    cout<<"\n Enter the amount whose exchange is required : ";
    cin>>amount;
    cout<<"\n Do you wish to enter your particular denominations ? : ";
    cout<<"\n 1.Yes \n 2. No ";
    cout<<"\n Enter your choice : ";
    cin>>opt;
    if(opt == 1)
    {
        int n;
        cout<<"\n How many types of denominations you want to enter ? : ";
        cin>>n;
        cout<<"\n Enter denominations : ";
        for(int i=0;i<n;i++)
        {
            cin>>user_deno[i];
        }
        size = n;
        cout<<"The minimum exchange calculated is : ";
        Min_Exchange(amount,size,user_deno);
    }
    else
    {
        size = sizeof(denominations) / sizeof(denominations[0]);
        cout<<"The minimum exchange calculated is : ";
        Min_Exchange(amount,size,denominations);
    }
    return 0;
}

```

### **Output:**

```
Enter the amount whose exchange is required : 80

Do you wish to enter your particular denominations ? :
1.Yes
2. No
Enter your choice : 1

How many types of denominations you want to enter ? : 3

Enter denominations : 10 40 60
The minimum exchange calculated is : 60 10 10
```

### **Using Dynamic Programming:**

```
#include<bits/stdc++.h>
using namespace std;
int Min_Exchange(int coins[], int m, int V)
{
    int table[V+1];
    table[0] = 0;

    for (int i=1; i<=V; i++)
    {
        table[i] = INT_MAX;
    }
    for (int i=1; i<=V; i++)
    {
        for (int j=0; j<m; j++)
        {
            if (coins[j] <= i)
            {
                int sub_res = table[i-coins[j]];
                if (sub_res != INT_MAX && sub_res + 1 < table[i])
                    table[i] = sub_res + 1;
            }
        }
    }
    if(table[V]==INT_MAX)
        return -1;
    return table[V];
}
int main()
{
    int amount;
    int opt;
    int size;
    int denominations[] = { 1, 2, 5, 10, 20,50, 100, 500, 1000 };
```



```

int user_deno[10];
cout<<"\n Enter the amount whose exchange is required : ";
cin>>amount;
cout<<"\n Do you wish to enter your particular denominations ? : ";
cout<<"\n 1.Yes \n 2. No ";
cout<<"\n Enter your choice : ";
cin>>opt;
if(opt == 1)
{
    int n;
    cout<<"\n How many types of denominations you want to enter ? : ";
    cin>>n;
    cout<<"\n Enter denominations : ";
    for(int i=0;i<n;i++)
    {
        cin>>user_deno[i];
    }
    size = n;
    cout<<"The minimum exchange calculated is :
"<<Min_Exchange(user_deno , size , amount);
}
else
{
    size = sizeof(denominations) / sizeof(denominations[0]);
    cout<<"The minimum exchange calculated is :
"<<Min_Exchange(denominations , size , amount);
}
return 0;
}

```

### **Output:**

```

G:\OneDrive - Amity University\Desktop\ADA Lab>coin_exchange_dp.exe

Enter the amount whose exchange is required : 80

Do you wish to enter your particular denominations ? :
1.Yes
2. No
Enter your choice : 1

How many types of denominations you want to enter ? : 3

Enter denominations : 10 40 60
The minimum exchange calculated is : 2

```

## **CASE STUDY**

### **Question**

**Question :** Given a string `s` and a dictionary of strings `wordDict` , return `True` if `s` can be segmented into a space-separated sequence of one or more dictionary words .  
Note that the same word in the dictionary may be reused multiple times in the segmentation.

### **Constraints :**

- Same word in the dictionary can be used multiple times.
- `S` and `wordDict[i]` consist of only lowercase English letters.
- All the strings of `wordDict` are unique.

### **Analysis Of the Problem :**

After analysing the above problem statement it can be said that it is a overlapping sub – problem statement . This problem can be solved using two approaches :

#### **1. Recursive Approach :**

- We will be iterating till prefix is found in dictionary.
- After this we would split the string into substring and use suffix in `wordbreak` function again
- If the prefix is present in dictionary, we recur for rest of the string (or suffix) in the `wordbreak` function again.

#### **2. Dynamic Programming :**

- Recursion causes overlapping of sub-structures. This will not happen in the dynamic programming approach
- When we are at a specific position `POS` and if the previous substring is present in the dictionary, then we need to check from `POS` to the end of the string
- If substring from `POS` to `END` is valid, then the entire string from `0` to `END` is valid.
- Hence, we need to store result of validity for subproblems from `POS` to `END`.

Hence after analysing it can be seen that DP Approach is the best

## **Code :**

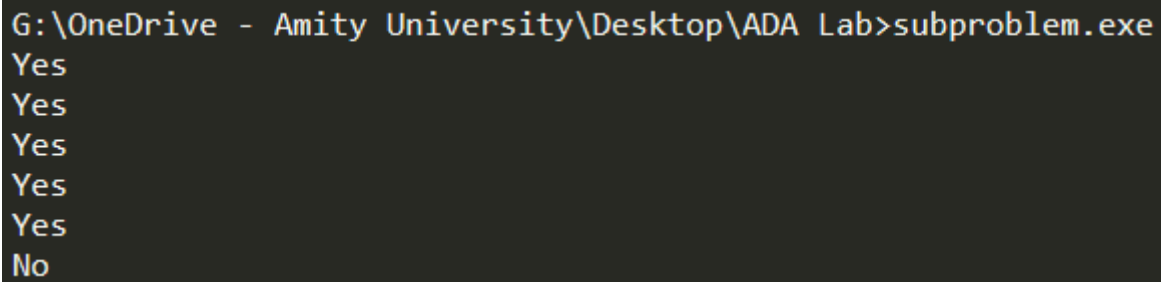
```
#include <iostream>
#include <string.h>
using namespace std;
int dictionaryContains(string word)
{
    string dictionary[] =
    {"mobile","samsung","sam","sung","man","mango","icecream","and","go","i","like","i
ce","cream"};
    int size = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
            return true;
    return false;
}
bool Break_word(string str)
{
    int size = str.size();
    if (size == 0) return true;
    bool wb[size+1];
    memset(wb, 0, sizeof(wb));

    for (int i=1; i<=size; i++)
    {
        if (wb[i] == false && dictionaryContains( str.substr(0, i) ))
            wb[i] = true;
        if (wb[i] == true)
        {
            if (i == size)
                return true;

            for (int j = i+1; j <= size; j++)
            {
                if (wb[j] == false && dictionaryContains( str.substr(i, j-i) ))
                    wb[j] = true;
                if (j == size && wb[j] == true)
                    return true;
            }
        }
    }
    return false;
}
int main()
{
    Break_word("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
    Break_word("iiiiiii")? cout <<"Yes\n": cout << "No\n";
}
```

```
Break_word("")? cout <<"Yes\n": cout <<"No\n";  
Break_word("ilikelikeimangoiii")? cout <<"Yes\n": cout <<"No\n";  
Break_word("samsungandmango")? cout <<"Yes\n": cout <<"No\n";  
Break_word("samsungandmangok")? cout <<"Yes\n": cout <<"No\n";  
return 0;  
}
```

## **Output :**



```
G:\OneDrive - Amity University\Desktop\ADA Lab>subproblem.exe  
Yes  
Yes  
Yes  
Yes  
Yes  
Yes  
No
```

## **Conclusion :**

After analysing both the approaches it was observed that Dynamic Programming Approach is the best and it gives us the optimal solution where it consumes less time i.e. It has less time complexity .

## **Time Complexity :**

- Time Complexity In Case of Recursion :  $O(2^n * s)$
- Time Complexity In Case of DP :  $O(n^3)$