

S.NO	Topic	Date	Signature
1.	Implement Linear Search and Binary Search and determine the number of steps taken to search for an element. Repeat for different value of n and plot a graph for comparison	20/07/22	
2.	To analyze and implement Bubble, Selection, and Insertion sort algorithm respectively.	03/08/22	
3.	To implement and analyze quick sort algorithm.	10/08/22	
4.	To implement and analyze merge sort algorithm	10/08/22	
5.	5a. To implement prims algorithm 5b. To implement Kruskal's algorithm	14/09/22	
6.	6A. To implement fractional knapsack 6B. To implement 0/1 knapsack using dynamic programming	28/09/22	
7.	7A. To implement breadth first search 7B. To implement depth first search	12/10/22	
8.	Travelling salesman problem using branch and bound	19/10/22	
9.	9A. To implement Dijkstra's algorithm 9B. To implement bellman ford algorithm	26/10/22	
10.	To implement n-queen problem	02/11/22	

### Open-ended

1.	To implement N- queen problem using backtracking approach		
2.	To implement 0/1 knapsack using backtracking method		

# Lab 1

**Aim:** Implement recursive linear and binary search and determine 56th time to search an element. Repeat for different values of n, number of elements in the list to be searched. Also plot the graph between time taken and n.

**Resources Used:** Ccompiler

## Theory:

### (A) Binary Search:

1. Binary search follows the 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.
2. It is used only for the sorted array.

### Linear Search:

1. Linear search is a search that finds an element in the list by searching the element sequentially until the element is found in the list.
2. It can be used for the array in any order.

## Algorithm:

### Linear search:

1. Step 1: set pos = -1
2. Step 2: set i = 1
3. Step 3: repeat step 4 while i <= n
4. Step 4: if a[i] == val
5. set pos = i
6. print pos
7. go to step 6
8. [end of if]
9. set ii = i + 1
10. [end of loop]
11. Step 5: if pos = -1
12. print "value is not present in the array "
13. [end of if]
14. Step 6: exit

### Binary search:

start

binarySearch(arr, x, low, high)

if low > high

return False

else

```

        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x > arr[mid]
            return binarySearch(arr, x, mid + 1, high)
        else
            return binarySearch(arr, x, low, mid - 1)
    end

```

### **Code1:**

```

#include <stdio.h>

int linearSearch(int a[], int n, int val) {

    for (int i = 0; i < n; i++)
    {
        if (a[i] == val)
            return i+1;
    }
    return -1;
}

int main() {
    int a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52};
    int val = 41;
    int n = sizeof(a) / sizeof(a[0]);
    int res = linearSearch(a, n, val);
    printf("The elements of the array are - ");
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\nElement to be searched is - %d", val);
    if (res == -1)
        printf("\nElement is not present in the array");
    else
        printf("\nElement is present at %d position of array", res);
    return 0;
}

```

### **Output:**

```
The elements of the array are - 70 40 30 11 57 41 25 14 52
Element to be searched is - 41
Element is present at 6 position of array

...Program finished with exit code 0
Press ENTER to exit console.□
```

## Code2:

```
#include <stdio.h>

int binarysearchrecur(int arr[],int num,int low,int high)
{
    if(low<=high)
    {
        int mid=(low+high)/2;
        if(num<arr[mid])
            binarysearchrecur(arr,num,low,mid-1);
        else if(num>arr[mid])
            binarysearchrecur(arr,num,mid+1,high);
        else
            return mid;
    }
    else
        return -1;
}

int main()
{
    int arr[100],num,len,i,low,high;
    printf("Enter the length of array: ");
    scanf("%d",&len);
    for(i=0;i<len;i++)
    {
        printf("Enter %d element: ",i);
        scanf("%d",&arr[i]);
    }
    printf("Enter the number to be searched: ");
```

```

scanf("%d",&num);
low=0;
high=len-1;
printf("Index of element %d is %d",num,binarysearchrecur(arr,num,low,high));
return 0;
}

```

### Output:

```

Enter the length of array: 5
Enter 0 element: 7
Enter 1 element: 9
Enter 2 element: 2
Enter 3 element: 4
Enter 4 element: 1
Enter the number to be searched: 4
Index of element 4 is 3

```

### Analysis:

Length of array at Iteration 1= n

Length of array at Iteration 2= n/2

Length of array at Iteration 3= (n/2)/2=n/2<sup>2</sup>

Length of array at Iteration k= n/2<sup>k</sup>

After k iterations length of array= 1

Length of array= n/2<sup>k</sup>=1

n=2<sup>k</sup>

Applying log on both sides

Log<sub>2</sub> n=log<sub>2</sub> 2<sup>k</sup>

Log<sub>2</sub> n=k log<sub>2</sub> 2

$$k = \log_2 n$$

Binary Search has complexity of order  $O(\log_2 n)$ .

### Analysis:

#### (A) Linear Search:

$$T(n) = 1/n + 2/n + 3/n + \dots + n/n.$$

$$T(n) = (1 + 2 + 3 + \dots + n) / n$$

$$T(n) = n(n+1)/2n \quad T(n) = n/2 + 1/2$$

Therefore, the time complexity is given by  $O(n/2) \sim O(n)$ .

#### Binary Search:

At iteration 1,  $T(n) = T(n/2) + c$  At iteration

$$2, T(n) = T(n/2^2) + 2c$$

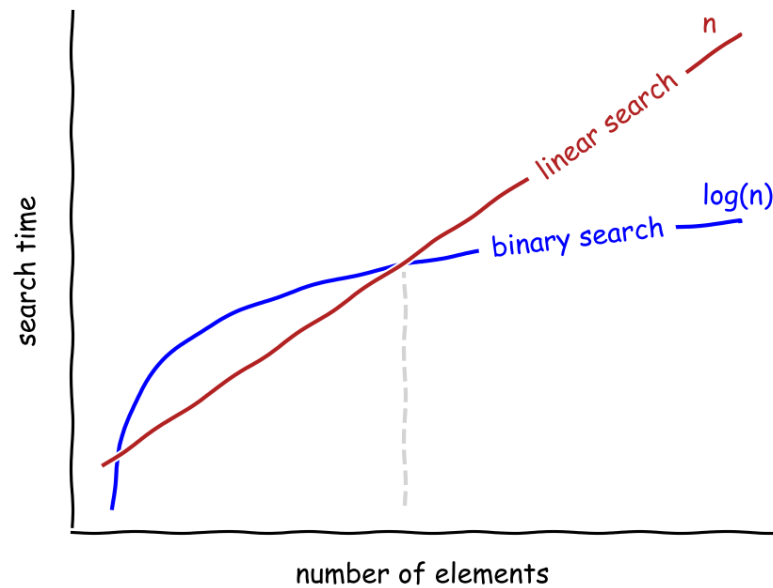
At iteration 3,  $T(n) = T(n/2^3) + 3c \dots$  At

iteration  $k$ ,  $T(n) = T(n/2^k) + kc$  Now since,

$$2^k = n \Rightarrow k = \log_2 n$$

Therefore, the time complexity is given by  $O(\log_2 n)$

### Graph:



## Lab 2

**Aim:** To implement insertion sort

Resource use: C++ compiler, c compiler

### Theory:

#### Selection Sort

- (1) The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.
- (2) The algorithm maintains two subarrays in a given array. The subarray which is already sorted. Remaining subarray which is unsorted.

#### Insertion Sort

- (3) Basically, Insertion sort is efficient for small data values.
- (4) Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

#### Bubble Sort

- (5) Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.
- (6) This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

### Algorithm:

#### Insertion sort:

```
Begin
for i := 1 to size-1 do
key:= array[i]
j := i
while j > 0 AND array[j-1] > key do
array[j]:= array[j-1];
j:= j - 1
done
array[j]:= key
done
End
```

#### Selection sort:

1. Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
2. Step 2: CALL SMALLEST(arr, i, n, pos)
3. Step 3: SWAP arr[i] with arr[pos]
4. [END OF LOOP]
5. Step 4: EXIT
6. SMALLEST (arr, i, n, pos)
7. Step 1: [INITIALIZE] SET SMALL = arr[i]
8. Step 2: [INITIALIZE] SET pos = i
9. Step 3: Repeat for j = i+1 to n
10. if (SMALL > arr[j])
11. SET SMALL = arr[j]
12. SET pos = j

13. [END OF if]
14. [END OF LOOP]
15. Step 4: RETURN pos

**Bubble sort:**

1. begin BubbleSort(arr)
2.   for all array elements
3.     if arr[i] > arr[i+1]
4.       swap(arr[i], arr[i+1])
5.     end if
6.   end for
7.   return arr
8. end BubbleSort

**Optimised bubble sort:**

1. bubbleSort(array)
2. n = length(array)
3. repeat
4.   swapped = false
5.   for i = 1 to n - 1
6.     if array[i - 1] > array[i], then
7.       swap(array[i - 1], array[i])
8.       swapped = true
9.     end if
10.   end for
11.   n = n - 1
12. until not swapped
13. end bubbleSort

**Code1:**

```
#include <iostream>
```

```
using namespace std;
```

```
void sort(int arr[], int a){
```



```

int i,j, key;
for(i=1;i<a;i++){
    key =arr[i];
    j=i-1;

    while (j>=0 && arr[j] > key){
        arr[j+1]=arr[j];
        j=j-1;
    }
    arr[j+1]=key;
}
}
int main()
{
    cout<<"enter the size of the array:";
    int a,b;
    cin>>a;
    int arr[a];
    for(int i=0;i<a;i++){
        cout<<"enter the array element:"<<" ";
        cin>>arr[i];
    }
    sort(arr,a);
    cout<<"your array is:";
    for(int i=0;i<a;i++){
        cout<<arr[i]<<" ";

    }

    return 0;
}

```

**Output:**

```
enter the size of the array:5
enter the array element: 9
enter the array element: 6
enter the array element: 3
enter the array element: 5
enter the array element: 2
your array is:2 3 5 6 9

...Program finished with exit code 0
Press ENTER to exit console.
```

## Code2:

```
#include <stdio.h>

void selection(int arr[], int n)
{
    int i, j, small;
    for (i = 0; i < n-1; i++)
    {
        small = i;
        for (j = i+1; j < n; j++)
            if (arr[j] < arr[small])
                small = j;
        int temp = arr[small];
        arr[small] = arr[i];
        arr[i] = temp;
    }
}

void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}

int main()
{
    int a[] = { 12, 31, 25, 8, 32, 17 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
```

```

    selection(a, n);

    printf("\nAfter sorting array elements are - \n");

    printArr(a, n);

    return 0;
}

```

### Output:

```

Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32

...Program finished with exit code 0
Press ENTER to exit console.

```

### Code3:

```

#include<stdio.h>

void print(int a[], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        printf("%d ",a[i]);
    }
}

void bubble(int a[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = i+1; j < n; j++)
        {
            if(a[j] < a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;    }}}}

```

```

void main ()
{
    int i, j, temp;
    int a[5] = { 10, 35, 32, 13, 26};
    int n = sizeof(a)/sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    print(a, n);
    bubble(a, n);
    printf("\nAfter sorting array elements are - \n");
    print(a, n);
}

```

Output:

```

Before sorting array elements are -
10 35 32 13 26
After sorting array elements are -
10 13 26 32 35

...Program finished with exit code 0
Press ENTER to exit console.

```

### Analysis:

#### Bubble Sort:

At pass 1 :

Number of comparisons = (n-1) Number of  
swaps = (n-1)

At pass 2 :

Number of comparisons = (n-2) Number of  
swaps = (n-2)

At pass 3 :

Number of comparisons = (n-3) Number of  
swaps = (n-3) . . .

At pass n-1 :

Number of comparisons = 1 Number of  
swaps = 1

Now , calculating total number of comparison required to sort the array = (n-1) + (n-2)  
+ (n-3) + . . . 2 + 1 = (n-1)\*(n-1+1)/2 { by using sum of N natural Number formula } = n (n-1)/2

Time Complexity=O(N<sup>2</sup>)

#### Insertion Sort:

The time complexity of insertion Sort is O(N<sup>2</sup>) as there are two nested loops: One loop to select an element of Array one by one = O(N) Another loop to compare that element with every other Array element = O(N) Therefore overall complexity = O(N)\*O(N) = O(N\*N) = O(N<sup>2</sup>)

#### Selection Sort:

The time complexity of Selection Sort is O(N<sup>2</sup>) as there are two nested loops: One loop to select an element of Array one by one = O(N) Another loop to compare that element with every other Array element = O(N) Therefore overall complexity = O(N)\*O(N) = O(N\*N) = O(N<sup>2</sup>)

# Lab 3

**Aim:** Implement and analyze the Quick Sort and merge sort algorithm.

Resource Used: C++ Compiler

**Theory:** Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as a pivot.
2. Always pick the last element as a pivot (implemented below)
3. Pick a random element as a pivot.
4. Pick median as the pivot.

## Partition Algorithm:

```
/* low -> Starting index, high -> Ending index */
quickSort(arr[], low, high) {
    if (low < high) {
        /* pi is partitioning index, arr[pi] is now at right place */
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

## Quick Sort Algorithm:

```
QUICKSORT (array A, start, end)
{
    1 if (start < end)
    2 {
    3 p = partition(A, start, end)
    4 QUICKSORT (A, start, p - 1)
    5 QUICKSORT (A, p + 1, end)
    6 }
}
```

## Source Code:

```
#include<iostream>
#include<algorithm>
```

```

#include <bits/stdc++.h>

using namespace std;

int part(int arr[],int l, int h){

    int pivot =arr[h];

    int i =l-1;

    for(int j=l;j<=h-1;j++){

        if(arr[j]<pivot){

            i++;

            swap(arr[i],arr[j]);

        }

    }

    swap (arr[i+1],arr[h]);

    return (i+1);

}

void quick_sort(int arr[], int l, int h){

    if(l<h){

        int p=part(arr,l,h);

        quick_sort(arr,l,p-1);

        quick_sort(arr,p+1,h);

    }

}

int main()

{

    cout<<"Enter the size of the array:";

    int a;

    cin>>a;

    int arr[a];

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

        cout<<"Enter the elements:";

        cin>>arr[i];

    }

    quick_sort(arr,0,a-1);

    cout<<"Your array is :";

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

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

    }

}

```

```
}  
    return 0;  
}
```

### Output:

```
Enter the size of the array:5  
Enter the elements:9  
Enter the elements:6  
Enter the elements:3  
Enter the elements:5  
Enter the elements:8  
Your array is :3 5 6 8 9  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

### Analysis:

The worst-case complexity of Quick-Sort algorithm is  $O(n^2)$ . However, using this technique, in average cases we get the output in  $O(n \log n)$  time.

# Lab 4

**Aim:** Implementation of Merge Sort

**Resources:** C compiler

**Algorithm:**

```
merge (a,p,q,r)
step 1: n1=q-p+1
step 2: n2= r-q
step 3: let l[1...n1+1] and r[1...n2+1] be new arrays
step 4: for i=1 to n1
step 5 : l[i] =a[p+i-1]
step 6: for j=1 to n2
step 7: r[j]=a[q+j]
step 8:l[n1+1] =infinity
step 9: r[n2+1] = infinity
step 10: i=1
step 11: j=1
step 12: for k=p to r
step 13: if l[i] <= r[j]
step 14: a[k]=l[i]
step 15: i=i+1
step 16 : else a[k]=r[j]
step 17 : j=j+1
```

**Complexity:**

**Time complexity analysis of merging algorithm :**

Memory allocation:  $O(1)$

Data copy process=  $O(n1)+O(n2)=O(n1+n2)=O(n)$

Merging loop in the worst case =  $O(n1+n2) = O(n)$

Boundary condition 1 in worst case =  $O(n1)$

Boundary condition 2 in the worst case =  $O(n2)$

Overall time complexity =  $O(1)+O(n)+O(n)+O(n1)+O(n2)=O(n)$

**Time complexity analysis of merge sort:**

**Code:**

```
#include <stdio.h>
#include <stdlib.h>
```

```
void merge(int a[], int left, int mid, int right)
```



```

{
    int i, j, k;
    int num1 = mid - left + 1;
    int num2 = right - mid;

    int L[num1], R[num2];

    for (i = 0; i < num1; i++)
        L[i] = a[left + i];
    for (j = 0; j < num2; j++)
        R[j] = a[mid + 1 + j];

    i = 0;    j = 0;
    k = left;
    while (i < num1 && j < num2) {
        if (L[i] <= R[j]) {
            a[k] = L[i];
            i++;
        }
        else {
            a[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < num1) {
        a[k] = L[i];
        i++;
        k++;
    }

    while (j < num2) {
        a[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int left, int right)
{
    if (left < right) {

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

```

```

        mergeSort(a, left, mid);
        mergeSort(a, mid + 1, right);

        merge(a, left, mid, right);
    }
}

```

```

void printArray(int A[], int n1)
{
    int i;
    for (i = 0; i < n1; i++)
        printf("%d ", A[i]);
    printf("\n");
}

```

```

int main()
{
    int a[] = { 39,9,81,45,90,27,72,18 };
    int n = sizeof(a) / sizeof(a[0]);

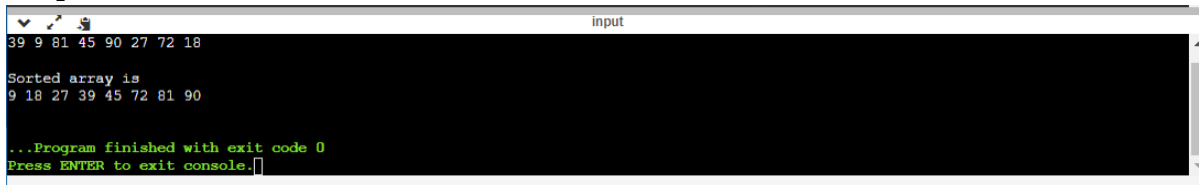
    printf("Given array is \n");
    printArray(a, n);

    mergeSort(a, 0, n - 1);

    printf("\nSorted array is \n");
    printArray(a, n);
    return 0;
}

```

### Output:



```

input
39 9 81 45 90 27 72 18

Sorted array is
9 18 27 39 45 72 81 90

...Program finished with exit code 0
Press ENTER to exit console.

```

# Lab-5a

**Aim:** To implement prims algorithm in c

**Resource used:** c compiler

**Theory:**

**Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

**Algorithm:**

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

**Source code:**

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#define infinity 9999
```

```
#define MAX 20
```

```
int G[MAX][MAX],spanning[MAX][MAX],n;
```

```
int prims();
```

```
int main()
```

```
{
```

```

int i,j,total_cost;

printf("Enter no. of vertices: ");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

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

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

scanf("%d",&G[i][j]);

total_cost=prims();

printf("\nspanning tree matrix:\n");

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

{

printf("\n");

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

printf("%d\t",spanning[i][j]);

}

printf("\n\nTotal cost of spanning tree=%d",total_cost);

return 0;

}

```

```

int prims()

{

int cost[MAX][MAX];

int u,v,min_distance,distance[MAX],from[MAX];

int visited[MAX],no_of_edges,i,min_cost,j;

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

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

```

```
{  
  
if(G[i][j]==0)  
  
cost[i][j]=infinity;  
  
else  
  
cost[i][j]=G[i][j];  
  
spanning[i][j]=0;  
  
}
```

```
  
distance[0]=0;  
  
visited[0]=1;  
  
for(i=1;i<n;i++)  
  
{  
  
distance[i]=cost[0][i];  
  
from[i]=0;  
  
visited[i]=0;  
  
}  
  
min_cost=0;  
  
no_of_edges=n-1;  
  
while(no_of_edges>0)  
  
{  
  
  
  
  
  
  
  
min_distance=infinity;  
  
for(i=1;i<n;i++)  
  
if(visited[i]==0&&distance[i]<min_distance)  
  
{  
  
v=i;  
  
min_distance=distance[i];
```

```

}

u=from[v];

spanning[u][v]=distance[v];
spanning[v][u]=distance[v];
no_of_edges--;
visited[v]=1;

for(i=1;i<n;i++)
if(visited[i]==0&&cost[i][v]<distance[i])
{
distance[i]=cost[i][v];
from[i]=v;
}

min_cost=min_cost+cost[u][v];
}

return(min_cost);
}

```

**Output:**

```
Enter no. of vertices: 4

Enter the adjacency matrix:
1 8 9 6
2 3 8 4
5 0 9 7
2 8 7 3

spanning tree matrix:

0      0      0      6
0      0      0      4
0      0      0      7
6      4      7      0

Total cost of spanning tree=21
```

### Analysis:

Prim's algorithm can be improved using Fibonacci Heaps (cf Cormen) to  $O(E + \log V)$

# Lab-5b

**Aim:** To implement kruskals algorithm using c

**Resource used:** c complier

**Theory:**

**Kruskal's Algorithm** is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

**Algorithm:**

1. Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
2. Step 2: Create a set E that contains all the edges of the graph.
3. Step 3: Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning
4. Step 4: Remove an edge from E with minimum weight
5. Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F
6. (for combining two trees into one tree).
7. ELSE
8. Discard the edge
9. Step 6: END

**Source code:**

```
#include<stdio.h>

#define MAX 30

typedef struct edge
{
    int u,v,w;
}edge;

typedef struct edgelist
{
    edge data[MAX];
    int n;
}edgelist;

edgelist elist;

int G[MAX][MAX],n;
```



```

edgelist spanlist;
void kruskal();
int find(int belongs[],int vertexno);
void union1(int belongs[],int c1,int c2);
void sort();
void print();
void main()
{
int i,j,total_cost;
printf("\nEnter number of vertices: ");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
kruskal();
print();
}
void kruskal()
{
int belongs[MAX],i,j,cno1,cno2;
elist.n=0;
for(i=1;i<n;i++)
for(j=0;j<i;j++)
{
if(G[i][j]!=0)
{
elist.data[elist.n].u=i;
elist.data[elist.n].v=j;
elist.data[elist.n].w=G[i][j];
elist.n++;
}
}
}

```

```

sort();
for(i=0;i<n;i++)
belongs[i]=i;
spanlist.n=0;
for(i=0;i<elist.n;i++)
{
cno1=find(belongs,elist.data[i].u);
cno2=find(belongs,elist.data[i].v);
if(cno1!=cno2)
{
spanlist.data[spanlist.n]=elist.data[i];
spanlist.n=spanlist.n+1;
union1(belongs,cno1,cno2);
}
}
}

int find(int belongs[],int vertexno)
{
return(belongs[vertexno]);
}

void union1(int belongs[],int c1,int c2)
{
int i;
for(i=0;i<n;i++)
if(belongs[i]==c2)
belongs[i]=c1;
}

void sort()
{
int i,j;
edge temp;
for(i=1;i<elist.n;i++)
for(j=0;j<elist.n-1;j++)

```

```

if(elist.data[j].w>elist.data[j+1].w)
{
temp=elist.data[j];
elist.data[j]=elist.data[j+1];
elist.data[j+1]=temp;
}
}

void print()
{
int i,cost=0;
for(i=0;i<spanlist.n;i++)
{
printf("\n%d\t%d\t%d",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);
cost=cost+spanlist.data[i].w;
}
printf("\n\nCost of the spanning tree=%d",cost);
}

```

Output:

```

Enter number of vertices: 4

Enter the adjacency matrix:
1 8 9 3
0 5 8 9
1 6 4 9
0 8 6 0

2      0      1
2      1      6
3      2      6

Cost of the spanning tree=13

```

### Analysis:

The time complexity of Kruskal's algorithm is  $O(E \log E)$  or  $O(V \log V)$ , where  $E$  is the no. of edges, and  $V$  is the no. of vertices.

# Lab 6

**Aim:** to implement fractional knapsack

**Resource used:** c++ compiler

## Theory:

The fractional knapsack problem is also one of the techniques which are used to solve the knapsack problem. In fractional knapsack, the items are broken in order to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.

## Algorithm:

STEP 1: CALCULATE THE RATIO(VALUE/WEIGHT) FOR EACH ITEM.

STEP 2: SORT ALL THE ITEMS IN DECREASING ORDER OF THE RATIO.

STEP 3: INITIALIZE RES =0, CURR\_CAP = GIVEN\_CAP.

STEP 4: DO THE FOLLOWING FOR EVERY ITEM “I” IN THE SORTED ORDER:

- IF THE WEIGHT OF THE CURRENT ITEM IS LESS THAN OR EQUAL TO THE REMAINING CAPACITY THEN ADD THE VALUE OF THAT ITEM INTO THE RESULT
- ELSE ADD THE CURRENT ITEM AS MUCH AS WE CAN AND BREAK OUT OF THE LOOP

SEP 5: RETURN RES

## Source code:

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
struct Item {
```

```
    int value, weight;
```

```
    Item(int value, int weight)
```

```
    {
```

```
        this->value = value;
```

```
        this->weight = weight;
```

```
    }
```

```
};
```

```

bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.value / (double)a.weight;
    double r2 = (double)b.value / (double)b.weight;
    return r1 > r2;
}

double fractionalKnapsack(int W, struct Item arr[], int N)
{
    sort(arr, arr + N, cmp);

    double finalvalue = 0.0;

    for (int i = 0; i < N; i++) {
        if (arr[i].weight <= W) {
            W -= arr[i].weight;
            finalvalue += arr[i].value;
        }

        else {
            finalvalue
            += arr[i].value
            * ((double)W / (double)arr[i].weight);
            break;
        }
    }

    return finalvalue;
}

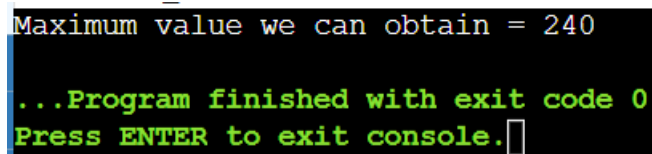
```

```
int main()
{
    int W = 50;
    Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };

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

    cout << "Maximum value we can obtain = "
        << fractionalKnapsack(W, arr, N);
    return 0;
}
```

Output:

A screenshot of a console window with a black background and green text. The text displays the output of the program, showing the maximum value and a completion message.

```
Maximum value we can obtain = 240
...Program finished with exit code 0
Press ENTER to exit console.
```

## Lab-6 B

**Aim:** To implement 0/1 knapsack problem using dynamic programming.

**Resource used:** c++ compiler

### Theory:

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack.

### Algorithm:

#### Dynamic-0-1-knapsack (v, w, n, W)

```
for w = 0 to W do
    c[0, w] = 0
for i = 1 to n do
    c[i, 0] = 0
    for w = 1 to W do
        if  $w_i \leq w$  then
            if  $v_i + c[i-1, w-w_i]$  then
                 $c[i, w] = v_i + c[i-1, w-w_i]$ 
            else  $c[i, w] = c[i-1, w]$ 
        else
```

#### Source code:

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

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

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

```
{
```

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

```
        return 0;
```

```
    if (wt[n - 1] > W)
```

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

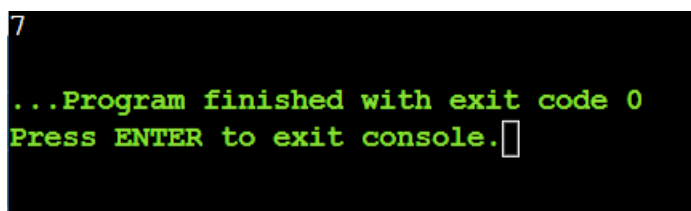
```

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

int main()
{
    int val[] = { 2,3,1,5 };
    int wt[] = { 3,4,6,5};
    int W = 8;
    int n = sizeof(val) / sizeof(val[0]);
    cout << knapSack(W, wt, val, n);
    return 0;
}

```

Output:



```

7
...Program finished with exit code 0
Press ENTER to exit console.

```



## Lab-7 A

**Aim:** To implement breadth firstsearch

**Resource used:** C++ compiler

### Theory:

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Algorithm:

**Step 1:** SET STATUS = 1 (ready state) for each node in G

**Step 2:** Enqueue the starting node A and set its STATUS = 2 (waiting state)

**Step 3:** Repeat Steps 4 and 5 until QUEUE is empty

**Step 4:** Dequeue a node N. Process it and set its STATUS = 3 (processed state).

**Step 5:** Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

(Waiting state)

[END OF LOOP]

Source code:

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
int cost[10][10],i,j,k,n,qu[10],front,rare,v,visit[10],visited[10];
int main()
{
```

```

int m;
cout <<"Enter no of vertices:";
cin >> n;
cout <<"Enter no of edges:";
cin >> m;
cout <<"\nEDGES \n";
for(k=1; k<=m; k++)
{
    cin >>i>>j;
    cost[i][j]=1;
}
cout <<"Enter initial vertex to traverse from: ";
cin >>v;
cout <<"Visited vertices: ";
cout <<v<<" ";
visited[v]=1;
k=1;
while(k<n)
{
    for(j=1; j<=n; j++)
        if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
        {
            visit[j]=1;
            qu[rare++]=j;
        }
    v=qu[front++];
    cout<<v <<" ";
    k++;
    visit[v]=0;
    visited[v]=1;
}

```

```
    }  
    return 0;  
}
```

Output:

```
Enter no of vertices:4  
Enter no of edges:6  
  
EDGES  
1 2  
2 3  
3 4  
4 1  
4 2  
3 1  
Enter initial vertex to traverse from:2  
Visited vertices:2 3 1 4
```

## Lab-7 B

**Aim:** To implement depth first search

**Resource used:** C++ compiler

### Theory:

The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm

Algorithm:

**Step 1:** SET STATUS = 1 (ready state) for each node in G

**Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)

**Step 3:** Repeat Steps 4 and 5 until STACK is empty

**Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)

**Step 5:** Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

**Step 6:** EXIT

**Source code:**

```
#include<iostream>

#include<conio.h>

#include<stdlib.h>

using namespace std;

int cost[10][10],i,j,k,n,stk[10],top,v,visit[10],visited[10];

int main()

{

    int m;
```

```

cout <<"Enter no of vertices:";

cin >> n;

cout <<"Enter no of edges:";

cin >> m;

cout <<"\nEDGES \n";

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

{

    cin >>i>>j;

    cost[i][j]=1;

}

cout <<"Enter initial vertex to traverse from: ";

cin >>v;

cout <<"DFS ORDER OF VISITED VERTICES: ";

cout << v <<" ";

visited[v]=1;

k=1;

while(k<n)

{

    for(j=n; j>=1; j--)

        if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)

        {

            visit[j]=1;

            stk[top]=j;

            top++;

        }

}

```

```

        v=stk[--top];

        cout<<v << " ";

        k++;

        visit[v]=0;

        visited[v]=1;

    }

    return 0;

}

```

Output:

```

Enter no of vertices:4
Enter no of edges:6

EDGES
1 2
2 3
3 4
4 1
4 2
3 1
Enter initial vertex to traverse from: 1
DFS ORDER OF VISITED VERTICES: 1 2 3 4

```

## Lab-8

**Aim:** To implement travelling salesman problem using branch and bound

Resource used: C++

### compilerTheory:

Travelling Salesman Problem is based on a real-life scenario, where a salesman from a company has to start from his own city and visit all the assigned cities exactly once and return to his home till the end of the day.

### Source code:

```
#include <bits/stdc++.h>

using namespace std;

const int N = 4;

int final_path[N+1];

bool visited[N];

int final_res = INT_MAX;

void copyToFinal(int curr_path[])
{
    for (int i=0; i<N; i++)
        final_path[i] = curr_path[i];
    final_path[N] = curr_path[0];
}

int firstMin(int adj[N][N], int i)
{
    int min = INT_MAX;
    for (int k=0; k<N; k++)
        if (adj[i][k]<min && i != k)
            min = adj[i][k];
    return min;
}

int secondMin(int adj[N][N], int i)
{
    int first = INT_MAX, second = INT_MAX;
```

```

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

    if (adj[i][j] <= first)
    {
        second = first;
        first = adj[i][j];
    }
    else if (adj[i][j] <= second &&
        adj[i][j] != first)
        second = adj[i][j];
}
return second;
}

void TSPRec(int adj[N][N], int curr_bound, int curr_weight,
    int level, int curr_path[])
{
    if (level==N)
    {

        if (adj[curr_path[level-1]][curr_path[0]] != 0)
        {

            int curr_res = curr_weight +
                adj[curr_path[level-1]][curr_path[0]];

            if (curr_res < final_res)

```



```

    {
        copyToFinal(curr_path);
        final_res = curr_res;
    }
}

return;
}

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

    if (adj[curr_path[level-1]][i] != 0 &&
        visited[i] == false)
    {
        int temp = curr_bound;
        curr_weight += adj[curr_path[level-1]][i];

        if (level==1)
            curr_bound -= ((firstMin(adj, curr_path[level-1]) +
                            firstMin(adj, i))/2);
        else
            curr_bound -= ((secondMin(adj, curr_path[level-1]) +
                            firstMin(adj, i))/2);

        if (curr_bound + curr_weight < final_res)
        {
            curr_path[level] = i;
            visited[i] = true;

            TSPRec(adj, curr_bound, curr_weight, level+1,
                    curr_path);

```

```

    }

    curr_weight -= adj[curr_path[level-1]][i];
    curr_bound = temp;

    memset(visited, false, sizeof(visited));
    for (int j=0; j<=level-1; j++)
        visited[curr_path[j]] = true;
    }
}

void TSP(int adj[N][N])
{
    int curr_path[N+1];

    int curr_bound = 0;
    memset(curr_path, -1, sizeof(curr_path));
    memset(visited, 0, sizeof(curr_path));

    for (int i=0; i<N; i++)
        curr_bound += (firstMin(adj, i) +
                      secondMin(adj, i));

    curr_bound = (curr_bound&1)? curr_bound/2 + 1 :
                curr_bound/2;

    visited[0] = true;
    curr_path[0] = 0;

    TSPRec(adj, curr_bound, 0, 1, curr_path);
}

```

```

}

int main()
{
    int adj[N][N] = { {0, 10, 15, 20},
                      {10, 0, 35, 25},
                      {15, 35, 0, 30},
                      {20, 25, 30, 0}
    };

    TSP(adj);

    printf("Minimum cost : %d\n", final_res);
    printf("Path Taken : ");
    for (int i=0; i<=N; i++)
        printf("%d ", final_path[i]);

    return 0;
}

```

Output:

```

Minimum cost : 80
Path Taken : 0 1 3 2 0

...Program finished with exit code 0
Press ENTER to exit console.

```

## Lab-9 A

**Aim:** To implement Dijkstra algorithm

**Resource used:** C++ compiler

### Theory:

Dijkstra algorithm is a single-source shortest path algorithm. Here, single source means that only one source is given, and we have to find the shortest path from the source to all the nodes.

### Source code:

```
#include<iostream>
```

```
#include<limits>
```

```
using namespace std;
```

```
int minimumDist(int dist[], bool Tset[])
```

```
{
```

```
    int min=INT_MAX,index;
```

```
    for(int i=0;i<6;i++)
```

```
    {
```

```
        if(Tset[i]==false && dist[i]<=min)
```

```
        {
```

```
            min=dist[i];
```

```
            index=i;
```

```
        }
```

```
    }
```

```
    return index;
```

```
}
```

```
void Dijkstra(int graph[6][6],int src) {
```

```
    int dist[6];
```

```
    bool Tset[6];
```

```

for(int i = 0; i<6; i++)
{
    dist[i] = INT_MAX;
    Tset[i] = false;
}

dist[src] = 0;

for(int i = 0; i<6; i++)
{
    int m=minimumDist(dist,Tset);
    Tset[m]=true;
    for(int i = 0; i<6; i++)
    {
        if(!Tset[i] && graph[m][i] && dist[m]!=INT_MAX &&
dist[m]+graph[m][i]<dist[i])
            dist[i]=dist[m]+graph[m][i];
    }
}

cout<<"Vertex\t\tDistance from source"<<endl;
for(int i = 0; i<6; i++)
{
    char str=65+i;
    cout<<str<<"\t\t"<<dist[i]<<endl;
}
}

int main()
{

```

```
int graph[6][6]={
    {0, 10, 20, 0, 0, 0},
    {10, 0, 0, 50, 10, 0},
    {20, 0, 0, 20, 33, 0},
    {0, 50, 20, 0, 20, 2},
    {0, 10, 33, 20, 0, 1},
    {0, 0, 0, 2, 1, 0}};

Dijkstra(graph,0);

return 0;

}
```

Output:

Vertex	Distance from source
A	0
B	10
C	20
D	23
E	20
F	21

## Lab-9B

**Aim:** To implement the bellman ford algorithm.

Resource used: C++

compilerTheory:

Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph. There are various other algorithms used to find the shortest path like Dijkstra algorithm, etc.

**Source code:**

```
#include<iostream>

#define MAX 10

using namespace std;

typedef struct edge
{
    int src;
    int dest;
    int wt;
}edge;

void bellman_ford(int nv,edge e[],int src_graph,int ne)
{
    int u,v,weight,i,j=0;
    int dis[MAX];

    for(i=0;i<nv;i++)
    {
        dis[i]=999;
    }

    dis[src_graph]=0;
```

```

for(i=0;i<nv-1;i++)
{
    for(j=0;j<ne;j++)
    {
        u=e[j].src;
        v=e[j].dest;
        weight=e[j].wt;

        if(dis[u]!=999 && dis[u]+weight < dis[v])
        {
            dis[v]=dis[u]+weight;
        }
    }
}

for(j=0;j<ne;j++)
{
    u=e[j].src;
    v=e[j].dest;
    weight=e[j].wt;

    if(dis[u]+weight < dis[v])
    {
        cout<<"\n\nNEGATIVE CYCLE PRESENT..!!\n";
        return;
    }
}

```



```

cout<<"\nVertex"<<" Distance from source";
for(i=1;i<=nv;i++)
{
    cout<<"\n"<<i<<"\t"<<dis[i];
}
}
int main()
{
    int nv,ne,src_graph;
    edge e[MAX];

    cout<<"Enter the number of vertices: ";
    cin>>nv;

    printf("Enter the source vertex of the graph: ");
    cin>>src_graph;

    cout<<"\nEnter no. of edges: ";
    cin>>ne;

    for(int i=0;i<ne;i++)
    {
        cout<<"\nFor edge "<<i+1<<"=">";
        cout<<"\nEnter source vertex :";
        cin>>e[i].src;
        cout<<"Enter destination vertex :";
        cin>>e[i].dest;
        cout<<"Enter weight :";
        cin>>e[i].wt;
    }
}

```

```
}
```

```
bellman_ford(nv,e,src_graph,ne);
```

```
return 0;
```

```
}
```

Output:

```
Enter the source vertex of the graph: 1
```

```
Enter no. of edges: 6
```

```
For edge 1=>
```

```
Enter source vertex :1
```

```
Enter destination vertex :2
```

```
Enter weight :-5
```

```
For edge 2=>
```

```
Enter source vertex :2
```

```
Enter destination vertex :3
```

```
Enter weight :6
```

```
For edge 3=>
```

```
Enter source vertex :3
```

```
Enter destination vertex :4
```

```
Enter weight :5
```

```
For edge 4=>
```

```
Enter source vertex :4
```

```
Enter destination vertex :1
```

```
Enter weight :-2
```

```
For edge 5=>
```

```
Enter source vertex :4
```

```
Enter destination vertex :2
```

```
Enter weight :4
```

```
For edge 6=>
```

```
Enter source vertex :3
```

```
Enter destination vertex :1
```

```
Enter weight :-1
```

```
Vertex Distance from source
```

```
1 -3
```

```
2 -8
```

```
3 -2
```

```
4 -1
```

## Lab-10

**Aim:** To implement n queen problem

**Resource used:** C++ compiler

**Theory:**

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

**Source code:**

```
#include<iostream>
```

```
using namespace
```

```
std;#define N 4
```

```
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++)
```

```
        if (board[row][i])
```

```
            return false;
```

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

```
        if (board[i][j])
```

```
            return false;
```

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

```
        if (board[i][j])
```

```
            return false;
```

```
    return true;
```

```
}
```

```

bool solveNQueen(int board[N][N], int col) {
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++) {
        if (isValid(board, i, col) ) {
            board[i][col] = 1;
            if ( solveNQueen(board, col + 1))

                board[i][col] = 0;
        }
    }
    return false;
}

```

```

bool checkSolution() {
    int board[N][N];
    for(int i = 0; i<N; i++)
        for(int j = 0; j<N; j++)
            board[i][j] = 0;

    if ( solveNQueen(board, 0) == false ) {
        return false;
    }
    printBoard(board);
    return true;
}

```

```

int main() {
    checkSolution();
}

```

```
}
```

Output:

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

...Program finished with exit code 0
Press ENTER to exit console.
```

## Open – ended experiment

**Aim:** To implement n- queen problem using back-tracking method

**Resource used:** C++ compiler

### Theory:

The N-queen is the problem of placing N chess queens on N X N chessboard so that no two queens attack each other.

### Backtracking:

#### Algorithm:

1. Start in the leftmost column
2. If all the queens are placedReturn true
3. Try all rows I the current column.  
Do following for every tired row,
  - a) If the queen can be placed safely in this row then mark this [row, column] as partof 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.
4. If all rows have been tried and nothing worked, return false to trigger backtracking

### Source code:

```
#define N 4

#include <stdbool.h>

#include <stdio.h>

void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}

bool isSafe(int board[N][N], int row, int col)
```

```

{
    int i, j;
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {
        //if it is safe to place the queen at position i,col -> place it
        if (isSafe(board, i, col)) {

            board[i][col] = 1;

            if (solveNQUtil(board, col + 1))
                return true;
        }
    }
}

```

```

//backtrack if the above condition is false
    board[i][col] = 0;
}
}
return false;
}
int main()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return 0;
    }

    printSolution(board);
    return true;
    return 0;
}

```

Output:

```

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

```



**Aim:** To implement 0/1 knapsack using back-tracking approach

**Resource used:** C compiler

## Theory:

In the 0-1 Knapsack problem, we are given a set of items with individual weights and profits and a container of fixed capacity (the knapsack) and are required to compute a loading of the knapsack with items such that the total profit is maximised. The 0-1 knapsack problem is NP-hard but can be solved quite efficiently using backtracking. Below is a backtracking implementation in C. The function knapsack () takes arrays of weights, and profits, their size, the capacity, and the address of a pointer through which the solution array is returned

## Source code:

```
#include <stdio.h>

#define max 10

int w[max],i,j,p[max];

int n,m;

float unit[max];

int y[max],x[max],fp=-1,fw;

void get()

{

printf("\n Enter total number of items: ");

scanf("%d",&n);

printf("\n Enter the Maximum capacity of the Sack: ");
```

```
scanf("%d",&m);
```

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

```
{
```

```
printf("\n Enter the weight of the item # %d : ",i+1);
```

```
scanf("%d",&w[i]);
```

```
printf("\n Enter the profit of the item # %d : ", i+1);
```

```
scanf("%d", &p[i]);
```

```
}
```

```
}
```

```
void show()
```

```
{
```

```
float s=0.0;
```

```
printf("\n\tItem\tWeight\tCost\tUnit Profit\tSelected ");
```

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

```
printf("\n \t%d \t%d \t%d \t%f \t%d",i+1,w[i],p[i],unit[i],x[i]);
```

```
printf("\n\n The Sack now holds following items : ");
```

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

```
if(x[i]==1)
```

```
{
```

```
printf("%d\t",i+1);
```

```
s += (float) p[i] * (float) x[i];
```

```
}
```

```
printf("\n Maximum Profit: %f ",s);
```

```
}
```

```
/*Arrange the item based on high profit per Unit*/
```

```
void sort()
```

```
{
```

```
int t,t1;
```

```
float t2;
```

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

```
unit[i] = (float) p[i] / (float) w[i];
```

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

```
{
```

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

```
{
```

```
if(unit[i] < unit[j])
```

```
{
```

```
t2 = unit[i];
```

```
unit[i] = unit[j];
```

```
unit[j] = t2;
```

```
t = p[i];
```

```
p[i] = p[j];
```

```
p[j] = t;
```

```
t1 = w[i];
```

```
w[i] = w[j];
```

```
w[j] = t1;
```

```
}
```

```
}
```

```
}
```

```
}
```

```
float bound(float cp,float cw,int k)
```

```
{
```

```
float b = cp;
```

```
float c = cw;
```

```
for(i=k;i<=n;i++)
```

```
{
```

```
c = c+w[i];
```

```
if( c < m)
```

```
b = b +p[i];
```

```
else
```

```
return (b+(1-(c-m)/ (float)w[i])*p[i]);
```

```
}
```

```
return b;
```

```
}
```

```
void knapsack(int k,float cp,float cw)
```

```
{
```

```
if(cw+w[k] <= m)
```

```
{
```

```
y[k] = 1;
```

```
if(k <= n)
```

```
knapsack(k+1,cp+p[k],cw+w[k]);
```

```
if(((cp+p[k]) > fp) && ( k == n))
```

```
{
```

```
fp = cp+p[k];
```

```
fw = cw+w[k];
```

```
for(j=0;j<=k;j++)
```

```
x[j] = y[j];
```

```
}
```

```
}
```

```
if(bound(cp,cw,k) >= fp)
```

```
{
```

```
y[k] = 0;
```

```
if( k <= n)
```

```
knapsack(k+1,cp,cw);
```

```
if((cp > fp) && (k == n))
```

```
{
```

```
fp = cp;
```

```
fw = cw;
```

```
for(j=0;j<=k;j++)
```

```
x[j] = y[j];
```

```
}
```

```
}
```

```
}
```

```

void main()
{
printf("\n\n\n\t ***** KNAPSACK PROBLEM *****");
printf("\n\t\t_____");
get();
printf("\n The Sack is arranged in the order...\n");
sort();
knapsack(0,0.0,0.0);
show();
}

```

Output:

```

***** KNAPSACK PROBLEM *****
_____
Enter total number of items: 3
Enter the Maximum capacity of the Sack: 7
Enter the weight of the item # 1 : 5
Enter the profit of the item # 1 : 6
Enter the weight of the item # 2 : 4
Enter the profit of the item # 2 : 5
Enter the weight of the item # 3 : 3
Enter the profit of the item # 3 : 4
The Sack is arranged in the order...

    Item    Weight    Cost    Unit Profit    Selected
    1         3         4     1.333333         1
    2         4         5     1.250000         1
    3         5         6     1.200000         0

The Sack now holds following items : 1 2
Maximum Profit: 9.000000 |

```