

HKE SOCIETY'S
POOJYA DODDAPPA APPA COLLEGE
OF ENGINEERING, GULBARGA



INFORMATION SCIENCE AND
ENGINEERING
LAB MANUAL
DESIGN AND ANALYSIS OF ALGORITHMS
(11IS48)
2014-2015

PREPARED BY: Ms. Suma S.

Mrs. Geeta V J.

CONTENTS

SLNO	TITLE	PAGE NO.
1	Implement Recursive Binary search and Linear search and determine the time required to search an element	4
2	Sort a given set of elements using Heapsort method and determine the time required to sort the element	9
3	Sort a given set of elements using Merge sort method and determine the time required to sort the elements.	13
4	Sort a given set of elements using Selection sort and determine the time required to sort elements	18
5	Implement 0/1 Knapsack problem using dynamic programming	21
6	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.	24
7	Sort a given set of elements using Quick sort method and determine the time required to sort the elements	28
8	Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm	32
9	a) Print all the nodes reachable from a given starting node in a digraph using BFS method. b) Check whether a given graph is connected or not using DFS method	35
10	Find a subset of a given set $S = \{s_1, s_2, \dots, s_n\}$ of n Positive integers whose sum is equal to a given positive integer d . For example, if $S = \{1, 2, 5, 6, 8\}$ and $d=9$ there are two solutions $\{1, 2, 6\}$ and $\{1, 8\}$. A suitable message is to be displayed if the given problem instance doesn't have a solution.	42
11	a. Implement Horspool algorithm for String Matching b. Find the Binomial Co-efficient using Dynamic Programming	45
12	Find Minimum Cost Spanning Tree of a given Undirected graph using Prim's algorithm.	52

13	a. Implement Floyd's algorithm for the All-Pairs- Shortest- Paths Problem. b. Compute the transitive closure of a given directed graph using Wars hall's algorithm	55
14	Implement N Queen's problem using Back Tracking.	61
15	Algorithm lab viva questions	65

LAB-1

Implement Recursive Binary search and linear search and determine the time required to search an element.

BINARY SEARCH METHOD

Binary search is a remarkable efficient algorithm for searching in a sorted array. It works by comparing a search key k with

the array's middle element $A[m]$, if they match, the algorithm stops; otherwise the same operation is repeated recursively for the first half of the arrays if $k < A[m]$ and for the second half $k > A[m]$

$A[0].....A[m-1] \quad A[m] \quad A[m+1].....A[n-1]$

Search here if $k < A[m]$

search here if $k > A[m]$

Algorithm: Binary search($A[0...n-1], k$)
 //implement non recursive binary search
 //input :an array $A[0...n-1]$ sorted in ascending order and a search key k
 //output:an index of the array's element that is equal to k or -1 if there is no such element
 $l \leftarrow 0; r \leftarrow n-1$
 while $l \leq r$ do
 $m \leftarrow (l+r)/2$
 if $k = A[m]$ return m
 else if $k < A[m]$ $r \leftarrow m-1$
 else $l \leftarrow m+1$
 return -1

Complexity: Best case :total number of comparisons required is one. $T(N) = \Omega(1)$

worst case :the worst case time complexity of binary search $T(n) = O(\log_2 n)$

average case :if search is successful then $T(n) = \log_2 n - 1$.if search is unsuccessful then $T(n) = \log_2 n + 1$

LINEAR SEARCH METHOD

linear search or sequential search is a method for finding a particular value in a list that checks each element in sequence until the desired element is found or the list is exhausted. This is a straightforward algorithm that searches for a given item (some search key K) in a list

of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted. Here is the algorithm's pseudocode, in which, for simplicity, a list is implemented as an array. It also assumes that the second condition $A[i] \neq K$ will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.

ALGORITHM *Linear Search*($A[0..n-1], K$)
 //Searches for a given value in a given array by sequential search
 //Input: An array $A[0..n-1]$ and a search key K
 //Output: The index of the first element in A that matches K
 // or -1 if there are no matching elements
 $i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
 $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Complexity

Best case: In best case algorithms make only one comparison. The number of comparisons=1
 $T(n)=1$

Worst case : In worst case the number of comparisons required is n . $T(n)=n$

Average case :if search is successful then $T(n)=(n+1)/2$ if search is unsuccessful then $T(n)=n$

PROGRAM

```
#include<stdio.h>

#include<time.h>

#define MAX 20

int a[MAX],n,key;

main()
{
    int i,key,ch,mid,low,high,l;

    clock_t start1,start2,end1,end2;

    clrscr();
```

```
printf("enter the limit\n");
scanf("%d",&n);
printf("enter the element \n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
printf("binary search\n");
printf("enter the key to b searchd\n");
scanf("%d",&key);
low=1;
high=n;
start1=clock();
ch=bs(low,high,key);
if(ch== -1)
printf("element not found \n");
else
printf("element found\n");
end1=clock();
printf("\n linear search\n");
printf("enter the element to b searchd\n");
scanf("%d",&key);
start2=clock();
l=ls(0,key);
if(l== -1)
printf("element is not found\n");
else
printf("element is found \n");
end2=clock();
printf("\n TIME FOR BINARY SEARCH=%f\n",(end1-start1)/CLK_TCK);
```

```
printf("\n TIME FOR LINEAR SEARCH=%f\n",(end2-start2)/CLK_TCK);
getch();
}

int bs(int low,int high,int key)
{
    int mid;
    long int j;
    for(j=0;j<1000000;j++)
    if(low>high)
    return -1;
    mid=(low+high)/2;
    if(a[mid]==key)
    return mid;
    else
    {
        if(key<a[mid])
        bs(low,mid-1,key);
        else
        bs(mid+1,high,key);
    }
}

int ls(int i,int key)
{
    long int j;
    for(j=0;j<10000000;j++)
    if(i>n)
    return -1;
    if(a[i]==key)
```

```

return i;

else

ls(++i,key);

}

```

OUTPUT:

```

enter the limit
5
enter the element
1 2 3 4 5
binary search
enter the key to b searchd
3
element found

linear search
enter the element to b searchd
5
element is found

TIME FOR BINARY SEARCH=0.054945
TIME FOR LINEAR SEARCH=0.274725

```

Binary search

Size n	Ascending order		Descending order	
	input	time taken	input	time taken
4	1 2 3 4	0.00000	4 3 2 1	0.00000
8	1 2 3 4 5 6 7 8	0.00000	8 7 6 5 4 3 2 1	0.00000
16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	0.00000	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	0.00000

Linear search

Size n	Ascending order		Descending order	
	input	time taken	input	time taken
4	1 2 3 4	0.109890	4 3 2 1	0.219780
8	1 2 3 4 5 6 7 8	0.219780	8 7 6 5 4 3 2 1	0.219780
16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	0.6543	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	0.439560

LAB-2

Sort a given set of elements using Heapsort method and determine the time required to sort the element.

HEAPSORT METHOD

A heap is defined as a binary tree with keys assigned to its nodes provided following two condition are met

1. The tree's shape requirement –the binary tree is essentially complete ,that is ,all its level are full except possibly the last level ,where only some rightmost leaves may be missing
2. The parental dominance requirement-the key at each node is greater than or equal to the key at its children.

```
Algorithm: heap bottom up(H[1...n])
//constructs a heap from the elements of a array by bottom up algorithm
//input:An array H[1..n] of oderable items
//output:A heap H[1...n]
For I <-n/2 downto 1 do
  K <-I; v <-H[k]
  Heap <- false
  While not heap and 2*k<=n do
    J <-2*k
    If j<n // there are two children
      If H[j] < H[j+1] j <-j+1
      If v >=H[j]
        Heap <- true
      Else H[k] <- H[j]; k <-j
    H[k] <-v
```

Complexity : The heap construction stage of the algorithm is in $O(n)$. for the $T(n)$ number of comparisons $O(n \log n)$.

PROGRAM

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

```
#include<conio.h>
```

```
#include<time.h>

int a[30];

main()
{
    int n,i;
    clock_t start,end,a[30];
    clrscr();
    start=clock();
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    printf("Enter the elements\n");
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
    heap(n);
    heapsort(n);
    end=clock();
    printf("Sorted array\n");
    for(i=1;i<=n;i++)
        printf("%d\n",a[i]);
    printf("\n\ntime=%f\n",(end-start)/CLK_TCK);
    getch();
}

heap(n)
{
    int ch,ps,temp;
    for(ch=1;ch<=n;ch++)
    {
        temp=a[ch];
```

```
ps=ch/2;
while(ch>1&&temp>a[ps])
{
a[ch]=a[ps];
ch=ps;
ps=ch/2;
if(ps<1)
ps=1;
}
a[ch]=temp;
}
}

heapsort(int n)
{
while(n>1)
{
swap(&a[1],&a[n]);
{
n--;
heap(n);
}
}
}

swap(int *x,int *y)
{
```

```

int temp;

temp=*x;

*x=*y;

*y=temp;

}
  
```

OUTPUT:

```

enter the size
5
enter the elements
25 64 89 10 12
the sorted array is:

10
12
25
64
89
time=15.494505_
  
```

HEAP SORT

Size n	Ascending order		Descending order		Random order	
	input	time taken	input taken	time	input taken	time
4	1 2 3 4	8.571429	4 3 2 1	9.670330	2 3 1 4	18.461538
8	1 2 3 4 5 6 7 8	19.890110	8 7 6 5 4 3 2 1	13.571429	5 4 2 3 1 8 6 7	21.263736
16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	33.956044	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	28.678495	5 8 6 4 3 2 1 7 9 12 13 11 16 15 14 10	43.076923

LAB-3

Sort a given set of elements using Merge sort method and determine the time required to sort the elements.

MERGE SORT ALGORITHM METHOD:

Merge sort is a perfect example of a successful application of the divide-and-conquer technique.

1. Split array A[1..n] in two and make copies of each half in arrays B[1.. n/2] and C[1.. n/2]
2. Sort arrays B and C
3. Merge sorted arrays B and C into array A as follows:
 - a) Repeat the following until no elements remain in one of the arrays:
 - i. compare the first elements in the remaining unprocessed portions of the arrays
 - ii. copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
 - b) Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Algorithm: MergeSort (A [0...n-1])
 //This algorithm sorts array A [0...n-1] by recursive mergesort.
 //Input: An array A [0...n-1] of orderable elements.
 //Output: Array A [0...n-1] sorted in non-decreasing order

```

{
if n>1
{ Copy A [0... ⌊n/2⌋ -1] to B [0... ⌊n/2⌋ -1]
Copy A [ ⌊n/2⌋ ... n-1] to C [0... ⌊n/2⌋ -1]
MergeSort (B [0... ⌊n/2⌋ -1])
MergeSort (C [0... ⌊n/2⌋ -1])
Merge (B, C, A)
}
}
  
```

Algorithm: Merge (B [0...p-1], C [0...q-1], A [0...p+q-1])
 //Merges two sorted arrays into one sorted array.
 //Input: Arrays B [0...p-1] and C [0...q-1] both sorted.
 //Output: Sorted array A [0...p+q-1] of the elements of B and C.

```

{
i ← 0; j ← 0; k ← 0
while i < p and j < q do
{
if B[i] <= C[j]
A[k] ← B[i]; i ← i+1
else
A[k] ← C[j]; j ← j+1
k ← k+1
}
if i = p
copy C [j...q-1] to A[k...p+q-1]
else
copy B [i...p-1] to A[k...p+q-1]
}

```

Complexity:

- ☐ All cases have same efficiency: $\Theta(n \log n)$
- ☐ Number of comparisons is close to theoretical minimum for comparison-based sorting:
 $\log n! \approx n \lg n - 1.44 n$
- ☐ Space requirement: $\Theta(n)$ (NOT in-place)

The time complexity of merge sort is $T(N) = O(N \log_2 N)$

PROGRAM

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
int a[25],b[25];
main()
{
int i,x;
clock_t start,end;
clrscr();

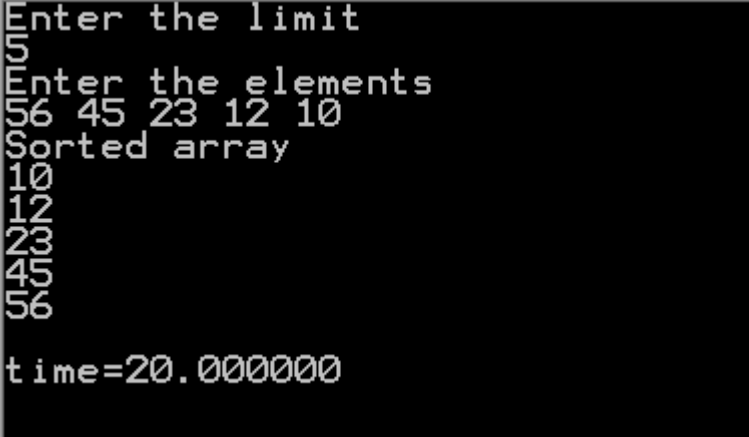
```

```
start=clock();
printf("Enter the limit\n");
scanf("%d",&x);
printf("Enter the elements\n");
for(i=1;i<=x;i++)
scanf("%d",&a[i]);
mergesort(1,x);
end=clock();
printf("Sorted array\n");
for(i=1;i<=x;i++)
printf("%d\n",a[i]);
printf("\ntime=%f",(end-start)/CLK_TCK);
getch();
}

mergesort(int low,int high)
{
int mid;
if(low<high)
{
mid=(low+high)/2;
mergesort(low,mid);
mergesort(mid+1,high);
merge(low,mid,high);
}
}

merge(int low,int mid,int high)
{
int i,j,k,h;
h=i=low;
```

```
j=mid+1;
while(h<=mid&& j<=high)
{
if(a[h]<a[j])
b[i++]=a[h++];
else
b[i++]=a[j++];
}
if(h>mid)
{
for(k=j;k<=high;k++)
b[i++]=a[k];
}
else
{
for(k=h;k<=mid;k++)
b[i++]=a[k];
}
for(k=low;k<=high;k++)
a[k]=b[k];
}
```

OUTPUT:

```
Enter the limit
5
Enter the elements
56 45 23 12 10
Sorted array
10
12
23
45
56

time=20.000000
```


MERGE SORT

Size n	Ascending order		Descending order		Random order	
	input	time taken	input taken	time	input taken	time
4	1 2 3 4	7.802198	4 3 2 1	5.659341	2 3 1 4	8.791209
8	1 2 3 4 5 6 7 8	12.582418	8 7 6 5 4 3 2 1	8.351648	5 4 2 3 1 8 6 7	9.340659
16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	27.967033	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	19.670330	5 8 6 4 3 2 1 7 9 12 13 11 16 15 14 10	31.703297

LAB -4

Sort a given set of elements using Selection sort and determine the time required to sort elements.

SELECTION SORT METHOD

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list then we scan the list, starting with the second element, to find the smallest among the last $n-1$ element and exchanges it with the second element, putting the smallest element in its final position. generally, on the i th pass through the list, which we number from 0 to $n-2$, the algorithm searches for the smallest item among the last $n-i$ element and swaps it with a_i

$A[0] \leq A[1] \leq \dots \leq A[i-1] \mid A[i], \dots, A[\min], \dots, A[n-1]$
 In their final position the last $n-i$ elements

Algorithm: selection sort ($A[0 \dots n-1]$)
 //the algorithm sorts a given array by selection sort
 //input: An array $A[0 \dots n-1]$ of orderable elements
 //output: Array $A[0 \dots n-1]$ sorted in ascending order
 For $i \leftarrow 0$ to $n-2$ do
 $\text{Min} \leftarrow i$
 For $j \leftarrow i+1$ to $n-1$ do
 If $A[j] < A[\text{min}]$ $\text{min} \leftarrow j$
 Swap $A[i]$ and $A[\text{min}]$

Complexity: The time complexity of selection sort is $T(n) = \theta(n^2)$ the selection sort behaves the same for worst and best cases.

PROGRAM

```
#include<stdio.h>

#include<conio.h>

#include<time.h>
```

```
int a[25];

main()
{
    int i,n;
    clock_t start,end;
    clrscr();
    start=clock();
    printf("\nEnter the number of elements\n");
    scanf("%d",&n);
    printf("Enter the elements\n");
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
    selsort(n);
    printf("\nSorted elements are\n");
    for(i=1;i<=n;i++)
        printf("%d\n",a[i]);
    end=clock();
    printf("\ntime=%f", (end-start)/CLK_TCK);
    getch();
}

selsort(int x)
{
    int i,j;
    for(i=1;i<=x;i++)
    {
        for(j=i+1;j<=x;j++)
            if(a[i]>a[j])
                swap(&a[i],&a[j]);
    }
}
```

```

}

swap(int *c,int *d)

{
int temp;

temp=*c;

*c=*d;

*d=temp;

}

```

OUTPUT:

```

Enter the number of elements
5
Enter the elements
21 12 25 10 89

Sorted elements are
10
12
21
25
89

time=22.087912_

```

SELECTION SORT

Size n	Ascending order		Descending order		Random order	
	input	time taken	input taken	time	input taken	time
4	1 2 3 4	8.681319	4 3 2 1	8.791209	2 3 1 4	13.021978
8	1 2 3 4 5 6 7 8	8.901012	8 7 6 5 4 3 2 1	9.890110	5 4 2 3 1 8 6 7	17.417582
16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	19.670330	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	26.549437	5 8 6 4 3 2 1 7 9 12 13 11 16 15 14 10	25.549451

LAB-5

Implement 0/1 Knapsack problem using dynamic programming

KNAPSACK PROBLEM METHOD

Given: A set S of n items, with each item i having

□ b_i - a positive benefit

□ w_i - a positive weight

Goal: Choose items with maximum total benefit but with weight at most W . i.e.

Objective: maximize $\sum_{i \in T} b_i$

Constraint: $\sum_{i \in T} w_i \leq W$

$F(i, j) = \text{Max}\{F(i-1, j), b_i + F(i-1, j-w_i)\}$ if $j-w_i \geq 0$,

$F(i-1, j)$ if $j-w_i < 0$.

It is convenient to define the initial conditions as follows:

$F(0, j) = 0$ for $j \geq 0$ and $F(i, 0) = 0$ for $i \geq 0$.

Algorithm: 0/1Knapsack(S, W)

//Input: set S of items with benefit b_i and weight w_i ; max. weight W

//Output: benefit of best subset with weight at most W

//Sk: Set of items numbered 1 to k .

//Define $B[k, w]$ = best selection from S_k with weight exactly equal to w

```
{
for w ← 0 to n-1 do
B[w] ← 0
for k ← 1 to n do
{
for w ← W downto wk do
{
if B[w-wk]+bk > B[w] then
B[w] ← B[w-wk]+bk
}}}
```

Complexity: The Time efficiency and Space efficiency of 0/1 Knapsack algorithm is $\Theta(nW)$.

PROGRAM

```
#include<stdio.h>

#include<conio.h>

int i,n,cap,maxprofit,w[25],p[25];

main()

{

clrscr();

printf("Enter the number of objects\n");

scanf("%d",&n);

printf("Enter the weights\n");

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

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

printf("Enter the profit\n");

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

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

printf("Enter the capacity\n");

scanf("%d",&cap);

maxprofit=sack(0,cap);

printf("Maximum profit is %d",maxprofit);

getch();

}

max(int x,int y)

{

return(x>y?x:y);

}

sack(int i,int y)

{

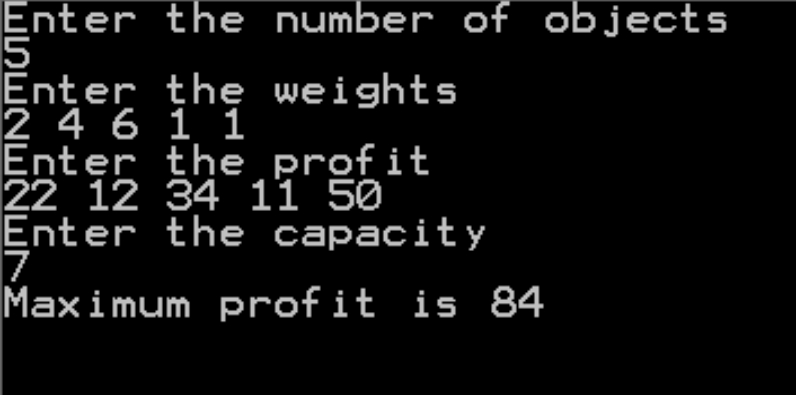
if(i==n)

if(y<w[n])
```

```
return 0;

else
return p[n];

if(y<w[i])
return sack(i+1,y);
return max(sack(i+1,y),sack(i+1,y-w[i])+p[i]);
}
```

OUTPUT:

```
Enter the number of objects
5
Enter the weights
2 4 6 1 1
Enter the profit
22 12 34 11 50
Enter the capacity
7
Maximum profit is 84
```

LAB-6

From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

SINGLE SOURCE SHORTEST PATHS PROBLEM:

For a given vertex called the source in a weighted connected graph, find the shortest paths to all its other vertices. Dijkstra's algorithm is the best known algorithm for the single source shortest paths problem. This algorithm is applicable to graphs with nonnegative weights only and finds the shortest paths to a graph's vertices in order of their distance from a given source. It finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. It is applicable to both undirected and directed graphs.

Algorithm : Dijkstra(G,s)

```
//Dijkstra's algorithm for single-source shortest paths
//Input :A weighted connected graph  $G=(V,E)$  with nonnegative weights and its vertex s
//Output : The length  $d_v$  of a shortest path from s to v and its penultimate vertex  $p_v$  for
//every v in V.
{
  Initialise(Q) // Initialise vertex priority queue to empty
  for every vertex v in V do
  {
     $d_v \leftarrow \infty$ ;  $p_v \leftarrow \text{null}$ 
    Insert(Q,v, $d_v$ ) //Initialise vertex priority queue in the priority queue
  }
   $d_s \leftarrow 0$ ; Decrease(Q,s  $d_s$ ) //Update priority of s with  $d_s$ 
   $V_t \leftarrow \emptyset$ 
  for i  $\leftarrow 0$  to  $|V|-1$  do
  {
     $u^* \leftarrow \text{DeleteMin}(Q)$  //delete the minimum priority element
     $V_t \leftarrow V_t \cup \{u^*\}$ 
    for every vertex u in  $V - V_t$  that is adjacent to  $u^*$  do
    {
      if  $d_{u^*} + w(u^*,u) < d_u$ 
      {
         $d_u \leftarrow d_{u^*} + w(u^*, u)$ ;  $p_u \leftarrow u^*$ 
        Decrease(Q,u, $d_u$ )
      }
    }
  }
}
```


Complexity: The Time efficiency for graphs represented by their weight matrix and the priority queue implemented as an unordered array and for graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is $O(|E| \log |V|)$.

PROGRAM

```
#include<stdio.h>

#include<conio.h>

main()
{
int n,i,j,cost[10][10],dist[10],v;

clrscr();

printf("enter the number of nodes\n");

scanf("%d",&n);

printf("enter the cost of matrix\n");

for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);

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

cost[i][j]=999;

}

printf("enter the source vertex\n");

scanf("%d",&v);

dij(n,v,cost,dist);

printf("shortest path from\n");

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

if(j!=v)

printf("%d->%d_%d\n",v,j,dist[j]);

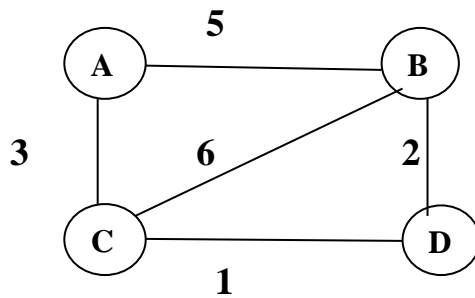
getch();
```

```
}  
  
dij(int n,int v,int cost[10][10],int dist[])  
{  
    int i,u,c,w,min,f[10];  
    for(i=1;i<=n;i++)  
    {  
        f[i]=0;  
        dist[i]=cost[v][i];  
    }  
    f[v]=1;  
    dist[v]=1;  
    c=2;  
    while(c<=n)  
    {  
        min=999;  
        for(w=1;w<=n;w++)  
            if(dist[w]<min&&!f[w])  
            {  
                min=dist[w];  
                u=w;  
            }  
        f[u]=1;  
        c++;  
        for(w=1;w<=n;w++)  
            if((dist[u]+cost[u][w]<dist[w])&&!f[w])  
                dist[w]=dist[u]+cost[u][w];  
    }  
}
```

OUTPUT:

```
enter the number of nodes
4
enter the cost of matrix
999 5 3 999
5 999 6 2
3 6 999 1
999 2 1 999
enter the source vertex
4
shortest path from
4->1_4
4->2_2
4->3_1
-
```

Input Graph

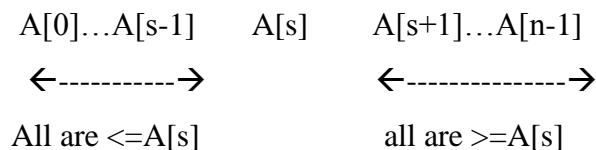


LAB-7

Sort a given set of elements using Quick sort method and determine the time required to sort the elements.

QUICK SORT METHOD

Quick Sort divides the array according to the value of elements. It rearranges elements of a given array $A[0..n-1]$ to achieve its partition, where the elements before position s are smaller than or equal to $A[s]$ and all the elements after position s are greater than or equal to $A[s]$.



Algorithm QUICKSORT(a[l..r])

//Sorts a subarray by quicksort

//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices l and r

//Output: Subarray $A[l..r]$ sorted in nondecreasing order

```
{
if l < r
{
s ← Partition(A[l..r]) //s is a split position
    QUICKSORT(A[l..s-1])
    QUICKSORT(A[s+1..r])
}}
```

Algorithm : Partition(A[l..r])

//Partition a subarray by using its first element as its pivot

//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices l and r ($l < r$)

//Output: A partition of $A[l..r]$, with the split position returned as this function's value

```
{
    p ← A[l]
    i ← l; j ← r+1
    repeat
    { repeat i ← i+1 until A[i] ≥ p
      repeat j ← j-1 until A[j] ≤ p
      swap(A[i], A[j])
    } until i ≥ j
    swap(A[i], A[j]) // undo last swap when i ≥ j
    swap(A[l], A[j])
    return j
}
```

Complexity: $C_{\text{best}}(n) = 2 C_{\text{best}}(n/2) + n$ for $n > 1$ $C_{\text{best}}(1) = 0$

$C_{\text{worst}}(n) \in \Theta(n^2)$

$C_{\text{avg}}(n) \approx 1.38n \log_2 n$

PROGRAM

```
#include<stdio.h>

#include<conio.h>

#include<time.h>

qsort(int a[20],int low,int high);

main()

{

int i,n,a[20],clock_t,start,end;

clrscr();

start=clock();

printf("Enter the number of elements\n");

scanf("%d",&n);

printf("Enter the elements\n");

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

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

qsort(a,0,n-1);

end=clock();

printf("Sorted list\n");

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

printf("%d\n",a[i]);

printf("time=%f",(end-start)/CLK_TCK);

getch();

}

qsort(int a[20],int low,int high)
```

```
{
int i,j,temp,key;
if(low<high)
{
key=low;
i=low;
j=high;
while(i<j)
{
while(a[i]<=a[key]&& i<high)
i++;
while(a[j]>a[key])
j--;
if(i<j)
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
temp=a[key];
a[key]=a[j];
a[j]=temp;
qsort(a,low,j-1);
qsort(a,j+1,high);
}
}
```

OUTPUT:

```

Enter the number of elements
5
Enter the elements
25
65
89
45
10
Sorted list
10
25
45
65
89
time=11.043956

```

QUICK SORT

Size n	Ascending order		Descending order		Random order	
	input	time taken	input taken	time	input taken	time
4	1 2 3 4	7.802198	4 3 2 1	5.659341	2 3 1 4	8.791209
8	1 2 3 4 5 6 7 8	12.582418	8 7 6 5 4 3 2 1	8.351648	5 4 2 3 1 8 6 7	9.340659
16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	27.967033	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	19.670330	5 8 6 4 3 2 1 7 9 12 13 11 16 15 14 10	31.703297

LAB-8

Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

KRUSKAL'S ALGORITHM:

Kruskal's algorithm finds the minimum spanning tree for a weighted connected graph $G=(V,E)$ to get an acyclic subgraph with $|V|-1$ edges for which the sum of edge weights is the smallest. Consequently the algorithm constructs the minimum spanning tree as an expanding sequence of subgraphs, which are always acyclic but are not necessarily connected on the intermediate stages of algorithm. The algorithm begins by sorting the graph's edges in non decreasing order of their weights. Then starting with the empty subgraph, it scans the sorted list adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

Algorithm : Kruskal(G)

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G = (V, E)$ 
//Output:  $ET$ , the set of edges composing a minimum spanning tree of  $G$ 
sort  $E$  in nondecreasing order of the edge weights  $w(e_1) \leq \dots \leq w(e_{|E|})$ 
 $ET \leftarrow \text{NULL}$ ;  $ecounter \leftarrow 0$  //initialize the set of tree edges and its size
 $k \leftarrow 0$  //initialize the number of processed edges
while  $ecounter < |V| - 1$  do
 $k \leftarrow k + 1$ 
if  $ET \cup \{e_k\}$  is acyclic
 $ET \leftarrow ET \cup \{e_k\}$ ;  $ecounter \leftarrow ecounter + 1$ 
return  $ET$ 
```

Complexity: With an efficient sorting algorithm, the time efficiency of kruskal's algorithm will be in $O(|E| \log |E|)$.

PROGRAM

```
#include<stdio.h>

#include<conio.h>

int min,n_0_ed=1;

int mincost=0,parent[10],cost[10][10];
```



```
int a,b,i,j,u,v,n;

main()

{

clrscr();

printf("\n enter the no of vertices\n");

scanf("%d",&n);

printf("enter cost matrix\n");

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

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

{

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

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

cost[i][j]=999;

}

while(n_0_ed<n)

{

for(i=1,min=999;i<n;i++)

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

if(cost[i][j]<min)

{

min=cost[i][j];

a=u=i;

b=v=j;

}

while(parent[u])

u=parent[u];

while(parent[v])

v=parent[v];

if(u!=v)
```

```

{
n_0_ed++;

printf("\n %d \t Edge \t(%d->%d)=%d",n_0_ed,a,b,min);

mincost+=min;

parent[v]=u;

}

cost[a][b]=cost[b][a]=999;

printf("\n \t mincost=%d",mincost);

}

getch();

}

```

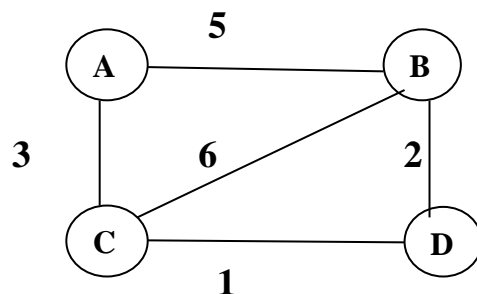
OUTPUT:

```

enter the no of vertices
4
enter cost matrix
999 5 3 999
5 999 6 2
3 6 999 1
999 2 1 999

2      Edge    <3->4>=1
      mincost=1
3      Edge    <2->4>=2
      mincost=3
4      Edge    <1->3>=3
      mincost=6_

```

Input Graph

LAB-9

- a) Print all the nodes reachable from a given starting node in a digraph using BFS method.
- b) Check whether a given graph is connected or not using DFS method

A) BFS METHOD

Breadth First Search: BFS explores graph moving across to all the neighbors of last visited vertex traversals i.e., it proceeds in a concentric manner by visiting all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it and so on, until all the vertices in the same connected component as the starting vertex are visited. Instead of a stack, BFS uses queue.

Algorithm : BFS(G)

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = (V, E)
//Output: Graph G with its vertices marked with consecutive integers in the order they
//have been visited by the BFS traversal
{
  mark each vertex with 0 as a mark of being "unvisited"
  count ← 0
  for each vertex v in V do
  {
    if v is marked with 0
    bfs(v)
  }}

```

Algorithm : bfs(v)

```
//visits all the unvisited vertices connected to vertex v and assigns them the numbers
//in order they are visited via global variable count
{
  count ← count + 1
  mark v with count and initialize queue with v
  while queue is not empty do
  {
    a := front of queue
    for each vertex w adjacent to a do

```

```

{
if  $w$  is marked with 0
{
count  $\leftarrow$  count + 1
mark  $w$  with count
add  $w$  to the end of the queue
}}
remove  $a$  from the front of the queue
}}
```

Complexity: BFS has the same efficiency as DFS: it is $\Theta(V^2)$ for Adjacency matrix representation and $\Theta(V+E)$ for Adjacency linked list representation.

PROGRAM

```

#include<stdio.h>

int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1,t;

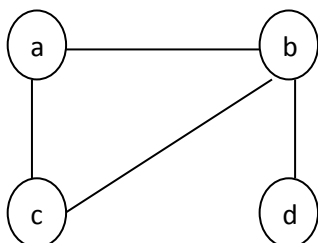
void bfs(int v)
{
for(i=1;i<=n;i++)
if(a[v][i]&&!visited[i])
q[++r]=i;
if(f<=r)
visited[q[f]]=1,bfs(q[f++]);
}

void main()
{
int v;
clrscr();
printf("enter the no of vertices\n");
scanf("%d",&n);
for(i=1;i<=n;i++)
q[i]=0,visited[i]=0;
```

```
printf("enter the adjacency matrix\n");  
for(i=1;i<=n;i++)  
for(j=1;j<=n;j++)  
scanf("%d",&a[i][j]);  
printf("enter the starting vertex\n");  
scanf("%d",&v);  
bfs(v);  
printf("the nodes which are reachable\n");  
for(i=1;i<=n;i++)  
if(visited[i])  
printf("%d\n",i);  
getch();  
}
```

OUTPUT:

```
enter the no of vertices  
4  
enter the adjacency matrix  
0 1 1 0  
0 0 1 1  
0 0 0 0  
0 0 0 0  
enter the starting vertex  
1  
the nodes which are reachable  
2  
3  
4
```



B) DFS METHOD**Depth First Search:**

Depth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. This process continues until a vertex with no adjacent unvisited vertices is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end.

Algorithm : DFS(G)

```
//Implements a depth-first search traversal of a given graph
//Input : Graph G = (V,E)
//Output : Graph G with its vertices marked with consecutive integers in the order they
//have been first encountered by the DFS traversal
{
mark each vertex in V with 0 as a mark of being “unvisited”.
count ← 0
for each vertex v in V do
if v is marked with 0
dfs(v)
}
```

Algorithm : dfs(v)

```
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered via global variable count
{
count ← count+1
mark v with count
for each vertex w in V adjacent to v do
if w is marked with 0
dfs(w)
}
```

Complexity: For the adjacency matrix representation, the traversal time efficiency is in $\Theta(|V|^2)$ and for the adjacency linked list representation, it is in $\Theta(|V|+|E|)$, where $|V|$ and $|E|$ are the number of graph's vertices and edges respectively.

PROGRAM

```
#include<stdio.h>

int a[20][20],vis[20],n,i,s,l[10],lab=0,k,j;

void init()

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

vis[i]=0;

l[i]=0;

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

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

a[i][j]=0;

}

void getdata()

{

printf("enter the adjacency matrix\n");

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

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

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

}

void dfs(int v)

{

vis[v]=1;

l[v]=lab;

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

if(a[v][i]&&vis[i]==0)\

dfs(i);

}
```

```
void clab()
{
    int i;
    for(i=1;i<=n;i++)
        if(l[i]==0)
        {
            lab++;
            for(k=1;k<=n;k++)
                vis[k]=0;
            dfs(i);
        }
    if(lab>1)
        printf("disconnected graph\n");
    else
        printf("connected graph\n");
    printf("\n No of components %d\n",lab);
    for(i=1;i<=n;i++)
        printf("\n vertex %d belongs to component=%d",i,l[i]);
}

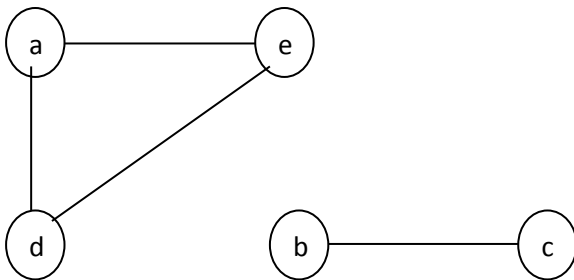
void main()
{
    clrscr();
    printf("enter the no of vertices\n");
    scanf("%d",&n);
    init();
    getdata();
    clab();
    getch();
}
```


OUTPUT:

```
enter the no of vertices
5
enter the adjacency matrix
0 0 0 1 1
0 0 1 0 0
0 1 0 0 0
1 0 0 0 1
1 0 0 1 0
disconnected graph

No of components 2

vertex 1 belongs to component=1
vertex 2 belongs to component=2
vertex 3 belongs to component=2
vertex 4 belongs to component=1
vertex 5 belongs to component=1_
```

Input graph

LAB-10

Find a subset of a given set $S = \{s_1, s_2, \dots, s_n\}$ of n Positive integers whose sum is equal to a given positive integer d . For example, if $S = \{1, 2, 5, 6, 8\}$ and $d=9$ there are two solutions $\{1, 2, 6\}$ and $\{1, 8\}$. A suitable message is to be displayed if the given problem instance doesn't have a solution.

SUM OF SUBSETS METHOD

Subset-Sum Problem is to find a subset of a given set $S = \{s_1, s_2, \dots, s_n\}$ of n positive integers whose sum is equal to a given positive integer d . It is assumed that the set's elements are sorted in increasing order. The state-space tree can then be constructed as a binary tree and applying backtracking algorithm, the solutions could be obtained. Some instances of the problem may have no solutions

Algorithm SumOfSub(s, k, r)

//Find all subsets of $w[1 \dots n]$ that sum to m . The values of $x[j]$, $1 \leq j < k$, have already been determined. $s = \sum_{j=1}^{k-1} w[j] * x[j]$ and $r = \sum_{j=1}^n w[j]$. The $w[j]$'s are in ascending order.

$j=1$ $j=k$

{

$x[k] \leftarrow 1$ //generate left child

if $(s + w[k] = m)$

write $(x[1 \dots n])$ //subset found

else if $(s + w[k] + w[k+1] \leq m)$

SumOfSub($s + w[k], k+1, r - w[k]$)

//Generate right child

if $(s + r - w[k] \geq m)$ and $(s + w[k+1] \leq m)$)

{

$x[k] \leftarrow 0$

SumOfSub($s, k+1, r - w[k]$)

}

}

Complexity: Subset sum problem solved using backtracking generates at each step maximal two new subtrees, and the running time of the bounding functions is linear, so the running time is $O(2^n)$.

PROGRAM

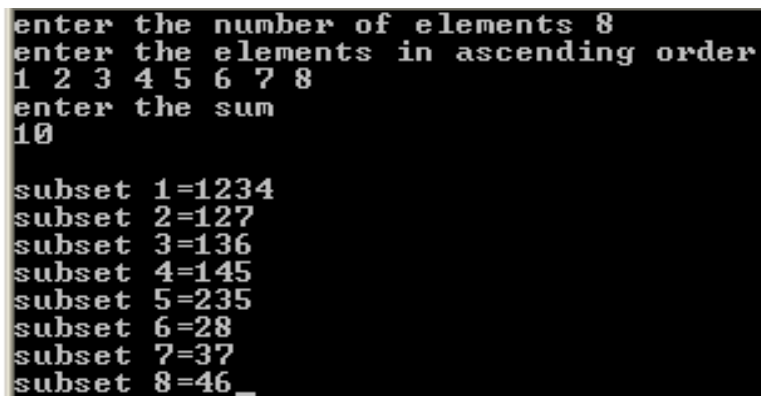
```
#include<stdio.h>

int w[10],i,d,n,c,x[10];

void so (int s,int k,int r)
{
    x[k]=1;
    if(s+w[k]==d)
    {
        printf("\nsubset %d=",++c);
        for(i=0;i<=k;i++)
            if(x[i])
                printf("%d",w[i]);
    }
    else
        if(s+w[k]+w[k+1]<=d)
            so(s+w[k],k+1,r-w[k]);
        if((s+r-w[k]>=d)&&(s+w[k+1]<=d))
        {
            x[k]=0;
            so(s,k+1,r-w[k]);
        }
}

void main()
{
    int su=0,k;
    clrscr();
    printf("enter the number of elements");
```

```
scanf("%d",&n);  
printf("enter the elements in ascending order\n");  
for(i=0;i<n;i++)  
scanf("%d",&w[i]);  
printf("enter the sum\n");  
scanf("%d",&d);  
for(i=0;i<=n;i++)  
x[i]=0;  
for(i=0;i<n;i++)  
su=su+w[i];  
if(su<d || w[0]>d)  
printf("the number of subset possible are\n");  
else  
so(0,0,su);  
getch();  
}
```

OUTPUT:

```
enter the number of elements 8  
enter the elements in ascending order  
1 2 3 4 5 6 7 8  
enter the sum  
10  
  
subset 1=1234  
subset 2=127  
subset 3=136  
subset 4=145  
subset 5=235  
subset 6=28  
subset 7=37  
subset 8=46_
```

LAB-11

- a. Implement Horspool algorithm for String Matching.
- b. Find the Binomial Co-efficient using Dynamic Programming.

A) HORSPPOOL ALGORITHM

Starting with the last of the pattern and moving right to left ,we compare the corresponding pairs of character in the pattern and the text.if all the pattern's character match successfully ,a matching is found if ,however we encounter a mismatch,we need to shift the pattern to the right . horspool algorithm determine the size of such a shift by looking at the character c of the text that was aligned against the last character of the pattern .in general ,the following for possibilities can occur

Case1: if there are no c's in the pattern we can safely shift the pattern by its entire length

Case 2: if there are occurrence of character c in the pattern but it is not the last one there ,the shift should again the rightmost occurrence of c in the the pattern with the c in the text

Case3:if c happens to be the last character in the pattern but there are no c's among its other m-1 character ,the shift should be similar to that of case1

Case4:finally if c happens to be last character in the pattern and there are c's among its first m-1 character ,the shift should be similar to that of case 2

Algorithm: shift table(p[0..m-1])

//fills the shift table used by horspool's algorithm

//input: pattern p[0...m-1] and an alphabet of possible character

//output: table [0...size-1] indexed by the alphabets character and filled with shift sizes computed by formula

Initialize all the element of table with m

For j <- 0 to m-2 do table [p[j]] <- m-1-j

Return table

Algorithm: horspool matching(p[0...m-1],t[0....n-1])

//implement horspool's algorithm for string matching

//input1 :pattern p[0...m-1] and text t[0....n-1]

//output: the index of the left end of the first matching substring or -1 if there are no matches

Shift table(p[0..m-1])// generate table of shifts

I <- m-1 //position of the pattern's right end

While i<=n-1 do

```

K <- 0           //number of matched character
While k<=m-1 and p[m-1-k]=t[i-k]
K <-k+1
If k=m
Return I <- m+ 1
Else I <-i+table[t[i]]
Return -1

```

Complexity: The worst case efficiency of horspool's algorithm is still same as brute force method i.e $O(nm)$.but for random texts, it is $O(n)$ on average.

PROGRAM

```

#include<stdio.h>

#include<string.h>

#define max 256

char p[max],t[max],ta[max];

int m,n,i,j,k;

void shifttable()
{
int in;
for(i=0;i<max;i++)
ta[i]=m;
for(j=0;j<m-1;j++)
{
in=(int)p[j]-'0';
ta[in]=m-1-j;
}
}

int horsepool_algo()
{
int index;

```

```
shifttable();

i=m-1;

for(;i<=n-1;)

{

k=0;

while((k<=m-1)&&(p[m-1-k]==t[i-k]))

k++;

if(k==m)

return i-m+1;

else

{

index=(int) t[i]-'0';

i=i+ta[index];

}

}

return -1;

}

void main()

{

int found;

clrscr();

printf("enter the text\n");

gets(t);

printf("enter the pattern\n");

scanf("%s",p);

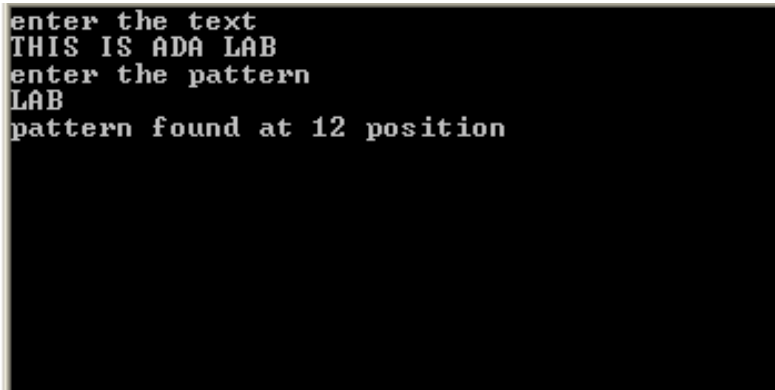
n=strlen(t);

m=strlen(p);

found=horsepool_algo();

if(found==-1)
```

```
printf("pattern not found\n");  
  
else  
  
printf("pattern found at %d position",found);  
  
getch();  
  
}
```

OUTPUT:

```
enter the text  
THIS IS ADA LAB  
enter the pattern  
LAB  
pattern found at 12 position
```


B) Binomial Co-efficient

Computing a binomial coefficient is a standard example of applying dynamic programming to a nonoptimization problem. you may recall from studies of elementary combinatorics that the binomial coefficient, denoted $c(n,k)$ is the number of combinations of k elements from an n -element set. $C(n,k)=c(n-1,k-1) + c(n-1,k)$ for $n>k>0$, $C(n,0)=c(n,n)=1$

Algorithm: binomial (n,k)

//compute $c(n,k)$ by the dynamic programming algorithm

//input: A pair of non negative integer $n \geq k \geq 0$

//output: the value of $c(n,k)$

For $j \leftarrow 0$ to n do

For $j \leftarrow$ to $\min(i,j)$ do

If $j=0$ or $j=k$

$C[i,j] \leftarrow 1$

Else $c[i,j] \leftarrow c[i-1,j-1] + c[i-1,j]$

Return $c[n,k]$

Complexity: Each entry takes $O(1)$ time to calculate and there are $O(n^2)$ of them. So this calculation of the coefficients takes $O(n^2)$ time. But it uses $O(n^2)$ space to store the coefficients

PROGRAM

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

```
void main()
```

```
{
```

```
int i,j,n,min,k,c[10][10];
```

```
clrscr();
```

```
printf("enter n and k values\n");
```

```
scanf("%d%d",&n,&k);
```

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

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

```
c[i][j]=0;
```

```
if(n>k)
```

```
{
```

```
for(i=0;i<=n;i++)
{
min=(i<j)?i:j;
for(j=0;j<=min;j++)
if(j==0 || (i==j))
c[i][j]=1;
else
c[i][j]=c[i-1][j-1]+c[i-1][j];
}
printf("the binomial coefficient\n");
for(i=0;i<n;i++)
{
for(j=0;j<k;j++)
if(i>=j)
printf("%4d",c[i][j]);
printf("\n");
}
printf("the value of c(%d %d) is %d",n,k,c[n-1][k-1]+c[n-1][k]);
}
else
printf("n must greater than k\n");
getch();
}
```

OUTPUT:

```
enter n and k values
6 3
the binomial coefficient
  1
 1  1
1  2  1
1  3  3
1  4  6
1  5 10
the value of c<6 3> is 20
```

LAB-12

Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

PRIM'S ALGORITHM

Prim's algorithm finds the minimum spanning tree for a weighted connected graph $G=(V,E)$ to get an acyclic subgraph with $|V|-1$ edges for which the sum of edge weights is the smallest. Consequently the algorithm constructs the minimum spanning tree as expanding sub-trees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

Algorithm : Prim(G)

```
// Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G=(V,E)$ 
//Output: ET,the set of edges composing a minimum spanning tree of  $G$ 
{
  VT  $\leftarrow$  {v0} //the set of tree vertices can be initialized with any vertex
  ET  $\leftarrow$  NULL
  for i  $\leftarrow$  0 to  $|V| - 1$  do
    find a minimum-weight edge  $e^* = (v^*, u^*)$  among all the edges  $(v, u)$ 
    such that  $v$  is in VT and  $u$  is in  $V-VT$ 
    VT  $\leftarrow$  VT  $\cup$  { $u^*$ }
    ET  $\leftarrow$  ET  $\cup$  { $e^*$ }
  return ET
}
```

Complexity: The time efficiency of prim's algorithm will be in $O(|E| \log |V|)$.

PROGRAM

```
#include<stdio.h>

int ne=1,min_cost=0;

void main()
```

```
{
int n,i,j,min,a,u,b,v,cost[20][20],parent[20];

clrscr();

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

scanf("%d",&n);

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

for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
scanf("%d",&cost[i][j]);

for(i=1;i<=n;i++)
parent[i]=0;

printf("\nThe edges of spanning tree are\n");

while(ne<n)
{
min=999;

for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(cost[i][j]<min)
{
min=cost[i][j];

a=u=i;

b=v=j;
}
}
}

while(parent[u])
```

```

u=parent[u];
while(parent[v])
v=parent[v];
if(u!=v)
{
printf("Edge %d\t(%d,%d)=%d\n",ne++,a,b,min);
min_cost=min_cost+min;
parent[v]=u;
}
cost[a][b]=cost[a][b]=999;
}
printf("\nMinimum cost=%d\n",min_cost);
getch();
}

```

OUTPUT:

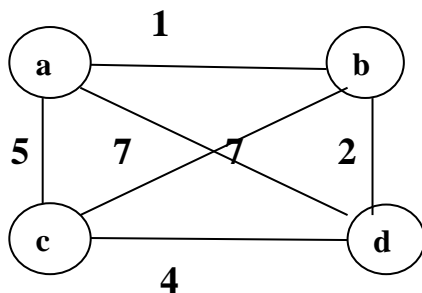
```

Enter the no. of vertices:4
Enter the cost matrix:
999 1 5 7
1 999 7 2
5 7 999 4
7 2 4 999

The edges of spanning tree are
Edge 1 <1,2>=1
Edge 2 <2,4>=2
Edge 3 <3,4>=4
Minimum cost=7

```

Input Graph



LAB-13

a. Implement Floyd's algorithm for the All-Pairs- Shortest-Paths Problem.

b. Compute the transitive closure of a given directed graph using Wars hall's algorithm.

A) FLOYD'S ALGORITHM

Floyd's algorithm is applicable to both directed and undirected graphs provided that they do not contain a cycle. It is convenient to record the lengths of shortest path in an n - by- n matrix D called the distance matrix. The element d_{ij} in the i th row and j th column of matrix indicates the shortest path from the i th vertex to j th vertex ($1 \leq i, j \leq n$). The element in the i th row and j th column of the current matrix $D(k-1)$ is replaced by the sum of elements in the same row i and k th column and in the same column j and the k th column if and only if the latter sum is smaller than its current value.

Algorithm Floyd($W[1..n, 1..n]$)

//Implements Floyd's algorithm for the all-pairs shortest paths problem

//Input: The weight matrix W of a graph

//Output: The distance matrix of shortest paths length

```
{
D ← W
for k ← 1 to n do
{
for i ← 1 to n do
{
for j ← 1 to n do
{
D[i,j] ← min (D[i, j], D[i, k]+D[k, j] )
}
}
}
return D
}
```

Complexity: The time efficiency of Floyd's algorithm is cubic i.e. $\Theta(n^3)$

PROGRAM

```
#include<stdio.h>

#include<conio.h>

int i,n,j,k,b[10][10];

void f()

{

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

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

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

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

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

}

main()

{

clrscr();

printf("Enter size\n");

scanf("%d",&n);

printf("Enter graph data\n");

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

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

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

f();

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

b[i][j]=0;

printf("The shortest path\n");

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

{

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

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



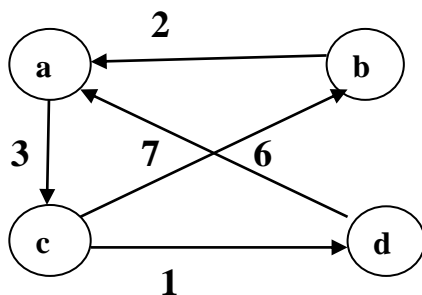
```
printf("\n");
}

getch();
}
```

OUTPUT:

```
Enter the size
4
Enter the graph data
0 999 3 999
2 0 999 999
999 7 0 1
6 999 999 0
Shortest path
      0      10      3      4
      2      0      5      6
      7      7      0      1
      6      16     9      0
_
```

Input Graph



B) WARS HALL'S ALGORITHM

The transitive closure of a directed graph with n vertices can be defined as the n -by- n boolean matrix $T=\{t_{ij}\}$, in which the element in the i th row ($1 \leq i \leq n$) and j th column ($1 \leq j \leq n$) is 1 if there exists a non trivial directed path from i th vertex to j th vertex, otherwise, t_{ij} is 0.

Warshall's algorithm constructs the transitive closure of a given digraph with n vertices through a series of n -by- n boolean matrices: $R(0), \dots, R(k-1), R(k), \dots, R(n)$ where, $R(0)$ is the adjacency matrix of digraph and $R(1)$ contains the information about paths that use the first vertex as intermediate. In general, each subsequent matrix in series has one more vertex to use as intermediate for its path than its predecessor. The last matrix in the series $R(n)$ reflects paths that can use all n vertices of the digraph as intermediate and finally transitive closure is obtained. The central point of the algorithm is that we compute all the elements of each matrix $R(k)$ from its immediate predecessor $R(k-1)$ in series.

Algorithm Warshall($A[1..n, 1..n]$)

//Implements Warshall's algorithm for computing the transitive closure

//Input: The Adjacency matrix A of a digraph with n vertices

//Output: The transitive closure of digraph

```
{
R(0) ← A
for k ← 1 to n do
{
for i ← 1 to n do
{
for j ← 1 to n do
{
R(k)[i,j] ← R(k-1) [i,j] or R(k-1) [i,k] and R(k-1) [k,j]
}
}
}
}
return R(n)
}
```

Complexity: The time efficiency of Warshall's algorithm is in $\Theta(n^3)$.

PROGRAM

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

```
void w(int c[10][10],int n)
{
    int i,j,k;
    for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
    if(c[i][k]==1)
    for(j=1;j<=n;j++)
    c[i][j]=c[i][j] | c[k][j];
}

void main()
{
    int c[10][10],i,j,n;
    clrscr();
    printf("Enter vertices\n");
    scanf("%d",&n);
    printf("Enter adjacency matrix\n");
    for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
    scanf("%d",&c[i][j]);
    w(c,n);
    printf("PATH MATRIX\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        printf("%d\t",c[i][j]);
        printf("\n");
    }
    getch();
}
```

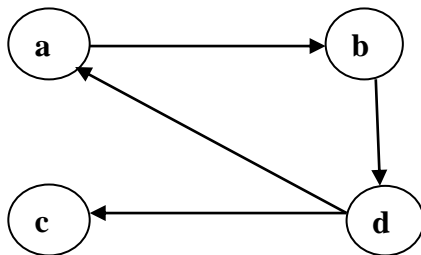
OUTPUT:

```

Enter vertices
4
Enter adjacency matrix
0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0
PATH MATRIX
1      1      1      1
1      1      1      1
0      0      0      0
1      1      1      1

```

Input Graph



LAB-14

Implement N Queen's problem using Back Tracking.

QUEEN'S PROBLEM

The n -queens problem consists of placing n queens on an $n \times n$ checker board in such a way that they do not threaten each other, according to the rules of the game of chess. Every queen on a checker square can reach the other squares that are located on the same horizontal, vertical, and diagonal line. So there can be at most one queen at each horizontal line, at most one queen at each vertical line, and at most one queen at each of the $4n-2$ diagonal lines. Furthermore, since we want to place as many queens as possible, namely exactly n queens, there must be exactly one queen at each horizontal line and at each vertical line. The concept behind backtracking algorithm which is used to solve this problem is to successively place the queens in columns. When it is impossible to place a queen in a column (it is on the same diagonal, row, or column as another token), the algorithm backtracks and adjusts a preceding queen

Algorithm NQueens (k, n)

```
//Using backtracking, this procedure prints all possible placements of n queens
//on an n x n chessboard so that they are non-attacking
{
  for i ← 1 to n do
  {
    if(Place(k,i) )
    {
      x[k] ← i
      if (k=n)
        write ( x[1...n])
      else
        Nqueens (k+1, n)
    }
  }
}
```

Algorithm Place(k, i)

```
//Returns true if a queen can be placed in kth row and ith column. Otherwise it
//returns false. x[] is a global array whose first (k-1) values have been set. Abs(r)
//returns the absolute value of r.
{
```

```
for j ← 1 to k-1 do
{
if ( (x[j]=i or Abs(x[j]-i) = Abs(j-k) )
{
return false
}
}
}
```

Complexity:

The power of the set of all possible solutions of the n queen's problem is n! and the bounding function takes a linear amount of time to calculate, therefore the running time of the n queens problem is $O(n!)$.

PROGRAM

```
#include<stdio.h>

int s[50][50];

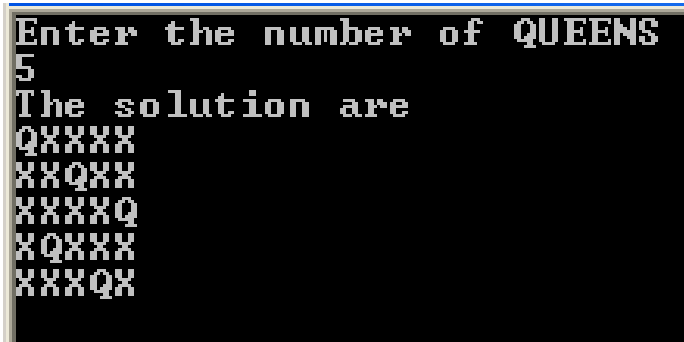
void dis(int m[],int n)
{
register int i,j;
for(i=0;i<n;i++)
s[i][m[i]]=1;
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
if(s[i][j])
printf("Q");
else
printf("X");
printf("\n");
}
}
```

```
    getch();
    exit(1);
}

int place(int m[],int k)
{
    int i;
    for(i=0;i<k;i++)
        if(m[i]==m[k] || (abs(m[i]-m[k])==abs(i-k)))
            return 0;
    return 1;
}

void main()
{
    int m[25],n,k;
    clrscr();
    printf("Enter the number of QUEENS\n");
    scanf("%d",&n);
    if(n==2 || n==3)
        printf("NO SOLUTION\n");
    else
        printf("The solution are\n");
    n--;
    for(m[0]=0,k=0;k<=n;m[k]+=1)
    {
        while(m[k]<=n&&!place(m,k))
            m[k]+=1;
        if(m[k]<=n)
            if(k==n)
```

```
dis(m,n+1);  
else  
k++,m[k]=-1;  
else  
k--;  
}  
getch();  
}
```

OUTPUT:

```
Enter the number of QUEENS  
5  
The solution are  
QXXXX  
XXQXX  
XXXXQ  
XQXXX  
XXXQX
```


ALGORITHMS LAB VIVA QUESTIONS

1.	What is an algorithm?
2.	What are the 3 ways of computing GCD?
3.	What is a pseudocode?
4.	What is a flow chart?
5.	Name some important problem types in algorithm
6.	What is an adjacency matrix?
7.	What is an adjacency Linked list
8.	Define weighted digraph
9.	What is a tree?
10.	What is forest?
11.	What is a Rooted tree?
12.	What is a free tree?
13.	Define ancestors,siblings,descendantsof a tree
14.	What is a binary tree and binary search tree?
15.	What are the two kinds of efficiency
16.	Define time and space efficiency
17.	What is a basic operation?
18.	How we estimate the running time
19.	Explain Brute force method. Give an example
20.	Explain Divide and conquer method. Give an example
21.	Explain Decrease and conquer method. Give an example
22.	Explain Dynamic programming method. Give an example
23.	Explain greedy technique. Give an example
24.	Define Order of growth
25.	What is an asymptotic notation?
26.	Explain basic efficiency classes?
27.	What is Master theorem?
28.	What is the time complexity for Mergesort, Quicksort, Binary search, Selection sort, DFS, BFS, heap sort
29.	Explain the 3 major variations of decrease and conquer
30.	What is DFS, BFS?

31.	What is a tree edge and back edge in DFS
32.	What is a tree edge and cross edge in BFS
33.	What are the algorithms for generating combinatorial objects?
34.	What is Exhaustive search?
35.	What is an AVL tree?
36.	What is an 2-3 tree?
37.	State the 2 different types of heap construction?
38.	Worst case efficiency of Horspool's algorithm
39.	What is the disadvantage of Horspool's algorithm
40.	What is Boyer Moore algorithm
41.	What is hashing?
42.	What is transitive closure?
43.	Explain warshall & floyds algorithm?
44.	Explain DFS & BFS
45.	What is a Memory function?
46.	What is a spanning tree?
47.	Differentiate Prims , Kruskals & Dijkstra's algorithm
48.	Explain Prims , Kruskals & Dijkstra's algorithm
49.	What is Union find algorithm?
50.	What is Huffman encoding?
51.	What is P,NP and NP complete problem?
52.	What is Bactracking?
53.	What is Branch and bound technique?
54.	Explain n-queens & sum of subsets problem with respect to backtracking