DEPARTMENT OF COMPUTER ENGINEERING & APPLICATIONS

Institute of Engineering & Technology

Lab Manual

NAME – SHUBHAM SINHA
UNIVERSITY ROLL NO.- 201510020
COURSE – B. TECH CS(CCV)
SECTION – 'O'-20
SUBJECT – DESIGN & ANALYSIS OF ALGORITHM
(BCSC-0807)

INDEX

PRACTICAL NO. 1.1

Aim: To sort the given list of elements by using Insertion sort.

Input : A list of elements stored in a array and the size of list or array

Output: A sorted list of elements stored in the same array(generally in the ascending order)

Algorithm :-

If the first few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place. This is called insertion sort. An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted).
Insertion sort is an example of an incremental algorithm; it builds the sorted sequence one number at a time.

INSERTION_SORT (A)

```
For j = 2 to length [A] do
    key = A[j]
   {Put A[j] into the sorted sequence A[1 . . j-1]
   i ← j –1
while i > 0 and A[i] > key do
      A[i+1] = A[i]
         i = i-1
   A[i+1] = key
```

Program:-

```
#include<stdio.h>
int main(){

  int i,j,s,temp,a[20];

  printf("Enter total elements: ");
  scanf("%d",&s);
```

```
 printf("Enter %d elements: ",s);
 for(i=0;i<s;i++)
    scanf("%d",&a[i]);

 for(i=1;i<s;i++){
    temp=a[i];
    j=i-1;
    while((temp<a[j])&&(j>=0)){
    a[j+1]=a[j];
       j=j-1;
    }
    a[j+1]=temp;
 }

 printf("After sorting: ");
 for(i=0;i<s;i++)
    printf(" %d",a[i]);

 return 0;
}
```


Execution:-

Enter total elements: 5
Enter 5 elements: 3 7 9 0 2
After sorting: 0 2 3 7 9

Complexity:

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

Questions:-

Q1.Insertion sort is 1.in-place 2.out-place 3.stable 4.non-stable
a.1 and 3 b.1 and 4 c.2 and 3 d.2 and 4
Ans:a

The array is sorted in-place because no extra memory is required. Stable sort retains the original ordering of keys when identical keys are present in the input data.

Q2.Sorting of playing cards is an example of
a.Bubble sort b. Insertion sort c.Selection sort d.Quick sort
Ans:b

Q3. For insertion sort, the number of entries we must index through when there are n elements in the array is
a. n entries b.n*n entries c.n-1 entries d.none of the above
Ans:c
Assuming there are n elements in the array we must index through n – 1 entries. For each entry we may need to examine and shift up to n – 1 other entries resulting in a O(n2) algorithm.

Q4.What is the average case complexity for insertion sort algorithm
a.O(n) b.O(n*n) c.O(nlogn) d.O(logn)
Ans:b

Q5.What is the worst case complexity for insertion sort algorithm
a.O(n) b.O(n*n) c.O(nlogn) d.O(logn)
Ans:b

PRACTICAL NO. 1.2

Aim: To sort the given list of elements by using Bubble Sort.

Input : A list of elements stored in a array and the size of list or array

Output: A sorted list of elements stored in the same array(generally in the ascending order)

Algorithm :

Bubble Sort is an elementary sorting algorithm. It works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

```
 BUBBLESORT (A)
for i ← 1 to length [A] do
   for j ← length [A] downto i +1 do
      If A[A] < A[j-1] then
         Exchange A[j] ↔ A[j-1]
```

PROGRAM:

```
#include<stdio.h>
int main(){

  int s,temp,i,j,a[20];

  printf("Enter total numbers of elements: ");
  scanf("%d",&s);

  printf("Enter %d elements: ",s);
  for(i=0;i<s;i++)
     scanf("%d",&a[i]);

  //Bubble sorting algorithm
  for(i=s-2;i>=0;i--){
     for(j=0;j<=i;j++){
        if(a[j]>a[j+1]){
```

```
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
          }
      }
  }

  printf("After sorting: ");
  for(i=0;i<s;i++)
     printf(" %d",a[i]);

  return 0;
}
```


Execution:-

Enter total numbers of elements: 5
Enter 5 elements: 6 2 0 11 9
After sorting: 0 2 6 9 11

Complexity:

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

Questions:-

Q1.The number of nested loops in a bubble sort algorithm

a.1
b.2
c.3
d.4

Ans:b

Q2. What is the output of bubble sort after the 1st pass given the following sequence of numbers: 25 57 48 37 12 92 86 33 23 15
a. 48 25 37 12 57 86 33 92 23 15

b. 25 48 37 12 57 86 33 23 15 92
c.12 25 33 37 48 57 86 92 23 15
d. 25 57 37 48 12 92 33 86 23 15
Ans:b

Q3. What is the output of bubble sort after the 2nd pass given the following sequence of numbers: 25 57 48 37 12 92 86 33 23 15
a. 48 25 37 12 57 86 33 92 23 15
b. 25 48 37 12 57 86 33 23 15 92
c. 25 37 12 48 57 33 23 15 86 92
d. 25 57 37 48 12 92 33 86 23 15
Ans:c

Q4. What is the output of bubble sort after the 3rd pass given the following sequence of numbers: 25 57 48 37 12 92 86 33 23 15
a. 48 25 37 12 57 86 33 92 23 15
b. 25 48 37 12 57 86 33 23 15 92
c. 25 37 12 48 57 33 23 15 86 92
d. 25 37 12 48 33 23 15 57 86 92
Ans:d

PRACTICAL NO. 1.3

Aim: To sort the given list of elements by using Heap sort.

Input : A list of elements stored in a array and the size of list or array

Output: A sorted list of elements stored in the same array(generally in the ascending order)

Algorithm :

The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full.

There are 3 basic procedures required to perform heap sort :

1. **Heapify**, is a procedure for manipulating heap data structures. It is given an array A and index i into the array. The subtree rooted at the children of A[i] are heap but node A[i] itself may possibly violate the heap property i.e., A[i] < A[2i] or A[i] < A[2i +1]. The procedure 'Heapify' manipulates the tree rooted at A[i] so it becomes a heap.

2. **Build-Heap**, We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . n] into a heap. Since the elements in the subarray A[$\lfloor$ n/2 $\rfloor$ +1 . . n] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

3. **Heap Sort**., The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . *n*]. Since the maximum element of the array stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*). If we now discard node n from the heap than the remaining elements can be made into heap.

The root of the tree A[1] and given index *i* of a node, the indices of its parent, left child and right child can be computed


```
PARENT (i)
      return floor(i/2)
LEFT (i)
      return 2i
RIGHT (i)
      return 2i + 1
```

heap property : A[PARENT (i)] ≥ A[i]

ALGORITHM

```
Heapify (A, i)
l ← left [i]
r ← right [i]
if l ≤ heap-size [A] and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size [A] and A[i] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
      Heapify (A, largest)
```

```
BUILD_HEAP (A)
heap-size (A) ← length [A]
For i ←  floor(length[A]/2) down to 1 do
      Heapify (A, i)
```

```
HEAPSORT (A)
BUILD_HEAP (A)
for i ← length (A) down to 2 do
exchange A[1] ↔ A[i]
heap-size [A] ← heap-size [A] - 1
Heapify (A, 1)
```

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void manage(int *, int);
void heapsort(int *, int, int);
int main()
{
 int arr[20];
 int i,j,size,tmp,k;
 printf("\n\t------- Heap sorting method -------\n\n");
 printf("Enter the number of elements to sort : ");
 scanf("%d",&size);
 for(i=1; i<=size; i++)
 {
   printf("Enter %d element : ",i);
   scanf("%d",&arr[i]);
   manage(arr,i);
 }
 j=size;
 for(i=1; i<=j; i++)
 {
   tmp=arr[1];
   arr[1]=arr[size];
   arr[size]=tmp;
   size--;
   heapsort(arr,1,size);
 }
 printf("\n\t------- Heap sorted elements -------\n\n");
 size=j;
 for(i=1; i<=size; i++)
     printf("%d ",arr[i]);
 getch();
 return 0;
}


void manage(int *arr, int i)
{
 int tmp;
 tmp=arr[i];
 while((i>1)&&(arr[i/2]<tmp))
 {
   arr[i]=arr[i/2];
   i=i/2;
 }
 arr[i]=tmp;
```

```
}


void heapsort(int *arr, int i, int size)
{
 int tmp,j;
 tmp=arr[i];
 j=i*2;
 while(j<=size)
 {
   if((j<size)&&(arr[j]<arr[j+1]))
      j++;
   if(arr[j]<arr[j/2])
      break;
   arr[j/2]=arr[j];
   j=j*2;
 }
 arr[j/2]=tmp;
}
```


Execution:

--------Heap sorting method--------

Enter the number of elements to sort : 5

Enter 1 element : 26
Enter 2 element : 976
Enter 3 element : 5
Enter 4 element : 69
Enter 5 element : 15

--------Heap sorted elements------

5 15 26 69 974

Complexity:

Best Case: O(nlogn)
Average Case: O(nlogn)
Worst Case: O(nlogn)

QUESTIONS:

Q1. What is the average case complexity for heap sort algorithm
a.O(n)
b.O(n*n)
c.O(nlogn)
d.O(logn)
Ans:c

Q2.What is the best case complexity for heap sort algorithm
a.O(n)
b.O(n*n)
c.O(nlogn)
d.O(logn)
Ans:c

Q3.The condition applicable for max-heap is
a.A[Parent(i)]>=A[i]
b.A[Parent(i)]<=A[i]
c.A[Parent(i+1)]>A[i]
d.A[Parent(i+1)]<a[i]
Ans:a

Q4.The condition applicable for min-heap is
a.A[Parent(i)]>=A[i]
b.A[Parent(i)]<=A[i]
c.A[Parent(i+1)]>A[i]
d.A[Parent(i+1)]<a[i]
Ans:b

Q5.The number of operations required for heap sort regardless of the order of input
a.O(1)
b.O(logn)
c.O(nlogn)
d.O(n*n)
Ans:c

PRACTICAL NO. 1.4

Aim: To sort the given list of elements by using Counting sort.

Input : A list of elements stored in a array and the size of list or array

Output: A sorted list of elements stored in the same array(generally in the ascending order)

Algorithm :

COUNTING_SORT (*A, B, k*)

1. for $i \leftarrow 1$ to k do
2. $c[i] \leftarrow 0$
3. for $j \leftarrow 1$ to n do
4. $c[A[j]] \leftarrow c[A[j]] + 1$
5. //c[i] now contains the number of elements equal to i
6. for $i \leftarrow 2$ to k do
7. $c[i] \leftarrow c[i] + c[i-1]$
8. // c[i] now contains the number of elements $\leq i$
9. for j $\leftarrow n$ downto 1 do
10. $B[c[A[i]]] \leftarrow A[j]$
11. $c[A[i]] \leftarrow c[A[j]] - 1$

PROGRAM:

```
#include <stdlib.h>

void printArray(int * array, int size){

  int curr;
  for(curr = 0; curr < size; curr++){
    printf("%d, ", array[curr]);
  }
  printf("\n");
}
```

```
int maximum(int * array, int size){

  int curr = 0;
  int max = 0;

  for(curr = 0; curr < size; curr++){
    if(array[curr] > max){ max = array[curr]; }
  }

  return max;
}

void countingSort(int * array, int size){

  int curr = 0;
  int max = maximum(array, size);
  int * counting_array = calloc(max, sizeof(int)); // Zeros out the array

  for(curr = 0; curr < size; curr ++){
    counting_array[array[curr]]++;
  }

  int num = 0;
  curr = 0;

  while(curr <= size){
    while(counting_array[num] > 0){
      array[curr] = num;
      counting_array[num]--;
      curr++;
      if(curr > size){ break; }
    }
    num++;
  }
  printArray(array, size);
}

int main(){

  int test1[] = {2, 6, 4, 3, 2, 3, 4, 6, 3, 4, 3, 5, 2, 6};
  int size1 = 14;

  countingSort(test1, size1);
```

```
  int test2[] = {5, 6, 7, 8, 5};
  int size2 = 5;

  countingSort(test2, size2);

  int test3[] = {8, 1, 2, 3, 3, 4};
  int size3 = 6;

  countingSort(test3, size3);

  return 0;
}
```


Complexity: O(nlogn)

QUESTIONS:

Q1. Whether Counting sort is Stable sort or not.?
Ans: Yes

Q2. If number of digits in a input number is very large, then counting sort useful or not.?
Ans: No

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


PRACTICAL NO. 1.5

Aim: To sort the given list of elements by using Selection sort.

Input : A list of elements stored in a array and the size of list or array

Output: A sorted list of elements stored in the same array(generally in the ascending order)

Algorithm :

```
SELECTION-SORT(A)
1.     for j ← 1 to n-1
2.          smallest ← j
3.           for i ← j + 1 to n
4.                   if A[ i ] < A[ smallest ]
5.                          smallest ← i
6.            Exchange A[ j ] ↔ A[ smallest ]
#include <stdio.h>
```

Program:

```
int main()
 {
    int data[100],i,n,steps,temp;
    printf("Enter the number of elements to be sorted: ");
    scanf("%d",&n);
    for(i=0;i<n;++i)
      {
       printf("%d. Enter element: ",i+1);
       scanf("%d",&data[i]);
    }
    for(steps=0;steps<n;++steps)
    for(i=steps+1;i<n;++i)
     {
         if(data[steps]>data[i])
/* To sort in descending order, change > to <. */
          {
             temp=data[steps];
             data[steps]=data[i];
             data[i]=temp;
         }
```

```
    }
    printf("In ascending order: ");
    for(i=0;i<n;++i)
        printf("%d  ",data[i]);
    return 0;
}
```


Output

```
Enter the number of elements to be sorted: 5
1. Enter element: 12
2. Enter element: 1
3. Enter element: 23
4. Enter element: 2
5. Enter element: 0
In ascending order: 0 1 2 12 23
```

Complexity:

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

QUESTIONS:

Q1. Whether Counting sort is Inplace sort or not.?
Ans: Yes

Q2. Number of Swap require in Selection sort in worst case is.?
Ans. N-1

PRACTICAL NO. 2.1

Aim: To sort the given list of elements by using Quick sort.

Input : A list of elements stored in a array and the size of list or array

Output: A sorted list of elements stored in the same array(generally in the ascending order)

Algorithm :

Quick sort works by partitioning a given array A[p . . r] into two non-empty sub array A[p . . q] and A[q+1 . . r] such that every key in A[p . . q] is less than or equal to every key in A[q+1 . . r]. Then the two subarrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

```
QuickSort
If p < r then
   q Partition (A, p, r)
   Recursive call to Quick Sort (A, p, q)
   Recursive call to Quick Sort (A, q + r, r)
```

Note that to sort entire array, the initial call Quick Sort (A, 1, length[A])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Partitioning the Array
Partitioning procedure rearranges the subarrays in-place.

```
PARTITION (A, p, r)
x ← A[p]
i ← p-1
j ← r+1
while TRUE do
   Repeat j ← j-1
```

```
   until A[j] ≤ x
   Repeat i ← i+1
   until A[i] ≥ x
   if i < j
      then exchange A[i] ↔ A[j]
      else return j
```

Partition selects the first key, A[p] as a pivot key about which the array will partitioned:
Keys ≤ A[p] will be moved towards the left .
Keys ≥ A[p] will be moved towards the right.

PROGRAM:

```
#include<stdio.h>

void quicksort(int [10],int,int);

int main(){
  int x[20],size,i;

  printf("Enter size of the array: ");
  scanf("%d",&size);

  printf("Enter %d elements: ",size);
  for(i=0;i<size;i++)
    scanf("%d",&x[i]);

  quicksort(x,0,size-1);

  printf("Sorted elements: ");
  for(i=0;i<size;i++)
    printf(" %d",x[i]);

  return 0;
}

void quicksort(int x[10],int first,int last){
   int pivot,j,temp,i;

    if(first<last){
       pivot=first;
       i=first;
       j=last;
```

```
        while(i<j){
           while(x[i]<=x[pivot]&&i<last)
              i++;
           while(x[j]>x[pivot])
              j--;
           if(i<j){
              temp=x[i];
               x[i]=x[j];
               x[j]=temp;
           }
        }

        temp=x[pivot];
        x[pivot]=x[j];
        x[j]=temp;
        quicksort(x,first,j-1);
        quicksort(x,j+1,last);

    }
}
```

Execution:

Enter size of the array: 5
Enter 5 elements: 3 8 0 1 2
Sorted elements: 0 1 2 3 8

Complexity:

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

QUESTIONS:

1.Quick sort is 1.in-place 2.non in-place 3.stable 4.non-stable a.1 and 3 b.1 and 4 c.2 and 3 d.2 and 4
Ans:b

2.What is the average case complexity for quick sort algorithm a.O(n) b.O(n*n) c.O(nlogn) d.O(logn)
Ans:c

3.What is the worst case complexity for quick sort algorithm a.O(n) b.O(n*n) c.O(nlogn) d.O(logn)
Ans:b

4.What is the best case complexity for quick sort algorithm a.O(n) b.O(n*n) c.O(nlogn) d.O(logn)
Ans:c

PRACTICAL NO. 2.2

Aim: To sort the given list of elements by using Merge sort.

Input : A list of elements stored in a array and the size of list or array

Output: A sorted list of elements stored in the same array(generally in the ascending order)

Algorithm :

MERGE (*A*, *p*, *q*, *r*)

1. $n_1 \leftarrow q - p + 1$
2. $n_2 \leftarrow r - q$
3. Create arrays L[1 . . n_1 + 1] and R[1 . . n_2 + 1]
4. **FOR** $i \leftarrow 1$ **TO** n_1
5. **DO** L[i] ← A[$p + i - 1$]
6. **FOR** $j \leftarrow 1$ **TO** n_2
7. **DO** R[j] ← A[$q + j$]
8. L[$n_1 + 1$] ← ∞
9. R[$n_2 + 1$] ← ∞
10. $i \leftarrow 1$
11. $j \leftarrow 1$
12. **FOR** $k \leftarrow p$ **TO** r
13. **DO IF** L[i] ≤ R[j]
14. **THEN** A[k] ← L[i]
15. $i \leftarrow i + 1$
16. **ELSE** A[k] ← R[j]
17. $j \leftarrow j + 1$

PROGRAM:

```
#include<stdio.h>
#define MAX 50

void mergeSort(int arr[],int low,int mid,int high);
void partition(int arr[],int low,int high);

int main(){
```

```
    int merge[MAX],i,n;

    printf("Enter the total number of elements: ");
    scanf("%d",&n);

    printf("Enter the elements which to be sort: ");
    for(i=0;i<n;i++){
        scanf("%d",&merge[i]);
    }

    partition(merge,0,n-1);

    printf("After merge sorting elements are: ");
    for(i=0;i<n;i++){
        printf("%d ",merge[i]);
    }

  return 0;
}

void partition(int arr[],int low,int high){

    int mid;

    if(low<high){
        mid=(low+high)/2;
        partition(arr,low,mid);
        partition(arr,mid+1,high);
        mergeSort(arr,low,mid,high);
    }
}

void mergeSort(int arr[],int low,int mid,int high){

    int i,m,k,l,temp[MAX];

    l=low;
    i=low;
    m=mid+1;

    while((l<=mid)&&(m<=high)){

        if(arr[l]<=arr[m]){
            temp[i]=arr[l];
```

```
            l++;
        }
        else{
            temp[i]=arr[m];
            m++;
        }
        i++;
    }

    if(l>mid){
        for(k=m;k<=high;k++){
            temp[i]=arr[k];
            i++;
        }
    }
    else{
        for(k=l;k<=mid;k++){
            temp[i]=arr[k];
            i++;
        }
    }

    for(k=low;k<=high;k++){
        arr[k]=temp[k];
    }
}
```

Execution:

Enter the total number of elements: 5
Enter the elements which to be sort: 2 5 0 9 1
After merge sorting elements are: 0 1 2 5 9

Complexity:

Best Case: O(nlogn)
Average Case: O(nlogn)
Worst Case: O(nlogn)

QUESTIONS:

Q1. Given the initial sequence : 3,41,52,26,38,57,9,49 ,at the last step in the merge sort .the sequences to be merged are
a.{ 3,26,52,41} and {38,49,57,9}
b.{ 3,26,41,52} and {9,38,49,57}
c. {3,9,41,52} and {26,38,49,57}
d. {3,9,26,38} and {41,49,52,57}
Ans. b

Q2. Mergesort makes two recursive calls. Which statement is true after these recursive calls finish, but before the merge step?
a. The array elements form a heap
b. Elements in each half of the array are sorted among themselves
c. Elements in the first half of the array are less than or equal to elements in the second half of the array
d. . Elements in the first half of the array are greater than or equal to elements in the second half of the array
Ans. B

Q3. Given the initial sequence: {85 24 63 47 17 31 96 50}, at the last step in the merge sort, the sequences to be merged are:
a). {24 47 63 85} and {17 31 50 96}
b). {17 24 31 47} and {50 63 85 96}
c). {24 85} {47 63} {17 31} and {50 96}
d). {17 24 31 47 63 85 96} and {50}
e). depends on choice of pivot
Ans. A

Q4.On each iteration,the size of the sorted lists in a merge sort
a. doubles
b. halves
c.remains the same
d. increases 4 times
Ans. a
On each iteration, the size of the sorted lists doubles, form 1 to 2 to 4 to 8 to 16 …to n.

Q5. What is the output of merge sort after the 1st pass given the following sequence of numbers: 12 13 1 5 7 9 11 14
a. 12 13 1 5 7 9 11 14
b. 1 5 12 13 7 9 11 14
c. 1 5 7 9 11 12 13 14
d. 1 5 7 9 12 13 11 14
Ans:a

Q6. What is the output of merge sort after the 2nd pass given the following sequence of numbers: 12 13 1 5 7 9 11 14
a. 12 13 1 5 7 9 11 14
b. 1 5 12 13 7 9 11 14
c. 1 5 7 9 11 12 13 14
d. 1 5 7 9 12 13 11 14
Ans:b

PRACTICAL NO. 3

Aim :To implement the DFS algorithm

Input : A connected undirected graph G=(v,e) with a weight function w :E-> R

Output : Starting and Finish time of visit.

Algorithm:

```
1   n ← number of nodes
2   Initialize visited[ ] to false (0)
3   for(i=0;i<n;i++)
4       visited[i] = 0;
5
6   void DFS(vertex i) [DFS starting from i]
7   {
8       visited[i]=1;
9       for each w adjacent to i
10          if(!visited[w])
11              DFS(w);
12  }
```

Program:

```
#include<stdio.h>
#include<conio.h>
int a [10][10],visited[10].n;
void main()
{
        int i,j;
        void search from(int);
        clrscr();
        printf("enter the no. of nodes\n");
        scanf("%d",&n);
        printf("enter the adjacency matrix\n");
for(i=1;<=n;i++)
for(j=1;<=n;j++)
scanf("%d",&a[i][j]);
for(i=1;i<=n;i++)
visited[i]=0;
printf("Depth First Path:");
for(i=1;i<=n;i++)
if(visited[i]==0)
searchfrom(i);
```

```
}
void search from(int k)
{
        int i;
        printf("%d\t",k);
        visited[k]=1;
        for(i=1;i<=n;i++)
        if(visited[i]==0)
        searchfrom(i);
        return;
}
```


Complexity:

O(V+E)

SAMPLE INPUT AND OUTPUT:

```
Enter the no. of nodes
4
Enter the adjacency matrix
0 1 0 1
0 0 1 1
0 0 0 1
0 0 0 0
Depth First Path
1    2    3    4
```

Questions:

Q1. Which of the following algorithms can be used to most efficiently determine the presence of a cycle in a given graph ?

Ans. DFS

Q2. Traversal of a graph is different from tree because

Ans. There can be a loop in graph so we must maintain a visited flag for every vertex

Q3. Let G be an undirected graph. Consider a depth-first traversal of G, and let T be the resulting depth-first search tree. Let u be a vertex in G and let v be the first new (unvisited) vertex visited after visiting u in the traversal. Which of the following statements is always true? (GATE CS 2000)

Ans If {u,v} is not an edge in G then u is a leaf in T

PRACTICAL NO. 4

Aim :To implement the BFS algorithm

Input : A connected undirected graph G=(v,e) with a weight function w :E-> R

Output : Visited Vertices in sequence

Algorithm:

BFS(G,s)

```
for each vertex u in V[G] - {s}
        do color[u] <-- white
                d[u] <-- infinity
                pi[u] <-- nil
color[s] <-- gray
d[s] <-- 0
pi[s] <-- nil
Q <-- {s} while Q != empty set
        do u <-- head[Q]
                for each v in Adj[u]
                        do if color[v] = white
                                then color[v] <-- gray
                                        d[v] <-- d[u] + 1
                                        pi[v] <-- u
                                        EnQueue(Q,v)
                DeQueue(Q)
                color[u] <-- black
```

Program:

```
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
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("\n Enter the number of vertices:");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
  q[i]=0;
  visited[i]=0;
 }
 printf("\n Enter graph data in matrix form:\n");
 for(i=1;i<=n;i++)
  for(j=1;j<=n;j++)
   scanf("%d",&a[i][j]);
 printf("\n Enter the starting vertex:");
 scanf("%d",&v);
 bfs(v);
 printf("\n The node which are reachable are:\n");
 for(i=1;i<=n;i++)
  if(visited[i])
   printf("%d\t",i);
  else
   printf("\n Bfs is not possible");
 getch();
}
```


Complexity: O(V+E)

Questions:

Q1. Given two vertices in a graph s and t, which of the two traversals (BFS and DFS) can be used to find if there is path from s to t?

Ans. Both BFS and DFS

Q2. Make is a utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program. What is standard graph algorithms is used by Make.

Ans. Topological Sorting

PRACTICAL NO. 5

Aim :To implement the minimum cost spanning tree using prim's algorithm

Input : A connected undirected graph G=(v,e) with a weight function w :E-> R

Output : A set of vertices of nodes that forms the minimum spanning tree

ALGORITHM:

Prim's algorithm has the property that the edges in the set A always form a single tree. We begin with some vertex v in a given graph G =(V, E), defining the initial set of vertices A. Then, in each iteration, we choose a minimum-weight edge (u, v), connecting a vertex v in the set A to the vertex u outside of set A. Then vertex u is brought in to A. This process is repeated until a spanning tree is formed.
Choose a node and build a tree from there selecting at every stage the shortest available edge that can extend the tree to an additional node.

ALGORITHM

```
MST_PRIM (G, w, v)
Q ← V[G]
for each u in Q do
   key [u] ←  ∞
key [r] ←  0
π[r] ← NIl
while queue is not empty do
   u ← EXTRACT_MIN (Q)
   for each v in Adj[u] do
      if v is in Q and w(u, v) < key [v]

   then π[v] ← w(u, v)
            key [v] ← w(u, v)
```

PROGRAM:

```
#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],u;
```

```
main()
{
        int m,c;
        cout <<"enterno of vertices";
        cin >> n;
        cout <<"ente no of edges";
        cin >> m;
        cout <<"\nEDGES Cost\n";
        for(k=1;k<=m;k++)
        {
                cin >>i>>j>>c;
                cost[i][j]=c;
        }
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
                if(cost[i][j]==0)
                cost[i][j]=31999;

        cout <<"ORDER OF VISITED VERTICES";
        k=1;
        while(k<n)
        {
                m=31999;
                if(k==1)
                {
                        for(i=1;i<=n;i++)
                                for(j=1;j<=m;j++)
                                if(cost[i][j]<m)
                                {
                                        m=cost[i][j];
                                        u=i;
                                }
                }
                else
                {
                        for(j=n;j>=1;j--)
                        if(cost[v][j]<m && visited[j]!=1 && visit[j]!=1)
                        {
                                visit[j]=1;
                                stk[top]=j;
                                top++;
                                m=cost[v][j];
                                u=j;
                        }
                }
```

```
                cost[v][u]=31999;
                v=u;
                cout<<v << " ";
                k++;
                visit[v]=0; visited[v]=1;
        }
}
```


Execution:

enterno of vertices7
ente no of edges9

EDGES Cost
1 6 10
6 5 25
5 4 22
4 3 12
3 2 16
2 7 14
5 7 24
4 7 18
1 2 28
ORDER OF VISITED VERTICES1 6 5 4 3 2

QUESTIONS:

Q1. Whether back tracking is possible or not in prims algorithm.
Ans: Yes

Q2. While executing Prims algorithm, whether graph remains connected or Disconnect.
Ans: Remains Connected

DESIGN ANALYSIS

The performance of Prim's algorithm depends of how we choose to implement the priority queue Q.

Sparse graphs are those for which |E| is much less than |V|2 i.e., |E| << |V|2 we preferred the adjacency-list representation of the graph in this case. On the other hand, dense graphs are those for which |E| is graphs are those for which |E| is close to |V|2. In this case, we like to represent graph with adjacency-matrix representation.

$$O(\sum u(V+dg[u])) = O(V2+E)$$
$$= O(V2)$$

PRACTICAL NO. 6

Aim :To implement the minimum cost spanning tree using Kruskal algorithm

Input : A connected undirected graph G=(v,e) with a weight function w :E-> R

Output : A set of vertices of nodes that forms the minimum spanning tree

ALGORITHM:

MST_KRUSKAL(G)

for each vertex *v* in V[G]
do define set S(v) ← {v}
Initialize priority queue Q that contains all edges of G, using the weights as keys
A ← { } ▷ A will ultimately contains the edges of the MST
while A has less than n − 1 edges
do Let set S(v) contains v and S(u) contain u
if S(v) ≠ S(u)
then Add edge (u, v) to A
Merge S(v) and S(u) into one set i.e., union
return A

program:

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
int cost[10][10],i,j,k,n,m,c,visit,visited[10],l,v,count,count1,vst,p;

main()
{
int dup1,dup2;
cout<<"enter no of vertices";
cin >> n;
cout <<"enter no of edges";
cin >>m;
cout <<"EDGE Cost";
for(k=1;k<=m;k++)
{
cin >>i >>j >>c;
cost[i][j]=c;
```

```
cost[j][i]=c;
}
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
if(cost[i][j]==0)
cost[i][j]=31999;
visit=1;
while(visit<n)
{
v=31999;
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
if(cost[i][j]!=31999 && cost[i][j]<v  && cost[i][j]!=-1 )
{
int count =0;
for(p=1;p<=n;p++)
{
if(visited[p]==i || visited[p]==j)
count++;
}
if(count >= 2)
{
for(p=1;p<=n;p++)
if(cost[i][p]!=31999 && p!=j)
dup1=p;
for(p=1;p<=n;p++)
if(cost[j][p]!=31999 && p!=i)
dup2=p;

if(cost[dup1][dup2]==-1)
continue;
}
l=i;
k=j;
v=cost[i][j];
}
cout <<"edge from " <<l <<"-->"<<k;
cost[l][k]=-1;
cost[k][l]=-1;
visit++;
int count=0;
count1=0;
for(i=1;i<=n;i++)
{
if(visited[i]==l)
```

```
count++;
if(visited[i]==k)
count1++;
}
if(count==0)
visited[++vst]=l;
if(count1==0)
visited[++vst]=k;
}
}
```


Execution:

```
enter no of vertices4
enter no of edges4
EDGE Cost
1 2 1
2 3 2
3 4 3
1 3 3

edge from 1–>2edge from 2–>3edge from 1–>3
```

Questions:

q1.How many edges does a minimum spanning tree has?
Ans: A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph

Q2. *What is Minimum Spanning Tree?*
Ans: Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

PRACTICAL NO.7

Aim : To implement the shortest path algorithm using dijkstra's algorithm

Input : To implement the minimum cost spanning tree using prim's algorithm

Output : Shortest path from source to each vertex

ALGORITHM:

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

DIJKSTRA (G, w, s)

INITIALIZE SINGLE-SOURCE (G, s)

S ← { } // S will ultimately contains vertices of final shortest-path weights from s

Initialize priority queue Q i.e., Q ← V[G]

while priority queue Q is not empty do

u ← EXTRACT_MIN(Q) // Pull out new vertex

S← S□□□{*u*}
// Perform relaxation for each vertex *v* adjacent to u

for each vertex v in Adj[u] do

Relax (*u*, *v*, *w*)

PROGRAM:

```
void dijkstra(int graph[V][V], int src)
{
    int dist[V];     // The output array.  dist[i] will hold the shortest
                     // distance from src to i

    bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
                    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
      dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V-1; count++)
    {
      // Pick the minimum distance vertex from the set of vertices not
      // yet processed. u is always equal to src in first iteration.
      int u = minDistance(dist, sptSet);

      // Mark the picked vertex as processed
      sptSet[u] = true;

      // Update dist value of the adjacent vertices of the picked vertex.
      for (int v = 0; v < V; v++)

        // Update dist[v] only if is not in sptSet, there is an edge from
        // u to v, and total weight of path from src to  v through u is
        // smaller than current value of dist[v]
        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                                    && dist[u]+graph[u][v] < dist[v])
          dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}
int main()
{
   /* Let us create the example graph discussed above */
   int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
```

```
                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                {0, 0, 4, 0, 10, 0, 2, 0, 0},
                {0, 0, 0, 14, 0, 2, 0, 1, 6},
                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                {0, 0, 2, 0, 0, 0, 6, 7, 0}
               };

   dijkstra(graph, 0);

   return 0;
}
```

Exexution:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

QUESTIONS:

Q1. Whether Dijkstra algo can handle –ve edge in graph?
Ans: No

Q2. Whether Dijkstra algo can handle –ve edge cycle.?
Ans: No

Q3 Whether Dijkstra algo can handle +ve edge cycle.?
Ans: yes

DESIGN ANALSIS

Like Prim's algorithm, Dijkstra's algorithm runs in O(|E|lg|V|) time.

PRACTICAL NO.8

Aim : To implement 0/1 knapsack Problem.

Input : 0/1 Knapsack problem

Output : Optimal solution of 0/1 knapsack problem using dynamic programming

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 IFf wi ≤ w
            THEN IF  vi + c[i-1, w-wi]
               THEN c[i, w] = vi + c[i-1, w-wi]
               ELSE c[i, w] = c[i-1, w]
            ELSE
               c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at *c*[*n. w*] and tracing backwards where the optimal values came from. If *c*[*i, w*] = *c*[*i*-1, *w*] item *i* is not part of the solution, and we are continue tracing with *c*[*i*-1, *w*]. Otherwise item *i* is part of the solution, and we continue tracing with *c*[*i*-1, *w*-*W*].

PROGRAM

```
#include<iostream.h>

#include<conio.h>

class knapsack

{
```

```
int w[20],v[20],d[10][10],n,c,i,j;
public:
void input( );
void knap( );
int max(int,int);
void output( );
};
void knapsack::input()
{
cout<<"\n\t Knapsack using Dynamic programming";
cout<<"\nHow many items u need:";
cin>>n;
cout<<"\nEnter the capacity";
cin>>c;
cout<<"\nEnter "<<n<<"items:";
for(i=1;i<=n;i++)
{
cout<<"\nEnter the "<<i<<"Weight:";
cin>>w[i];
cout<<"\nEnter the "<<i<<" values";
cin>>v[i];
}
}
```

```
void knapsack::knap()
{
for(i=0;i<=n;i++)
d[i][0]=0;
for(j=1;j<=c;j++)
d[0][j]=0;

for(i=1;i<=n;i++)
{
for(j=1;j<=c;j++)
{
if((j-w[i])<0)
d[i][j]=d[i-1][j];
else
d[i][j]=max(d[i-1][j],v[i]+(d[i-1][j-w[i]]));
}
}
}
void knapsack::output()
{
cout<<"Highest value computation for"<< n <<" items:"<<endl;
for(i=0;i<=n;i++)
{
```

```
for(j=0;j<=c;j++)
{
```



```
cout<<"\t"<<d[i][j];
}
cout<<endl;
}
}

int knapsack::max(int a,int b)
{
if(a>b)
return a;
else
return b;
}

void main()
{
clrscr();
knapsack k;
k.input();
k.knap();
```

```
k.output();
getch();
}
```


Execution:

```
Knapsack Using Dynamic Programming
How many items you need : 3
Enter the Capacity 10
Enter 3 items :
Enter the 1 Weight : 2
Enter the 1 Weight : 5
Enter the 2 Weight : 2
Enter the 2 Weight : 3
Enter the 3 Weight : 4
Enter the 3 Weight : 6
Highest value computation for 3 items:
0       0       0       0       0       0       0       0       0
0       0
        0       0       5       5       5       5       5       5       5
5       5
        5       0       5       5       8       8       8       8       8
8       8
        8       0       5       5       11      8       11      11      14
14      14
```

PRACTICAL NO.9

Aim : To implement Matrix chain Multiplication Problem.

Input : All Matrices Coordinates.

Output : Minimum number of Multiplications

Algorithm:

```
Matrix-Chain(array p[1 .. n], int n) {
    Array s[1 .. n − 1, 2 .. n];
    FOR i = 1 TO n DO m[i, i] = 0;                    // initialize
    FOR L = 2 TO n DO {                               // L=length of
subchain
        FOR i = 1 TO n − L + 1 do {
            j = i + L − 1;
            m[i, j] = infinity;
            FOR k = i TO j − 1 DO {                   // check all splits
                q = m[i, k] + m[k + 1, j] + p[i − 1] p[k] p[j];
                IF (q < m[i, j]) {
                    m[i, j] = q;
                    s[i, j] = k;
                }
            }
        }
    }
    return m[1, n](final cost) and s (splitting markers);
    }
```

Program:

```
int MatrixChainOrder(int p[], int n)
{

    int m[n][n];

    int i, j, k, L, q;

    // cost is zero when multiplying one matrix.
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L=2; L<n; L++)
    {
        for (i=1; i<=n-L+1; i++)
```

```
        {
            j = i+L-1;
            m[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n-1];
}

int main()
{
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                        MatrixChainOrder(arr, size));

    getchar();
    return 0;
}
```

Execution: 12

Questions:

Q1. Four matrices M1, M2, M3 and M4 of dimensions pxq, qxr, rxs and sxt respectively can be multiplied is several ways with different number of total scalar multiplications. For example, when multiplied as ((M1 X M2) X (M3 X M4)), the total number of multiplications is pqr + rst + prt. When multiplied as (((M1 X M2) X M3) X M4), the total number of scalar multiplications is pqr + prs + pst. If p = 10, q = 100, r = 20, s = 5 and t = 80, then the number of scalar multiplications needed is

Ans: 19000

Q2. Which technique is best to solve MCM.?

Ans. Dynamic Programming

PRACTICAL NO.10

Aim : To implement LCS Problem.

Input : Two Strings

Output : Largest Common Subsequence of two strings

Algorithm:

```
function LCSLength(X[1..m], Y[1..n])
    C = array(0..m, 0..n)
    for i := 0..m
       C[i,0] = 0
    for j := 0..n
       C[0,j] = 0
    for i := 1..m
        for j := 1..n
            if X[i] = Y[j]
                C[i,j] := C[i-1,j-1] + 1
            else
                C[i,j] := max(C[i,j-1], C[i-1,j])
    return C[m,n]
```

Program:

```
int lcs( char *X, char *Y, int m, int n )
{
   int L[m+1][n+1];
   int i, j;

   /* Following steps build L[m+1][n+1] in bottom up fashion. Note
      that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1]
*/
   for (i=0; i<=m; i++)
   {
     for (j=0; j<=n; j++)
     {
       if (i == 0 || j == 0)
         L[i][j] = 0;

       else if (X[i-1] == Y[j-1])
         L[i][j] = L[i-1][j-1] + 1;

       else
         L[i][j] = max(L[i-1][j], L[i][j-1]);
```

```
      }
    }

    /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";

  int m = strlen(X);
  int n = strlen(Y);

  printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

  return 0;
}
```


Execution:

4

Questions:

Q1. We use dynamic programming approach when.

Ans. The solution has optimal substructure

Q2 Consider two strings A = "qpqrr" and B = "pqprqrp". Let x be the length of the longest common subsequence (not necessarily contiguous) between A and B and let y be the number of such longest common subsequences between A and B. Then x + 10y = ___.

Ans. 34

PRACTICAL NO.11

Aim : To implement 4 Queen Problem.

Input : 4 Queen problem

Output : Optimal solution of 4 Queen problem

Algorithm:

Place(k,i)
For j→1 to k-1
Do if(x[j]=i) or Abs (x[i]-i)=Abs(j-k)
Then return False
NQUEENS(k,n)
For i→1 to n do
If PLACE (k,i)
Then x[k]→i
If k=n then print x[1….n]
Else NQUEENS (K+1, N)

Program:

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

char a[10][10];
int n;

void printmatrix() {
  int i, j;
  printf("\n");

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
      printf("%c\t", a[i][j]);
    printf("\n\n");
  }
}

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

```
    if (a[row][i] == 'Q') {
      return (i);
      break;
    }
}
```



```
int feasible(int row, int col) {
  int i, tcol;
  for (i = 0; i < n; i++) {
    tcol = getmarkedcol(i);
    if (col == tcol || abs(row - i) == abs(col - tcol))
      return 0;
  }
  return 1;
}

void nqueen(int row) {
  int i, j;
  if (row < n) {
    for (i = 0; i < n; i++) {
      if (feasible(row, i)) {
        a[row][i] = 'Q';
        nqueen(row + 1);
        a[row][i] = '.';
      }
    }
  } else {
    printf("\nThe solution is:- ");
    printmatrix();
  }
}

int main() {
  int i, j;

  printf("\nEnter the no. of queens:- ");
  scanf("%d", &n);

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      a[i][j] = '.';

  nqueen(0);
  return (0);
```

Execution:

Enter the no. of queens:- 4

The solution is:-

. Q . .

. . . Q

Q . . .

. . Q .

The solution is:-

. . Q .

Q . . .

. . . Q

. Q

Questions:

Q1. Which technique is used to solve N Queen problem.?
Ans: Backtraking

Q2. How to check diagonal conflict for Queen1(i,j) & Queen2(k,l).?
Ans: |i-k| = |j-l|

.