Assignment

Name: - Mann Kothari

Reg. No.: - 22BCB7064

Course Title: - Design and Analysis of Algorithm (Embedded Lab)

Course Code: - CSE3023

Slot: - L21+L22

Submitted to: - Prof. Tanikella Divya Naga Pavani

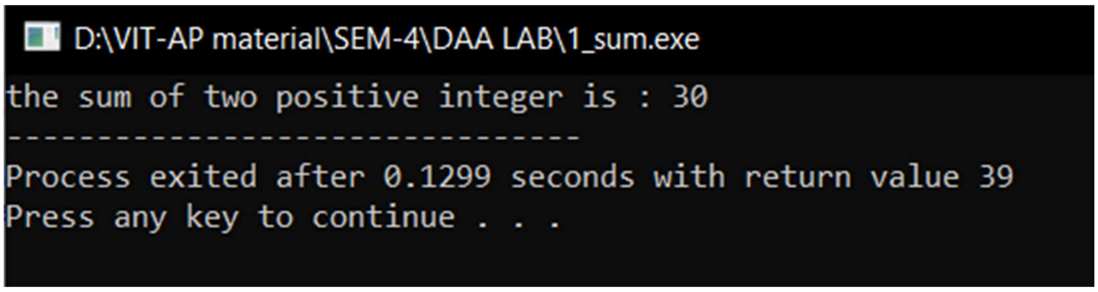
- **INDEX**

- Implement the following problems using recursion in C/C++.

1. Find the Sum of two Positive Integers.

```c
#include<stdio.h>
void main()
{
	int a=10, b=20;
	printf("the sum of two positive integer is : %d",a+b);

}
```

- **Output:**

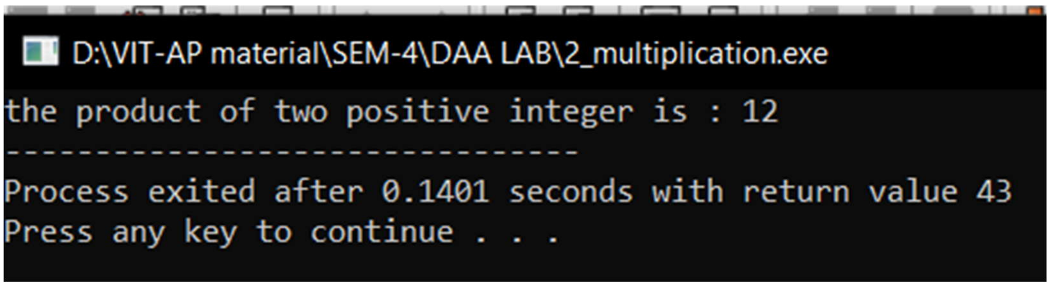


```
D:\VIT-AP material\SEM-4\DAA LAB\1_sum.exe
the sum of two positive integer is : 30
--------------------------------
Process exited after 0.1299 seconds with return value 39
Press any key to continue . . .
```

2. Find the Multiplication of two Positive Integers.

```c
#include<stdio.h>
void main()
{
	int a=2, b=6;
	printf("the product of two positive integer is : %d",a*b);

}
```

- **Output:**



```
D:\VIT-AP material\SEM-4\DAA LAB\2_multiplication.exe
the product of two positive integer is : 12
--------------------------------
Process exited after 0.1401 seconds with return value 43
Press any key to continue . . .
```

3. Implement N terms of the Fibonacci series using recursion and find out the number of functions calls for different values of n.

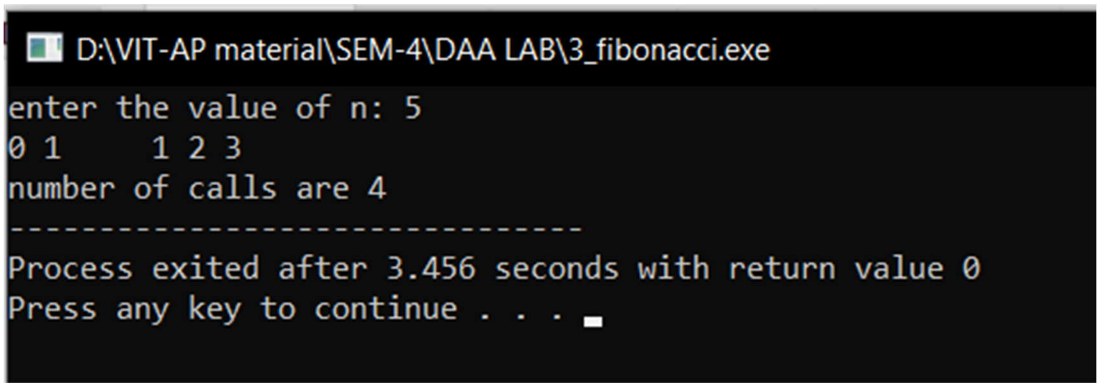
Algorithm: -

```
Algorithm fibonacci (n: nonnegative integer)
    if n = 0 then return 0;
    else
        x: = 0;
        y: = 1;
        for i: =1 to n -1
            z: = x + y;
            x: = y;
            y: = z;
            fibonacci(n-1);
{output is the nth Fibonacci number}
```

Code: -

```c
#include<stdio.h>
void fibonacci(int n, int *calls){
    (*calls)++;
    static int n1=0,n2=1,n3;
    if(n>0){
         n3 = n1 + n2;
         n1 = n2;
         n2 = n3;
         printf("%d ",n3);
         fibonacci(n-1, calls);
    }
}
int main()
{
     int n;
     int n1=0, n2=1;
     int calls=0;
     printf("enter the value of n: ");
     scanf("%d", &n);
     printf("%d %d\t", n1,n2);
     fibonacci(n-2, &calls);
     printf("\nnumber of calls are %d", calls);
     return 0;
}
```

- **Output:**



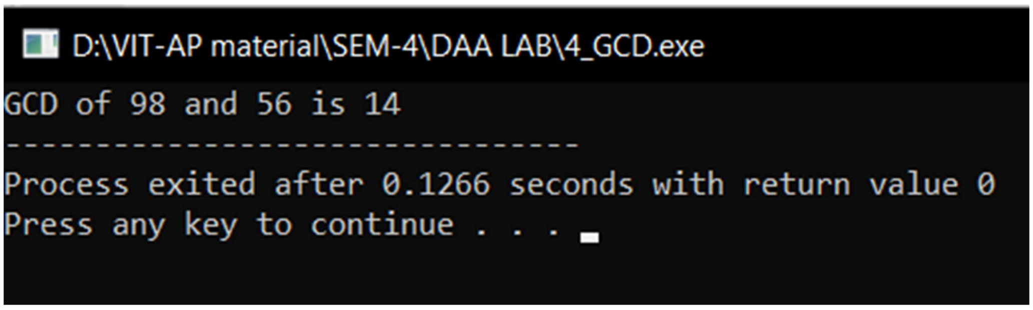
D:\VIT-AP material\SEM-4\DAA LAB\3_fibonacci.exe

```
enter the value of n: 5
0 1     1 2 3
number of calls are 4
--------------------------------
Process exited after 3.456 seconds with return value 0
Press any key to continue . . .
```

4. Find the GCD of two numbers.

```c
#include <stdio.h>
#include <math.h>
int gcd(int a, int b)
{
    int result = ((a < b) ? a : b);
    while (result > 0) {
        if (a % result == 0 && b % result == 0) {
            break;
        }
        result--;
    }
    return result;
}

int main()
{
    int a = 98, b = 56;
    printf("GCD of %d and %d is %d ", a, b, gcd(a, b));
    return 0;
}
```

- **Output:**


D:\VIT-AP material\SEM-4\DAA LAB\4_GCD.exe

```
GCD of 98 and 56 is 14
--------------------------------
Process exited after 0.1266 seconds with return value 0
Press any key to continue . . .
```

5. Find the factorial of a given number.

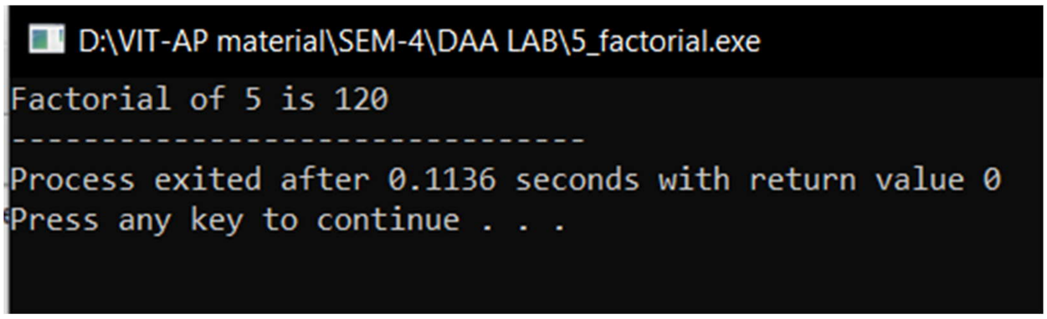
Algorithm: -

```
Algorithm factorial(n)
{
    res: = 1;
    for i: = n to 0 step-1 do
    {
        res: = res*I;
    }
return res;
```

Code: -

```c
#include <stdio.h>
int factorial(int n)
{
    int result = 1, i;
    for (i = n; i > 0; i--) {
        result *= i;
    }
    return result;
}
int main()
{
    int num = 5;
    printf("Factorial of %d is %d", num, factorial(num));
    return 0;
}
```

- **Output:**



```
D:\VIT-AP material\SEM-4\DAA LAB\5_factorial.exe
Factorial of 5 is 120
--------------------------------
Process exited after 0.1136 seconds with return value 0
Press any key to continue . . .
```

6. Towers of Hanoi.

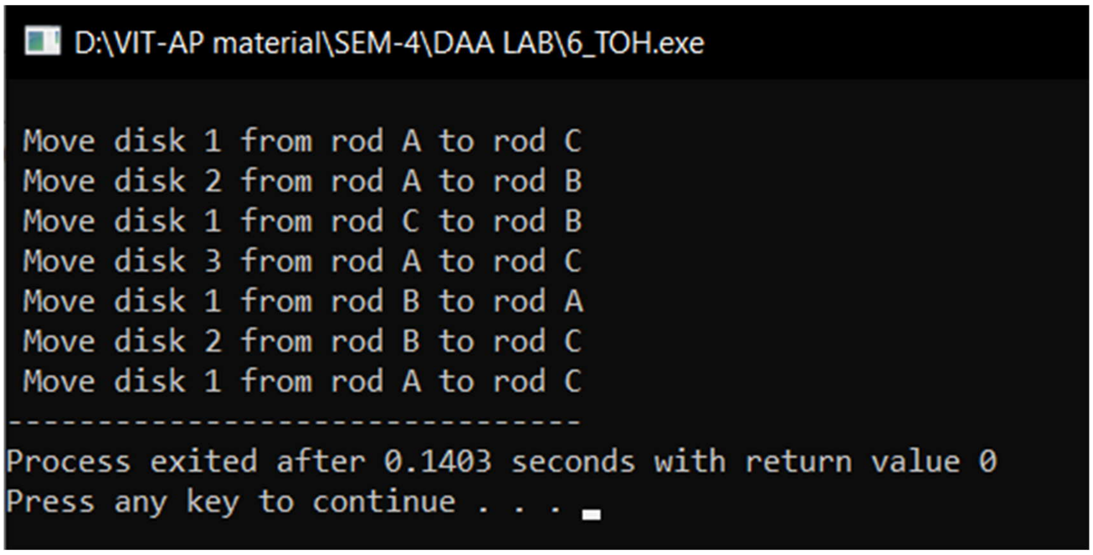
Algorithm: -

```
Algorithm TowerOfHanoi(n, from_rod, to_rod, aux_rod)
{
    if (n= =1)then
    {
        write(" Move disk 1 from rod %c to rod %c");
        return;
    }
     TowerOfHanoi(n-1, from_rod, aux_rod, to_rod);
     Write("Move disk %d from rod %c to rod %c");
     TowerOfHanoi((n-1, aux_rod, to_rod, from_rod);
}
```

Code: -

```
#include <stdio.h>
void TowerOfHanoi(int n, char from_rod, char to_rod, char
aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c",
from_rod, to_rod);
        return;
    }
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n,
from_rod, to_rod);
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 3;
    TowerOfHanoi(n, 'A', 'C', 'B');
    return 0;
}
```

- **Output:**



```
D:\VIT-AP material\SEM-4\DAA LAB\6_TOH.exe

Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
--------------------------------
Process exited after 0.1403 seconds with return value 0
Press any key to continue . . .
```

7. Permutation generator

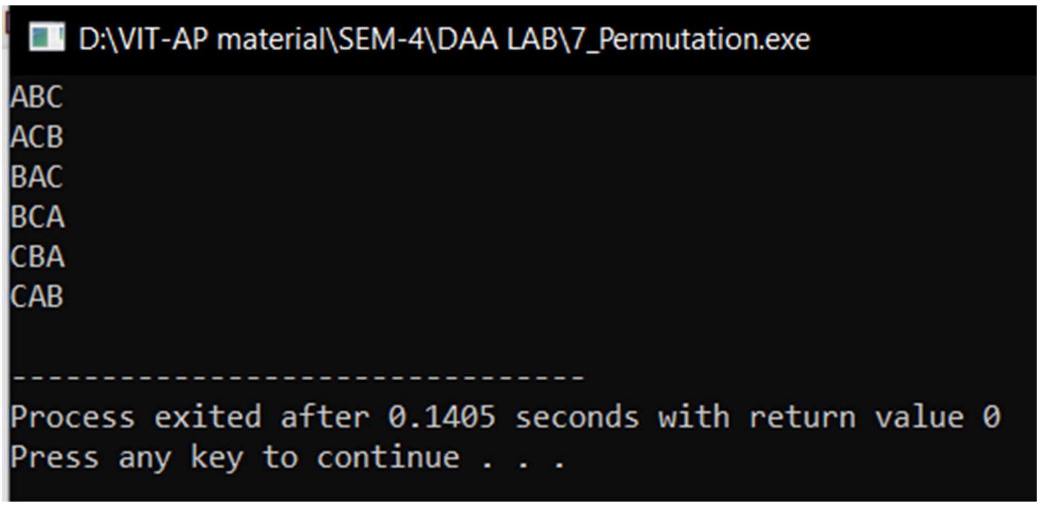
    input: {a.b.c}

    output: a,b,c; a,c,b; b,a,c; b,c,a; c,a,b; c,b,a;

   all permutations without repetition.

```c
#include <stdio.h>
#include <string.h>
void swap(char* x, char* y)
{
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
void permute(char* a, int l, int r)
{
    int i;
    if (l == r)
        printf("%s\n", a);
    else {
        for (i = l; i <= r; i++) {
            swap((a + l), (a + i));
            permute(a, l + 1, r);
            swap((a + l), (a + i));
        }
    }
}
int main()
{
    char str[] = "ABC";
    int n = strlen(str);
    permute(str, 0, n - 1);
    return 0;
}
```

- **Output:**

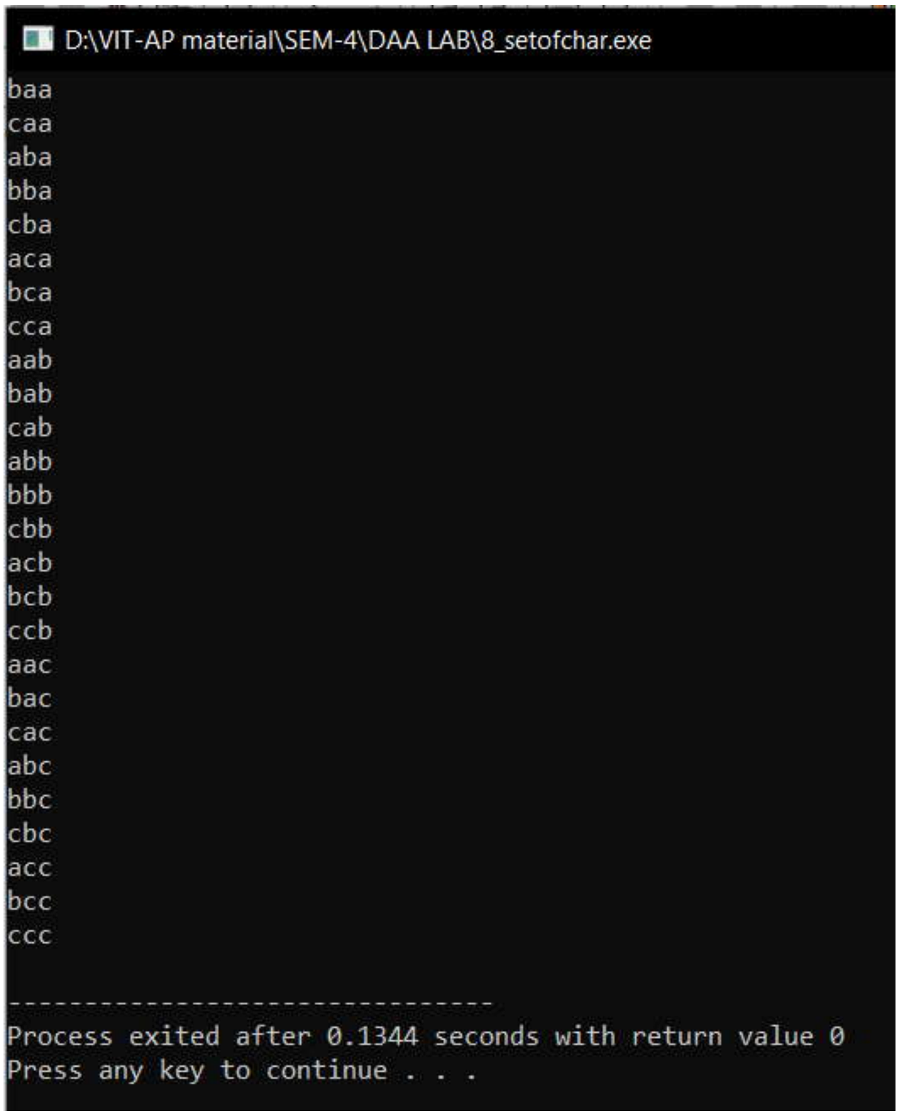


```
D:\VIT-AP material\SEM-4\DAA LAB\7_Permutation.exe
ABC
ACB
BAC
BCA
CBA
CAB

--------------------------------
Process exited after 0.1405 seconds with return value 0
Press any key to continue . . .
```

8. Given a set of characters and a positive integer k, print all possible strings of length k that can be formed from the given set.
   {I/P: a.b. K=3, O/P: aaa, aab, abb, aba, …}

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

void generateStrings(const char *characters, int k, char
*current)
{
    if (k == 0)
    {
        printf("%s\n", current);
        return;
    }

    for (int i = 0; i < strlen(characters); i++)
    {
        current[k - 1] = characters[i];
        generateStrings(characters, k - 1, current);
    }
}

int main()
{
    const char *input_characters = "abc"; // replace with your
set of characters
    int k = 3;                            // replace with your
desired length

    char current[k + 1];
    current[k] = '\0'; // null-terminate the string

    generateStrings(input_characters, k, current);

    return 0;
}
```

- **Output:**



D:\VIT-AP material\SEM-4\DAA LAB\8_setofchar.exe

```
baa
caa
aba
bba
cba
aca
bca
cca
aab
bab
cab
abb
bbb
cbb
acb
bcb
ccb
aac
bac
cac
abc
bbc
cbc
acc
bcc
ccc

--------------------------------
Process exited after 0.1344 seconds with return value 0
Press any key to continue . . .
```

9. Write a c/c++ program to implement Linear Search.
   Algorithm: -

   Procedure search(i, j, x: i, j, x integers, $1 \leq i \leq j \leq n$)

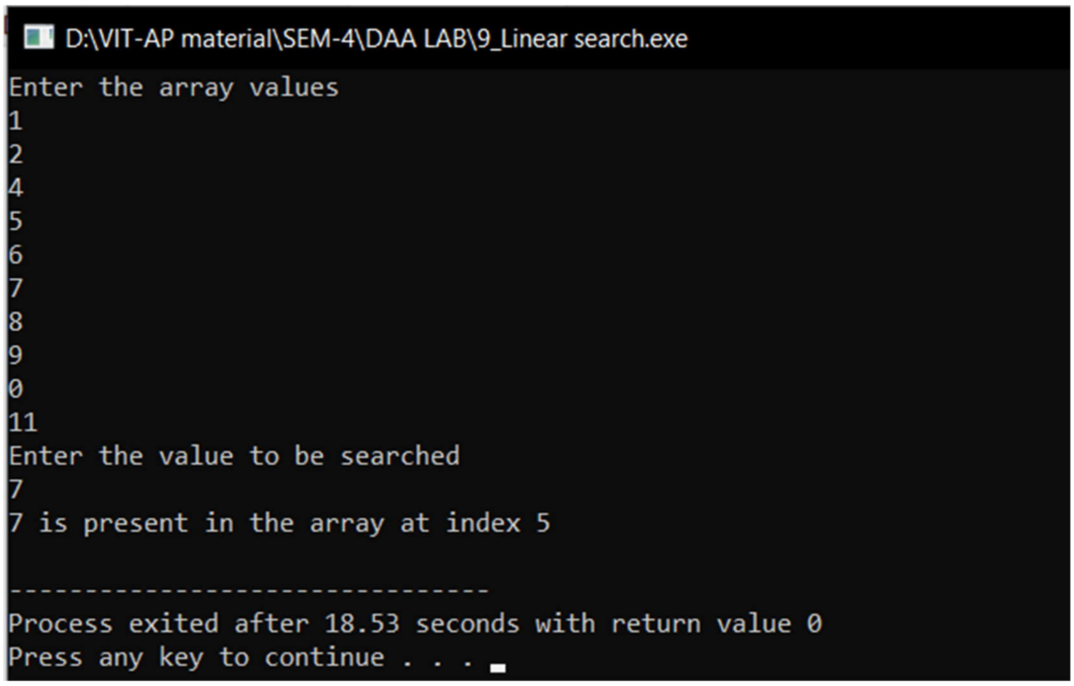
```
if a; = x then
return i
else if i j then
return 0
else
return search(i + 1, j, x) {output is the location of x in a1, a2,..., an if it appears;
otherwise it is 0)
```

Time Complexity: -
O(n)

Code: -

```
#include <stdio.h>
int linear_search(int a[], int n, int x)
{
    int i, flag = 0, index;
    for (i = 0; i < n; i++)
    {
        if (a[i] == x)
        {
            flag = 1;
            index = i;
        }
    }
    if (flag == 1)
        printf("%d is present in the array at index %d\n", x, index);
    else
        printf("%d is not present in the array \n", x);
    return 0;
}

void main()
{
    int arr[10], i, n = 10, x;
    printf("Enter the array values\n");
    for (i = 0; i < 10; i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Enter the value to be searched\n");
    scanf("%d", &x);
    linear_search(arr, n, x);
}
```

- **<u>Output:</u>**



D:\VIT-AP material\SEM-4\DAA LAB\9_Linear search.exe

```
Enter the array values
1
2
4
5
6
7
8
9
0
11
Enter the value to be searched
7
7 is present in the array at index 5

--------------------------------
Process exited after 18.53 seconds with return value 0
Press any key to continue . . .
```

10. Write a c/c++ program to implement Binary Search.

Algorithm: -

Binary Search Algorithm

```
1.  Def. binary Search (A, x):
2.  n = len (A)
3.  beg = 0
4.  end = n - 1
5.  result = -1
6.  While (beg <= end):
7.          mid = (beg + end) / 2
8.          If (A[mid] <= x):
9.              beg = mid + 1
10.             result = mid
11.         Else:
12.             end = mid - 1
13. Return result
```

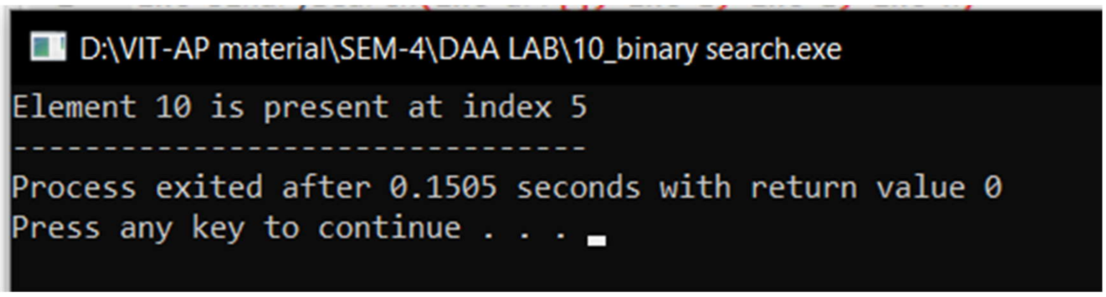
Time Complexity: -

O(n)

Code: -

```
#include <stdio.h>
int binarySearch(int arr[], int l, int i, int x)
{
    int mid;
    if (l = i)
    {
        if (x = arr[i])
        {
            return i;
        }
        else
            return 0;
    }
    else
    {
        mid = ((i + l) / 2);
        if (x = arr[mid])
            return mid;
        else
        {
            if (x < arr[mid])
                return binarySearch(arr, l, mid - 1, x);
            else
                return binarySearch(arr, mid + 1, i, x);
        }
    }
}

int main()
{
    int arr[] = {2, 3, 4, 40, 10, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present in array") :
printf("Element %d is present at index %d", x, result);
    return 0;
}
```

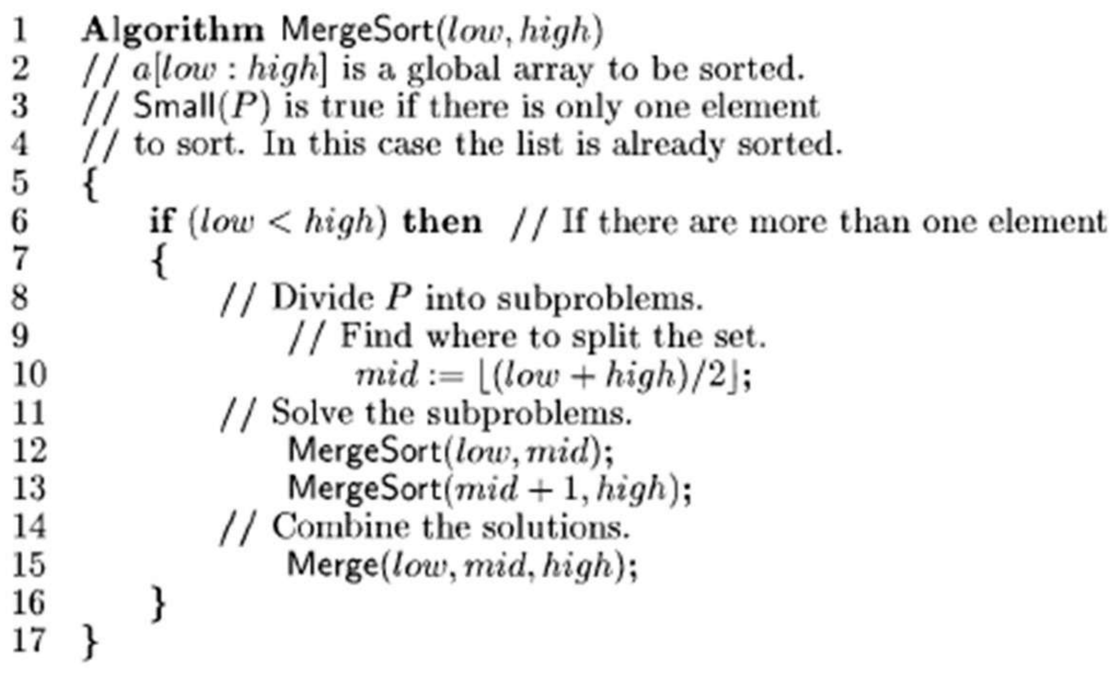
- **Output:**



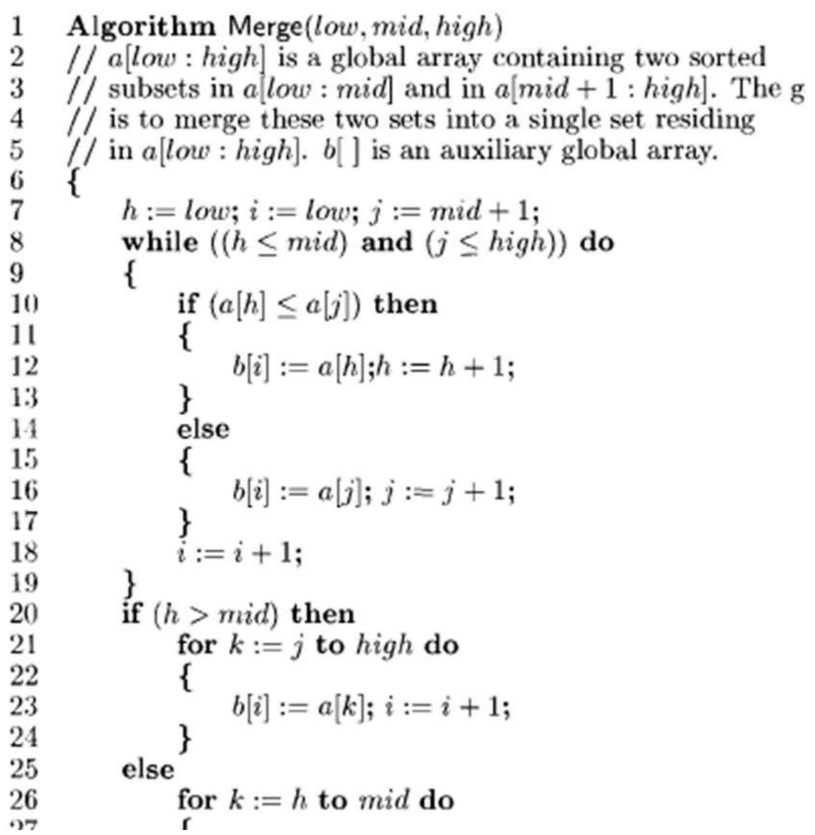
```
D:\VIT-AP material\SEM-4\DAA LAB\10_binary search.exe
Element 10 is present at index 5
--------------------------------
Process exited after 0.1505 seconds with return value 0
Press any key to continue . . .
```

11. Write a c/c++ program to implement Merge Sort.

Algorithm: -



```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```



```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The g
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
```

Time Complexity: -

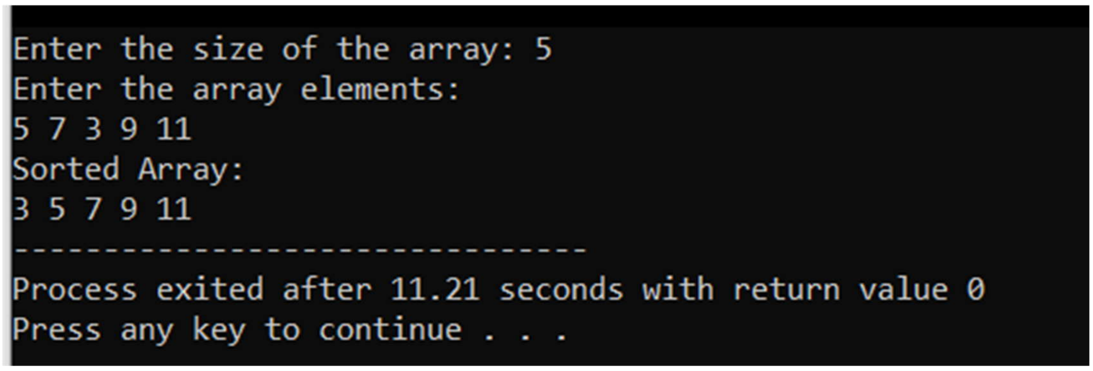
O(nlogn)

Code: -

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

```
#include <stdlib.h>
#define MAX_SIZE 100
int a[MAX_SIZE], b[MAX_SIZE];
void Merge(int low, int mid, int high)
{
    int h = low, i = low, j = mid + 1;
    while ((h <= mid) && (j <= high))
    {
        if (a[h] <= a[j])
        {
            b[i] = a[h];
            h += 1;
        }
        else
        {
            b[i] = b[j];
            j += 1;
        }
        i += 1;
    }
    if (h > mid)
    {
        for (int k = j; k <= high; k++)
        {
            b[i] = a[k];
            i += 1;
        }
    }
    else
    {
        for (int k = h; k <= mid; k++)
        {
            b[i] = a[k];
            i += 1;
        }
    }
    for (int k = low; k <= high; k++)
    {
        a[k] = b[k];
    }
}

void MergeSort(int low, int high)
{
    int mid = 0;
    if (low < high)
    {
        mid = ((low + high) / 2);
        MergeSort(low, mid);
        MergeSort(mid + 1, high);
        Merge(low, mid, high);
    }
}

int main()
{
    int n;
    printf("Enter array size\n");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the array elemnts");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    MergeSort(0, n - 1);
    printf("The sorted array is: \n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t", arr[i]);
    }
    return 0;
}
```

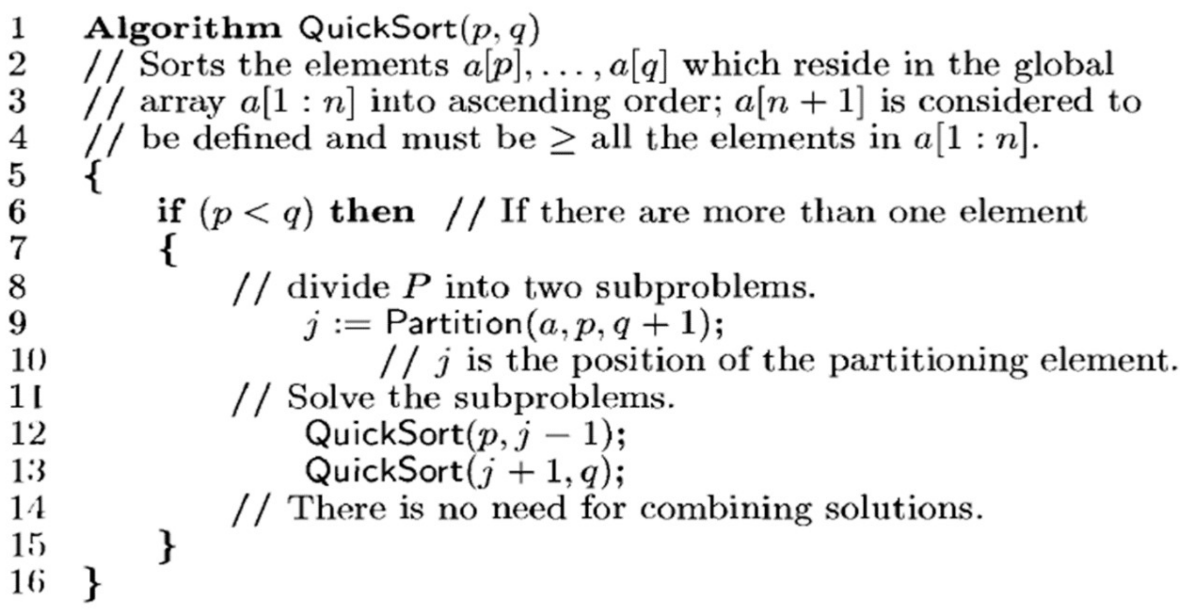
- **Output:**



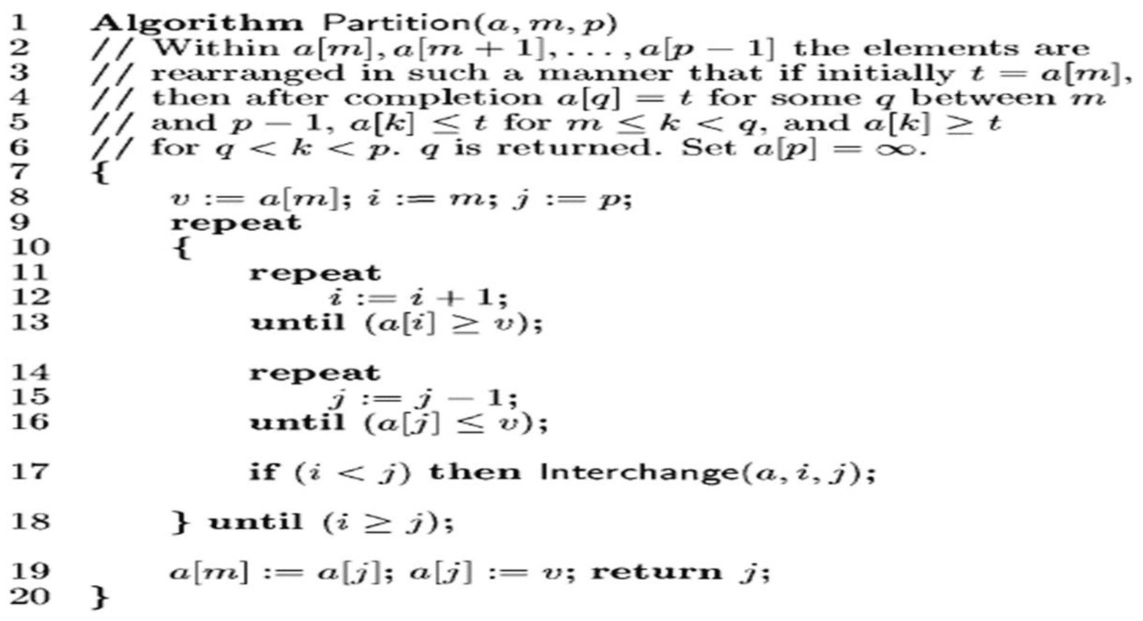
```
Enter the size of the array: 5
Enter the array elements:
5 7 3 9 11
Sorted Array:
3 5 7 9 11
--------------------------------
Process exited after 11.21 seconds with return value 0
Press any key to continue . . .
```

12. Write a c/c++ program to implement Quick Sort.

Algorithm: -



```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if ($p < q$) then // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j :=$ Partition($a, p, q+1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j-1$);
            QuickSort($j+1, q$);
        // There is no need for combining solutions.
    }
}
```



```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \geq v$);

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

        if ($i < j$) then Interchange($a, i, j$);

    } until ($i \geq j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}
```

Time Complexity: -

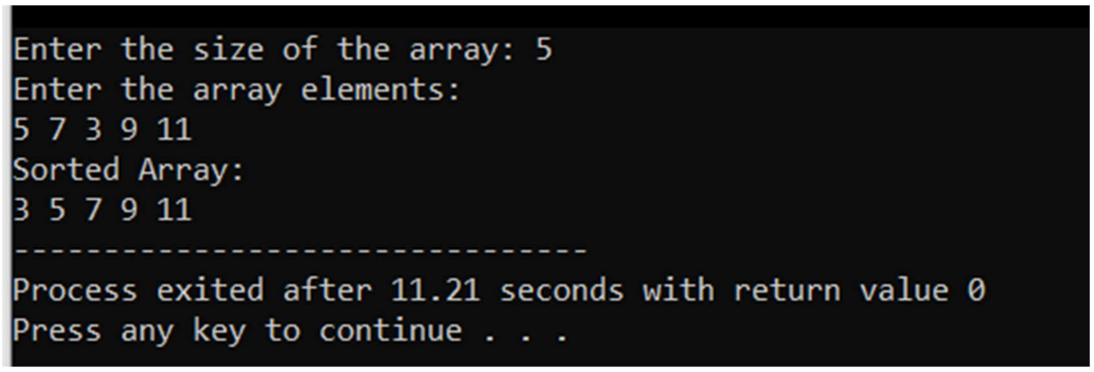
O(nlogn)

Code: -

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
int partition(int arr[], int low, int high) {
    int pivot = arr[high];

    int i = (low - 1);
    for (int j = low; j <= high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

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

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

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

    int arr[n];
    printf("Enter the array elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    quickSort(arr, 0, n - 1);

    printf("Sorted Array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

- **Output:**



```
Enter the size of the array: 5
Enter the array elements:
5 7 3 9 11
Sorted Array:
3 5 7 9 11
--------------------------------
Process exited after 11.21 seconds with return value 0
Press any key to continue . . .
```

13. Write a c/c++ program to implement the Travelling Salesperson Problem(TSP) using Brute Force.

Algorithm: -

```
Algorithm TSP(graph, current, visited, path, cost, min_cost)
{
    If(all vertices visited)
        cost = CalculateTotalCost(path);
    if(cost<min_cost)
    {
        min_cost = cost;
        return;
    }
    for(vertex in graph(
    {
        if (vertex is not visited)
        {
            Add vertex to path;
            Mark vertex as visited;
            TSP(graph, current, visited, path, cost, min_cost);
        }
        Remove last vertex from path;
        Mark vertex a unvisited;
    }
}
```

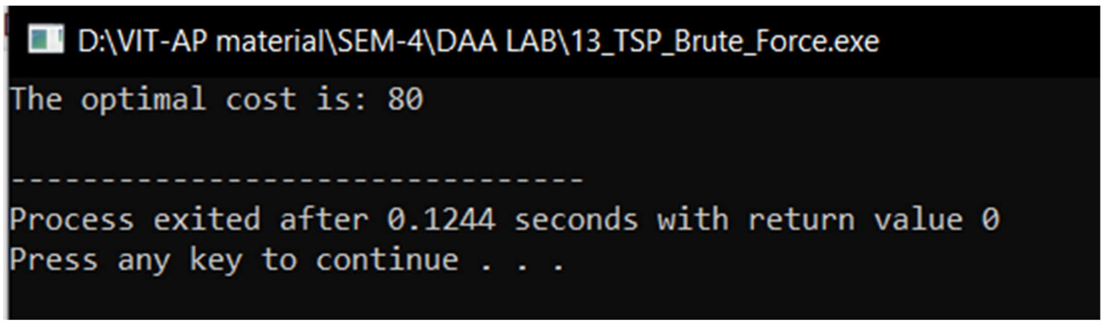
Time Complexity: -

O(2^n*n^2)

Code: -

```
#include <stdio.h>
#include <limits.h>
#define V 4

int next_permutation(int arr[], int size)
{
    int i = size - 1;
    while (i > 0 && arr[i - 1] >= arr[i])
    {
        i--;
    }
    if (i <= 0)
    {
        return 0;
    }
    int j = size - 1;
    while (arr[j] <= arr[i - 1])
    {
        j--;
    }
    int temp = arr[i - 1];
    arr[i - 1] = arr[j];
    arr[j] = temp;
    j = size - 1;
    while (i < j)
    {
        temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        i++;
        j--;
    }
    return 1;
}

int travllingSalesmanProblem(int graph[][V], int s)
{
    int vertex[V - 1];
    for (int i = 0, k = 0; i < V; i++)
    {
        if (i != s)
        {
            vertex[k] = i;
            k++;
        }
    }
    int min_path = INT_MAX;
    do
    {
        int current_pathweight = 0;
        int k = s;
        for (int i = 0; i < V - 1; i++)
        {
            current_pathweight += graph[k][vertex[i]];
            k = vertex[i];
        }
        current_pathweight += graph[k][s];
        if (current_pathweight < min_path)
        {
            min_path = current_pathweight;
        }
    } while (next_permutation(vertex, V - 1));
    return min_path;
}

int main()
{
    int graph[][V] = {{0, 10, 15, 20},
                              {10, 0, 35, 25},
                              {15, 35, 0, 30},
                              {20, 25, 30, 0}};
    int s = 0;
    printf("The optimal cost is: %d\n",
travllingSalesmanProblem(graph, s));
    return 0;
}
```

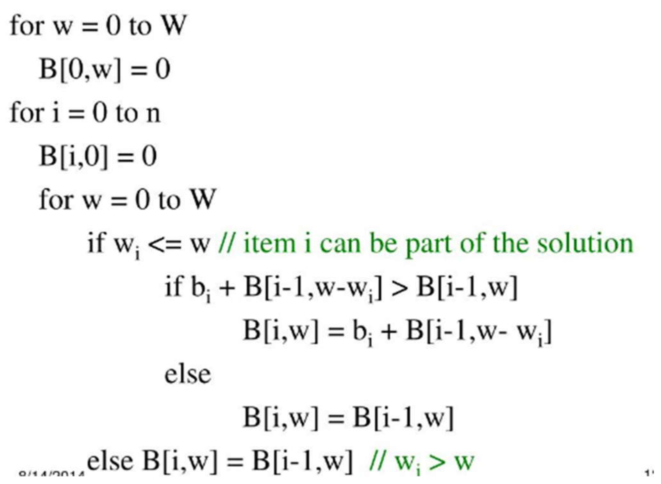
Output: -



```
D:\VIT-AP material\SEM-4\DAA LAB\13_TSP_Brute_Force.exe
The optimal cost is: 80

--------------------------------
Process exited after 0.1244 seconds with return value 0
Press any key to continue . . .
```

14. Write a C/C++ program to Implement the 0/1 Knapsack Problem using Brute Force.

Algorithm: -



```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 0 to W
        if $w_i$ <= w // item i can be part of the solution
                if $b_i$ + B[i-1,w-$w_i$] > B[i-1,w]
                        B[i,w] = $b_i$ + B[i-1,w- $w_i$]
                else
                        B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // $w_i$ > w
```

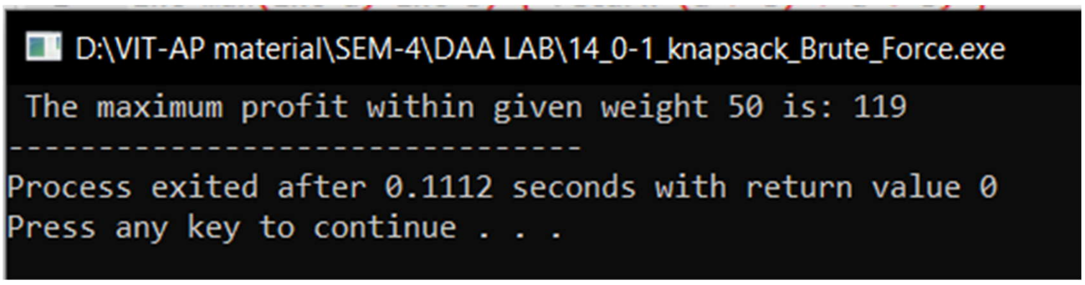
Time Complexity: -

O(2^n)

Code: -

```c
#include <stdio.h>
int max(int a, int b) { return (a > b) ? a : b; }

int knapSack(int W, int wt[], int val[], int n)
{
    if (n == 0 || W == 0)
        return 0;

    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);

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

int main()
{
    int profit[] = {42, 12, 40, 25};
    int weight[] = {7, 3, 4, 5};
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    printf(" The maximum profit within given weight %d is: %d", W,
knapSack(W, weight, profit, n));
    return 0;
}
```

Output: -



D:\VIT-AP material\SEM-4\DAA LAB\14_0-1_knapsack_Brute_Force.exe

```
 The maximum profit within given weight 50 is: 119
--------------------------------
Process exited after 0.1112 seconds with return value 0
Press any key to continue . . .
```

15. Write a C/C++ program to Implement the Job Assignment Problem using Brute Force.

Algorithm: -

```
Algorithm JobAssign(CostMatrix, Cost, Assigned, index)
{
    if(index==N)then//N is the size of 2-D matrix
    {
        if(Cost<min_cost)
        {
            min_cost:=cost;
            for i:=0 to N do
            {
                min_assignment[i]:=assigned[i];
            }
        }
        return;
    }
    for i:=index to N do
    {
        swap(assigned[index],assigned[i]);
        JobAssign(CostMatrix, Cost, Assigned, index);
        swap(assigned[index],assigned[i]);
    }
}
```

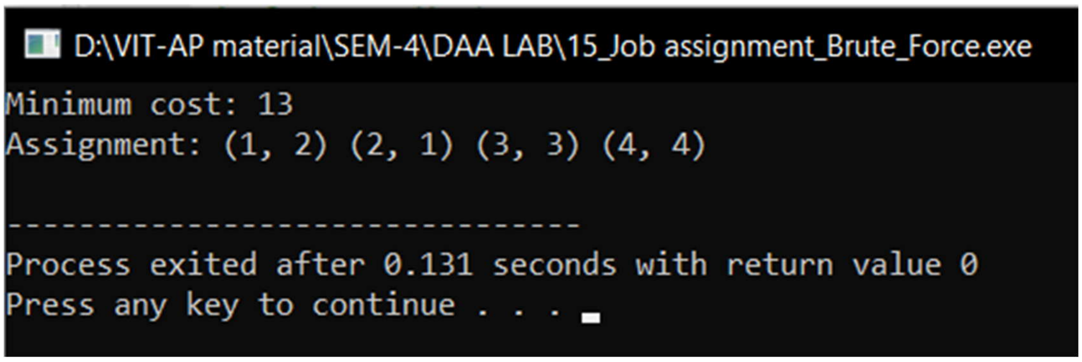
Time Complexity: -

O(n!)

Code: -

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

#define N 4 // Number of workers and jobs

int minCost = INT_MAX;
int minAssignment[N];

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void findMinCost(int costMatrix[N][N], int cost, int assigned[], int
index)
{
    if (index == N)
    {
        if (cost < minCost)
        {
            minCost = cost;
            for (int i = 0; i < N; i++)
            {
                minAssignment[i] = assigned[i];
            }
        }
        return;
    }

    for (int i = index; i < N; i++)
    {
        swap(&assigned[index], &assigned[i]);
        findMinCost(costMatrix, cost + 
costMatrix[index][assigned[index]], assigned, index + 1);
        swap(&assigned[index], &assigned[i]);
    }
}

int main()
{
    int costMatrix[N][N] = {
        {9, 2, 7, 8},
        {6, 4, 3, 7},
        {5, 8, 1, 8},
        {7, 6, 9, 4}};
    int assigned[N];
    for (int i = 0; i < N; i++)
    {
        assigned[i] = i;
    }

    findMinCost(costMatrix, 0, assigned, 0);

    printf("Minimum cost: %d\n", minCost);
    printf("Assignment: ");
    for (int i = 0; i < N; i++)
    {
        printf("(%d, %d) ", i + 1, minAssignment[i] + 1);
    }
    printf("\n");

    return 0;
}
```

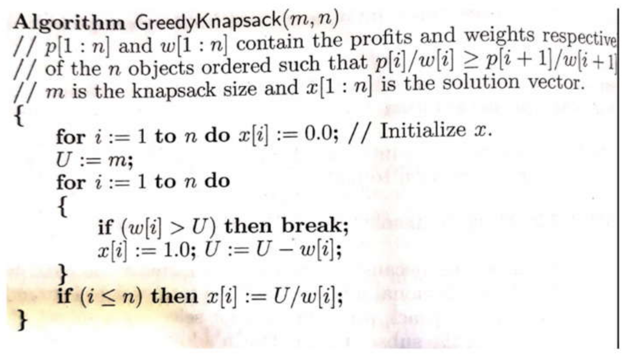
Output: -


D:\VIT-AP material\SEM-4\DAA LAB\15_Job assignment_Brute_Force.exe

```
Minimum cost: 13
Assignment: (1, 2) (2, 1) (3, 3) (4, 4)

--------------------------------
Process exited after 0.131 seconds with return value 0
Press any key to continue . . .
```

16. Write a C/C++ program to Implement the Frac▯on Knapsack Problem using Greedy Method.

Algorithm: -



```
Algorithm GreedyKnapsack(m, n)
// p[1 : n] and w[1 : n] contain the profits and weights respective
// of the n objects ordered such that p[i]/w[i] ≥ p[i + 1]/w[i + 1]
// m is the knapsack size and x[1 : n] is the solution vector.
{
    for i := 1 to n do x[i] := 0.0; // Initialize x.
    U := m;
    for i := 1 to n do
    {
        if (w[i] > U) then break;
        x[i] := 1.0; U := U − w[i];
    }
    if (i ≤ n) then x[i] := U/w[i];
}
```

Time Complexity: -

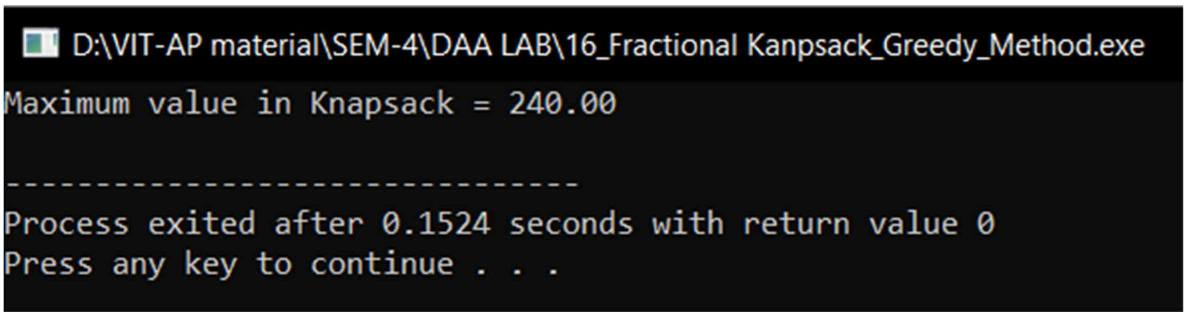
O(2^n)

Code: -

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

typedef struct
{
    int weight;
    int value;
    float ratio;
} Item;

void swap(Item *a, Item *b)
{
    Item temp = *a;
    *a = *b;
    *b = temp;
}

void sortItemsByRatio(Item items[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (items[j].ratio < items[j + 1].ratio)
            {
                swap(&items[j], &items[j + 1]);
            }
        }
    }
}
```

```c
float fractionalKnapsack(int capacity, Item items[], int n)
{
    float totalValue = 0.0;
    int currentWeight = 0;

    sortItemsByRatio(items, n);

    for (int i = 0; i < n; i++)
    {
        if (currentWeight + items[i].weight <= capacity)
        {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        }
        else
        {
            int remainingWeight = capacity - currentWeight;
            totalValue += items[i].ratio * remainingWeight;
            break;
        }
    }

    return totalValue;
}

int main()
{
    int capacity = 50;
    Item items[] = {
        {10, 60, 0.0},
        {20, 100, 0.0},
        {30, 120, 0.0}};
    int n = sizeof(items) / sizeof(items[0]);

    for (int i = 0; i < n; i++)
    {
        items[i].ratio = (float)items[i].value / items[i].weight;
    }

    float totalValue = fractionalKnapsack(capacity, items, n);
    printf("Maximum value in Knapsack = %.2f\n", totalValue);

    return 0;
}
```

Output: -



D:\VIT-AP material\SEM-4\DAA LAB\16_Fractional Kanpsack_Greedy_Method.exe

```
Maximum value in Knapsack = 240.00

--------------------------------
Process exited after 0.1524 seconds with return value 0
Press any key to continue . . .
```

17. Write a C/C++ program to Implement the Job Sequencing with deadlines Problem using the Greedy Method.

Algorithm: -

```
Algorithm JS($d, j, n$)
// $d[i] \geq 1$, $1 \leq i \leq n$ are the deadlines, $n \geq 1$. The jobs
// are ordered such that $p[1] \geq p[2] \geq \cdots \geq p[n]$. $J[i]$
// is the $i$th job in the optimal solution, $1 \leq i \leq k$.
// Also, at termination $d[J[i]] \leq d[J[i+1]]$, $1 \leq i < k$.
{
    $d[0] := J[0] := 0$; // Initialize.
    $J[1] := 1$; // Include job 1.
    $k := 1$;
    for $i := 2$ to $n$ do
    {
        // Consider jobs in nonincreasing order of $p[i]$. Find
        // position for $i$ and check feasibility of insertion.
        $r := k$;
        while $((d[J[r]] > d[i])$ and $(d[J[r]] \neq r))$ do $r := r - 1$;
        if $((d[J[r]] \leq d[i])$ and $(d[i] > r))$ then
        {
            // Insert $i$ into $J[\,]$.
            for $q := k$ to $(r+1)$ step $-1$ do $J[q+1] := J[q]$;
            $J[r+1] := i$; $k := k + 1$;
        }
    }
    return $k$;
}
```

Time Complexity: -

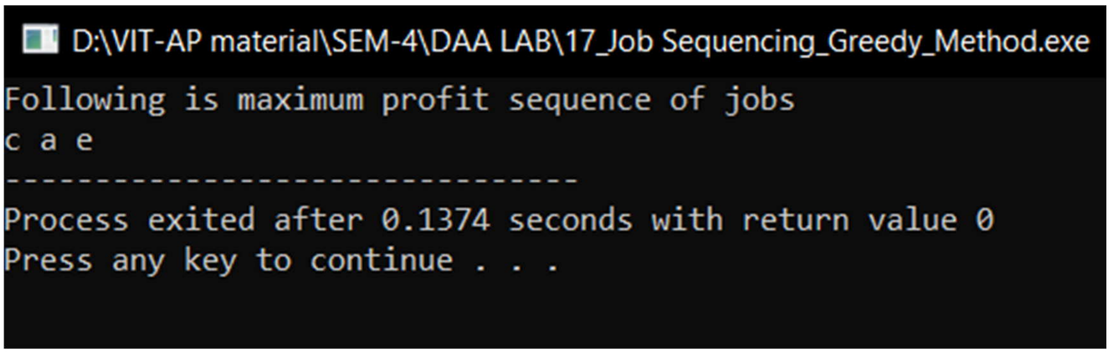
O(n^2)

Code: -

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

typedef struct Job
{

    char id;
    int dead;
    int profit;
} Job;

int compare(const void *a, const void *b)
{
    Job *temp1 = (Job *)a;
    Job *temp2 = (Job *)b;
    return (temp2->profit - temp1->profit);
}

int min(int num1, int num2)
{
    return (num1 > num2) ? num2 : num1;
}
```

```c
void printJobScheduling(Job arr[], int n)
{
    qsort(arr, n, sizeof(Job), compare);

    int result[n];
    bool slot[n];

    for (int i = 0; i < n; i++)
        slot[i] = false;

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

        for (int j = min(n, arr[i].dead) - 1; j >= 0; j--)
        {

            if (slot[j] == false)
            {
                result[j] = i;
                slot[j] = true;
                break;
            }
        }
    }

    for (int i = 0; i < n; i++)
        if (slot[i])
            printf("%c ", arr[result[i]].id);
}

int main()
{
    Job arr[] = {{'a', 2, 100},
                 {'b', 1, 19},
                 {'c', 2, 27},
                 {'d', 1, 25},
                 {'e', 3, 15}};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf(
        "Following is maximum profit sequence of jobs \n");
    printJobScheduling(arr, n);
    return 0;
}
```

Output: -



```
D:\VIT-AP material\SEM-4\DAA LAB\17_Job Sequencing_Greedy_Method.exe
Following is maximum profit sequence of jobs
c a e
--------------------------------
Process exited after 0.1374 seconds with return value 0
Press any key to continue . . .
```

18. Write a C/C++ program to Implement the Single Source Shortest Path (Dijkstra's Algorithm) Problem using the Greedy Method.

Algorithm: -

```
Algorithm ShortestPaths($v, cost, dist, n$)
// $dist[j], 1 \le j \le n$, is set to the length of the shortest
// path from vertex $v$ to vertex $j$ in a digraph $G$ with $n$
// vertices. $dist[v]$ is set to zero. $G$ is represented by its
// cost adjacency matrix $cost[1:n, 1:n]$.
{
    for $i := 1$ to $n$ do
    { // Initialize $S$.
        $S[i]$ := false; $dist[i] := cost[v, i]$;
    }
    $S[v]$ := true; $dist[v] := 0.0$; // Put $v$ in $S$.
    for $num := 2$ to $n - 1$ do
    {
        // Determine $n - 1$ paths from $v$.
        Choose $u$ from among those vertices not
        in $S$ such that $dist[u]$ is minimum;
        $S[u]$ := true; // Put $u$ in $S$.
        for (each $w$ adjacent to $u$ with $S[w]$ = false) do
            // Update distances.
            if ($dist[w] > dist[u] + cost[u, w]$) then
                    $dist[w] := dist[u] + cost[u, w]$;
    }
}
```

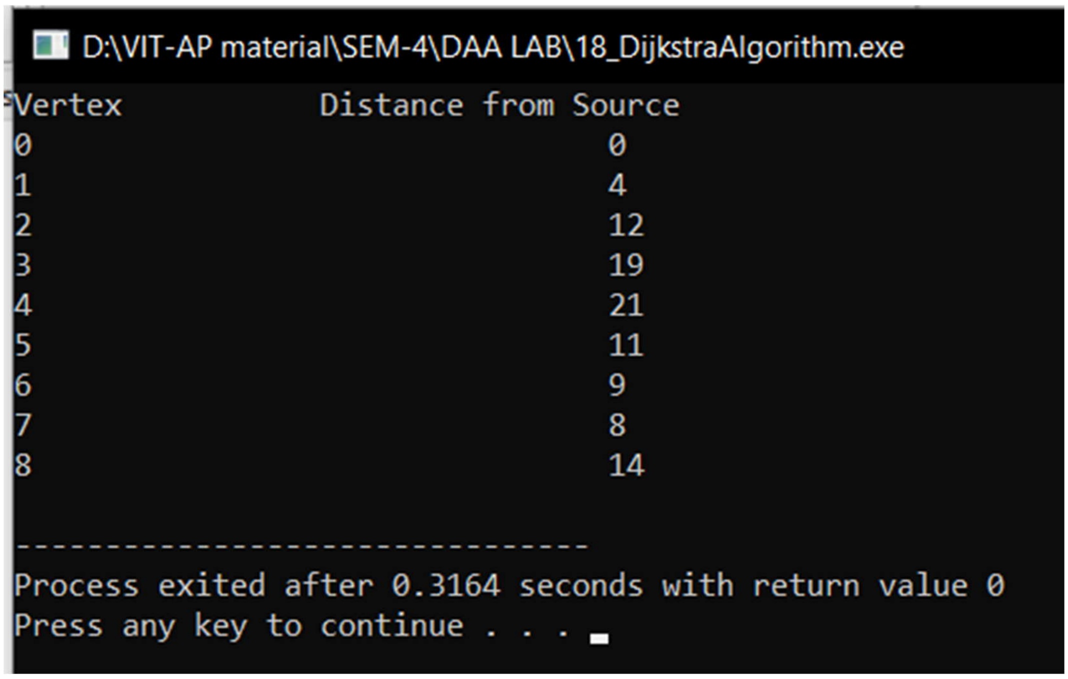
Time Complexity: -

$O(n^2)$

Code: -

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

#define V 9

int minDistance(int dist[], bool sptSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

void printSolution(int dist[])
{
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t\t\t %d\n", i, dist[i]);
}

void dijkstra(int graph[V][V], int src)
{
    int dist[V];

    bool sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    dist[src] = 0;

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

        int u = minDistance(dist, sptSet);

        sptSet[u] = true;

        for (int v = 0; v < V; 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];
    }

    printSolution(dist);
}

int main()
{
    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, 14, 10, 0, 2, 0, 0},
                       {0, 0, 0, 0, 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;
}
```

Output: -



D:\VIT-AP material\SEM-4\DAA LAB\18_DijkstraAlgorithm.exe

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

--------------------------------
Process exited after 0.3164 seconds with return value 0
Press any key to continue . . .
```

19. Write a C/C++ program to Implement Prim's Algorithm for construction of a minimum cost-spanning tree using the Greedy Methodology

Algorithm: -

```
Algorithm Prim($E, cost, n, t$)
// $E$ is the set of edges in $G$. $cost[1:n, 1:n]$ is the cost
// adjacency matrix of an $n$ vertex graph such that $cost[i,j]$ is
// either a positive real number or $\infty$ if no edge $(i,j)$ exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array $t[1:n-1, 1:2]$. $(t[i,1], t[i,2])$ is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let $(k,l)$ be an edge of minimum cost in $E$;
    $mincost := cost[k,l]$;
    $t[1,1] := k$; $t[1,2] := l$;
    for $i := 1$ to $n$ do  // Initialize near.
        if $(cost[i,l] < cost[i,k])$ then $near[i] := l$;
        else $near[i] := k$;
    $near[k] := near[l] := 0$;
    for $i := 2$ to $n-1$ do
    { // Find $n-2$ additional edges for $t$.
        Let $j$ be an index such that $near[j] \neq 0$ and
        $cost[j, near[j]]$ is minimum;
        $t[i,1] := j$; $t[i,2] := near[j]$;
        $mincost := mincost + cost[j, near[j]]$;
        $near[j] := 0$;
        for $k := 1$ to $n$ do // Update $near[\,]$.
            if $((near[k] \neq 0)$ and $(cost[k, near[k]] > cost[k,j]))$
                then $near[k] := j$;
    }
    return $mincost$;
}
```

Time Complexity: -

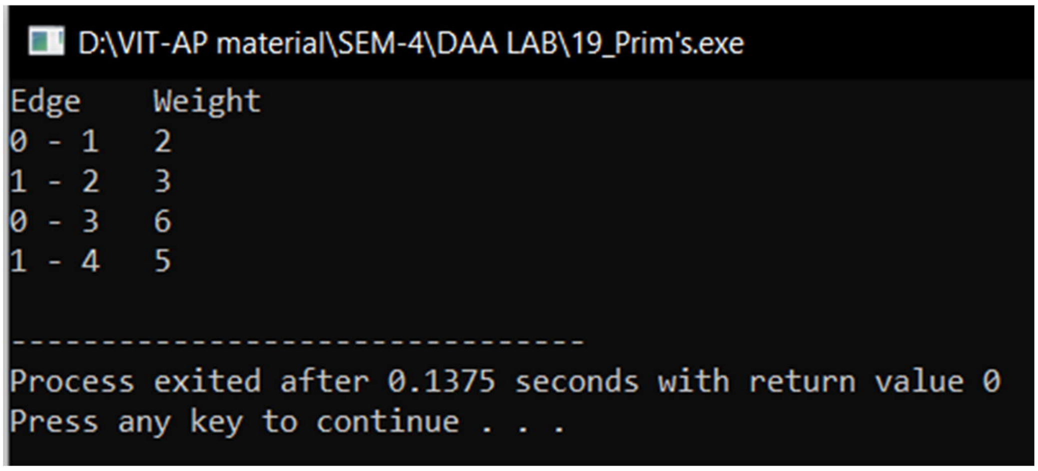
O(n^2)

Code: -

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

#define V 5

int minKey(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

int printMST(int parent[], int graph[V][V])
{
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i,
               graph[i][parent[i]]);
}
```

```
void primMST(int graph[V][V])
{
    int parent[V];

    int key[V];

    bool mstSet[V];

    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    key[0] = 0;

    parent[0] = -1;

    for (int count = 0; count < V - 1; count++)
    {
        int u = minKey(key, mstSet);

        mstSet[u] = true;

        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] <
key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    printMST(parent, graph);
}

int main()
{
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0}};

    primMST(graph);

    return 0;
}
```

Output: -



D:\VIT-AP material\SEM-4\DAA LAB\19_Prim's.exe

```
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5

--------------------------------
Process exited after 0.1375 seconds with return value 0
Press any key to continue . . .
```

20. Write a C/C++ program to Implement the Kruskal’s Algorithm for the construction of a minimum cost-spanning tree using the Greedy Methodology.

Algorithm: -

```
Algorithm Kruskal($E, cost, n, t$)
// $E$ is the set of edges in $G$. $G$ has $n$ vertices. $cost[u, v]$ is the
// cost of edge $(u, v)$. $t$ is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for $i := 1$ to $n$ do $parent[i] := -1$;
    // Each vertex is in a different set.
    $i := 0$; $mincost := 0.0$;
    while (($i < n - 1$) and (heap not empty)) do
    {
        Delete a minimum cost edge $(u, v)$ from the heap
        and reheapify using Adjust;
        $j :=$ Find($u$); $k :=$ Find($v$);
        if ($j \neq k$) then
        {
            $i := i + 1$;
            $t[i, 1] := u$; $t[i, 2] := v$;
            $mincost := mincost + cost[u, v]$;
            Union($j, k$);
        }
    }
    if ($i \neq n - 1$) then write ("No spanning tree");
    else return $mincost$;
}
```

```
Algorithm Adjust($a, i, n$)
// The complete binary trees with roots $2i$ and $2i + 1$ are
// combined with node $i$ to form a heap rooted at $i$. No
// node has an address greater than $n$ or less than 1.
{
    $j := 2i$; $item := a[i]$;
    while ($j \leq n$) do
    {
        if (($j < n$) and ($a[j] < a[j + 1]$)) then $j := j + 1$;
            // Compare left and right child
            // and let $j$ be the larger child.
        if ($item \geq a[j]$) then break;
            // A position for $item$ is found.
        $a[\lfloor j/2 \rfloor] := a[j]$; $j := 2j$;
    }
    $a[\lfloor j/2 \rfloor] := item$;
}
```

```
Algorithm Heapify($a, n$)
// Readjust the elements in $a[1 : n]$ to form a heap.
{
    for $i := \lfloor n/2 \rfloor$ to 1 step $-1$ do Adjust($a, i, n$);
}
```

```
Algorithm SimpleUnion($i, j$)
{
    $p[i] := j$;
}
```

```
Algorithm SimpleFind($i$)
{
    while ($p[i] \geq 0$) do $i := p[i]$;
    return $i$;
}
```

Time Complexity: -

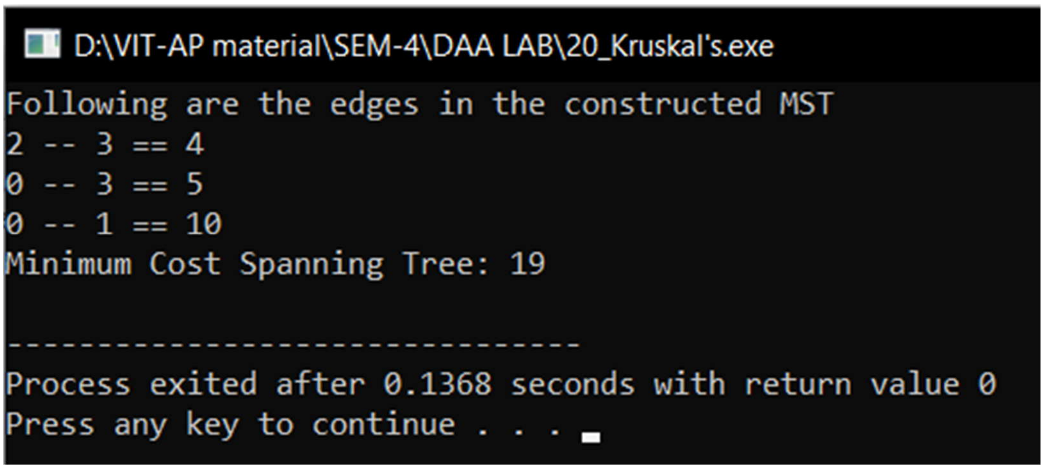
O(E*logE)

Code: -

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

int comparator(const void *p1, const void *p2)
{
    const int(*x)[3] = p1;
    const int(*y)[3] = p2;

    return (*x)[2] - (*y)[2];
}

void makeSet(int parent[], int rank[], int n)
{
    for (int i = 0; i < n; i++)
    {
        parent[i] = i;
        rank[i] = 0;
    }
}

int findParent(int parent[], int component)
{
    if (parent[component] == component)
        return component;

    return parent[component] = findParent(parent, parent[component]);
}

void unionSet(int u, int v, int parent[], int rank[], int n)
{

    u = findParent(parent, u);
    v = findParent(parent, v);

    if (rank[u] < rank[v])
    {
        parent[u] = v;
    }
    else if (rank[u] > rank[v])
    {
        parent[v] = u;
    }
    else
    {
        parent[v] = u;

        rank[u]++;
    }
}
```

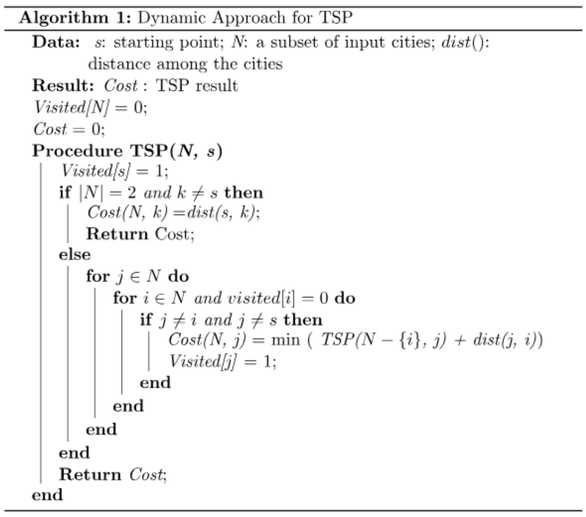
```c
void kruskalAlgo(int n, int edge[n][3])
{

    qsort(edge, n, sizeof(edge[0]), comparator);

    int parent[n];
    int rank[n];

    makeSet(parent, rank, n);

    int minCost = 0;

    printf(
        "Following are the edges in the constructed MST\n");
    for (int i = 0; i < n; i++)
    {
        int v1 = findParent(parent, edge[i][0]);
        int v2 = findParent(parent, edge[i][1]);
        int wt = edge[i][2];

        if (v1 != v2)
        {
            unionSet(v1, v2, parent, rank, n);
            minCost += wt;
            printf("%d -- %d == %d\n", edge[i][0],
                   edge[i][1], wt);
        }
    }

    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

int main()
{
    int edge[5][3] = {{0, 1, 10},
                      {0, 2, 6},
                      {0, 3, 5},
                      {1, 3, 15},
                      {2, 3, 4}};

    kruskalAlgo(5, edge);

    return 0;
}
```

Output: -


D:\VIT-AP material\SEM-4\DAA LAB\20_Kruskal's.exe

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19

--------------------------------
Process exited after 0.1368 seconds with return value 0
Press any key to continue . . .
```

21. Write a C/C++ program to Implement the Travelling Salesperson (TSP) Problem using Dynamic Programming.
Algorithm: -



```
Algorithm 1: Dynamic Approach for TSP
Data: $s$: starting point; $N$: a subset of input cities; $dist()$:
      distance among the cities
Result: $Cost$ : TSP result
$Visited[N] = 0$;
$Cost = 0$;
Procedure TSP($N$, $s$)
    $Visited[s] = 1$;
    if $|N| = 2$ and $k \neq s$ then
        $Cost(N, k) = dist(s, k)$;
        Return Cost;
    else
        for $j \in N$ do
            for $i \in N$ and $visited[i] = 0$ do
                if $j \neq i$ and $j \neq s$ then
                    $Cost(N, j) = \min(\ TSP(N - \{i\}, j) + dist(j, i))$
                    $Visited[j] = 1$;
                end
            end
        end
    end
    Return $Cost$;
end
```

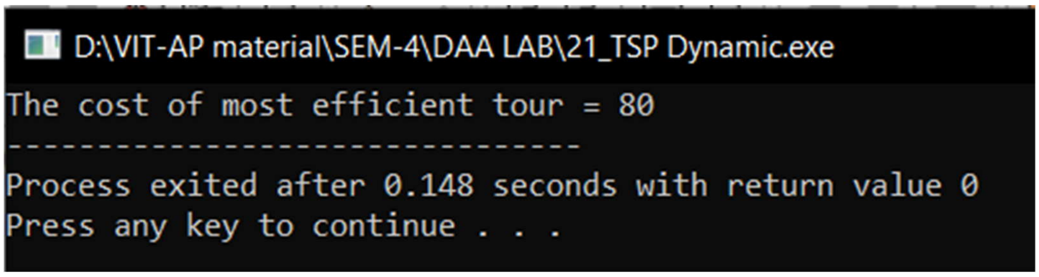
Time Complexity: -
$O(n^2 * 2^n)$

Code: -

```
#include <stdio.h>

#define n 4
#define MAX 10000

int dist[n + 1][n + 1] = {
    {0, 0, 0, 0, 0},
    {0, 0, 10, 15, 20},
    {0, 10, 0, 25, 25},
    {0, 15, 25, 0, 30},
    {0, 20, 25, 30, 0},
};

int memo[n + 1][1 << (n + 1)];

int min(int a, int b) { return a < b ? a : b; }

int fun(int i, int mask)
{
    if (mask == ((1 << i) | 3))
        return dist[1][i];

    if (memo[i][mask] != 0)
        return memo[i][mask];

    int res = MAX;
```

```
        for (int j = 1; j <= n; j++)
            if ((mask & (1 << j)) && j != i && j != 1)
                res = min(res, fun(j, mask & (~(1 << i))) + dist[j][i]);
        return memo[i][mask] = res;
    }

    int main()
    {
        int ans = MAX;
        for (int i = 1; i <= n; i++)
            ans = min(ans, fun(i, (1 << (n + 1)) - 1) + dist[i][1]);

        printf("The cost of most efficient tour = %d", ans);

        return 0;
    }
```

Output: -



```
D:\VIT-AP material\SEM-4\DAA LAB\21_TSP Dynamic.exe
The cost of most efficient tour = 80
--------------------------------
Process exited after 0.148 seconds with return value 0
Press any key to continue . . .
```

22. Write a C/C++ program to Implement the All Pairs Shortes Path (Floyd's-Warshall Algorithm) Problem using Dynamic Programming.

Algorithm: -

```
Algorithm AllPaths($cost, A, n$)
// $cost[1:n, 1:n]$ is the cost adjacency matrix of a graph with
// $n$ vertices; $A[i,j]$ is the cost of a shortest path from vertex
// $i$ to vertex $j$. $cost[i,i] = 0.0$, for $1 \le i \le n$.
{
    for $i := 1$ to $n$ do
        for $j := 1$ to $n$ do
            $A[i,j] := cost[i,j]$; // Copy $cost$ into $A$.
    for $k := 1$ to $n$ do
        for $i := 1$ to $n$ do
            for $j := 1$ to $n$ do
                $A[i,j] := \min(A[i,j], A[i,k] + A[k,j])$;
}
```

Time Complexity: -

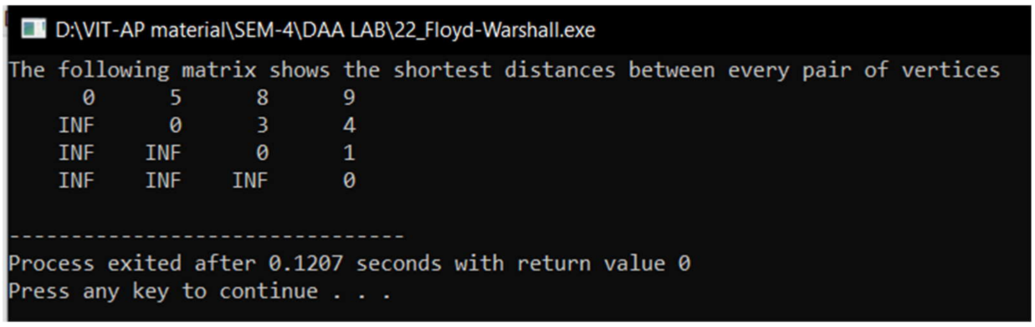
$O(n^3)$

Code: -

```
#include <stdio.h>

#define V 4
#define INF 99999

void printSolution(int dist[][V])
{
    printf("The following matrix shows the shortest distances"
           " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

void floydWarshall(int dist[][V])
{
    int i, j, k;
    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
```

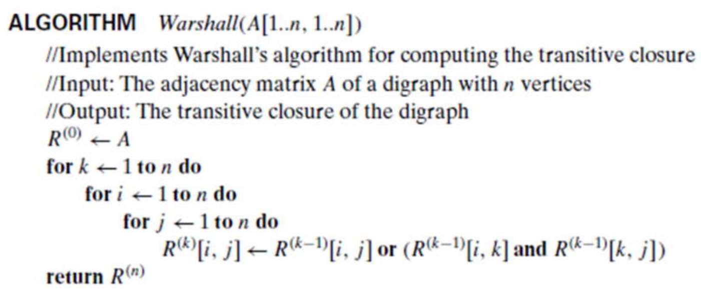
```
                {
                    if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }

        printSolution(dist);
    }

    int main()
    {
        /* Let us create the following weighted graph
                10
        (0)------->(3)
         |         /|\
        5|          |
         |          | 1
        \|/         |
        (1)------->(2)
                 3       */
        int graph[V][V] = {{0, 5, INF, 10},
                           {INF, 0, 3, INF},
                           {INF, INF, 0, 1},
                           {INF, INF, INF, 0}};

        floydWarshall(graph);
        return 0;
    }
```

Output: -



```
D:\VIT-AP material\SEM-4\DAA LAB\22_Floyd-Warshall.exe
The following matrix shows the shortest distances between every pair of vertices
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0

--------------------------------
Process exited after 0.1207 seconds with return value 0
Press any key to continue . . .
```

23. Write a C/C++ program to Implement theWarshall’s Algorithm (Transitive Closure).
Algorithm: -



```
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 the digraph
    $R^{(0)} \leftarrow A$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

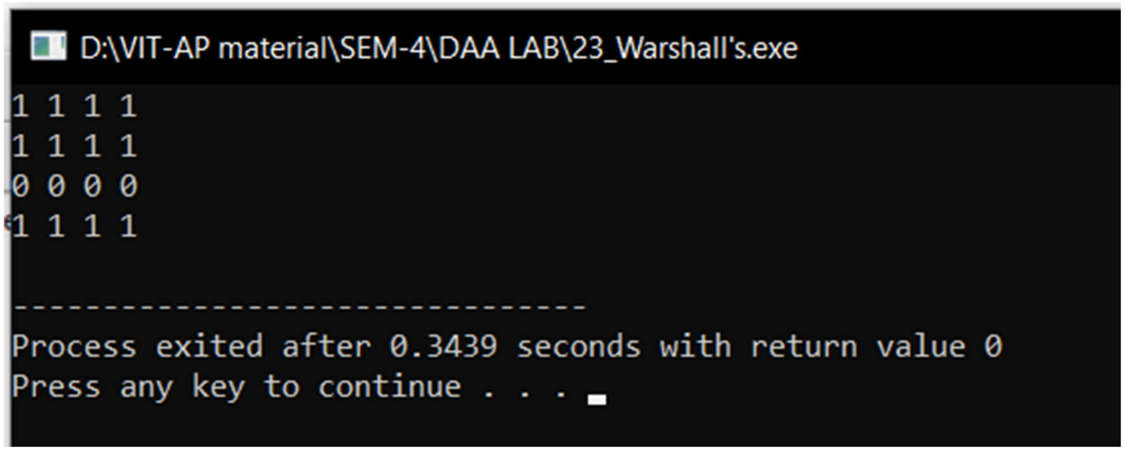
Time Complexity: -
O(n^3)

Code: -

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

#define V 4 // Number of vertices in the graph

void printMatrix(int reach[][V]) {
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << reach[i][j] << " ";
        }
        cout << endl;
    }
}

void transitiveClosure(int graph[][V]) {
    int reach[V][V];

    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            reach[i][j] = graph[i][j];

    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                reach[i][j] = reach[i][j] || (reach[i][k] &&
reach[k][j]);
            }
        }
    }

    printMatrix(reach);
}
```

```
int main() {
    int graph[V][V] = { {0, 1, 0, 0},
                        {0, 0, 0, 1},
                        {0, 0, 0, 0},
                        {1, 0, 1, 0} };

    transitiveClosure(graph);
    return 0;
}
```

Output: -



D:\VIT-AP material\SEM-4\DAA LAB\23_Warshall's.exe

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

--------------------------------
Process exited after 0.3439 seconds with return value 0
Press any key to continue . . .
```

24. Write a C/C++ program that uses Dynamic Programming Algorithm to solve the Optimal Binary Search Tree Problem.

Algorithm: -

```
Algorithm optCost(freq, i, j){
    if (j < i)then
        return 0;
    if (j == i) then
        return freq[i];

    int fsum:= sum(freq, i, j);

    int min:= INT_MAX;
    for r:= i to j do
    {
        int cost:= optCost(freq, i, r-1) + optCost(freq, r+1, j);
        if (cost < min)
            min:= cost;
    }
    return min + fsum;
}
```

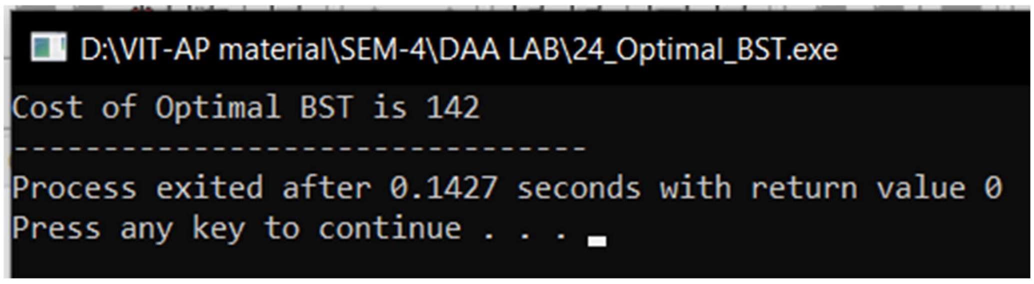
Time Complexity: -

O(n^3)

Code: -

```
#include <stdio.h>
#include <limits.>
#define INT_MAX 100
int sum(int freq[], int i, int j)
{
    int s = 0;
    int k;
    for (k = i; k <=j; k++)
    s += freq[k];
    return s;
}

int optCost(int freq[], int i, int j){
    if (j < i)
        return 0;
    if (j == i)
        return freq[i];

    int fsum = sum(freq, i, j);

    int min = INT_MAX;
    int r;
    for (r = i; r <= j; ++r)
    {
        int cost = optCost(freq, i, r-1) +
                   optCost(freq, r+1, j);
        if (cost < min)
            min = cost;
    }

    return min + fsum;
}

int optimalSearchTree(int keys[], int freq[], int n)
{
    return optCost(freq, 0, n-1);
}

int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ",
            optimalSearchTree(keys, freq, n));
    return 0;
}
```

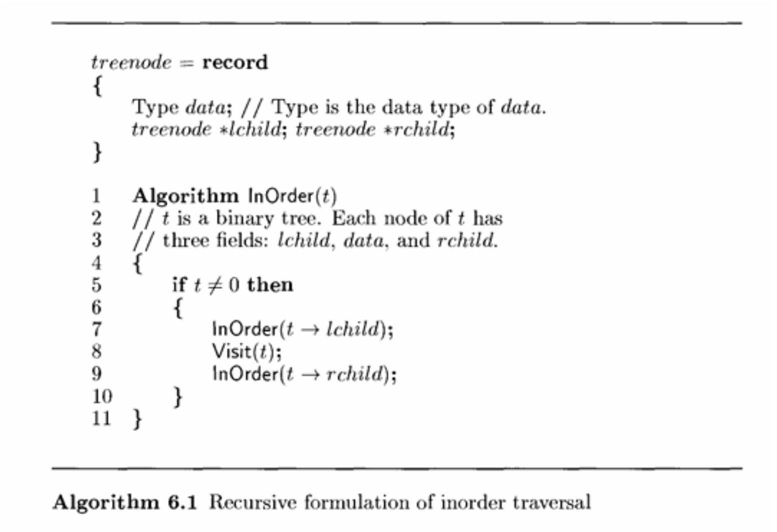
Output: -



```
D:\VIT-AP material\SEM-4\DAA LAB\24_Optimal_BST.exe
Cost of Optimal BST is 142
--------------------------------
Process exited after 0.1427 seconds with return value 0
Press any key to continue . . .
```

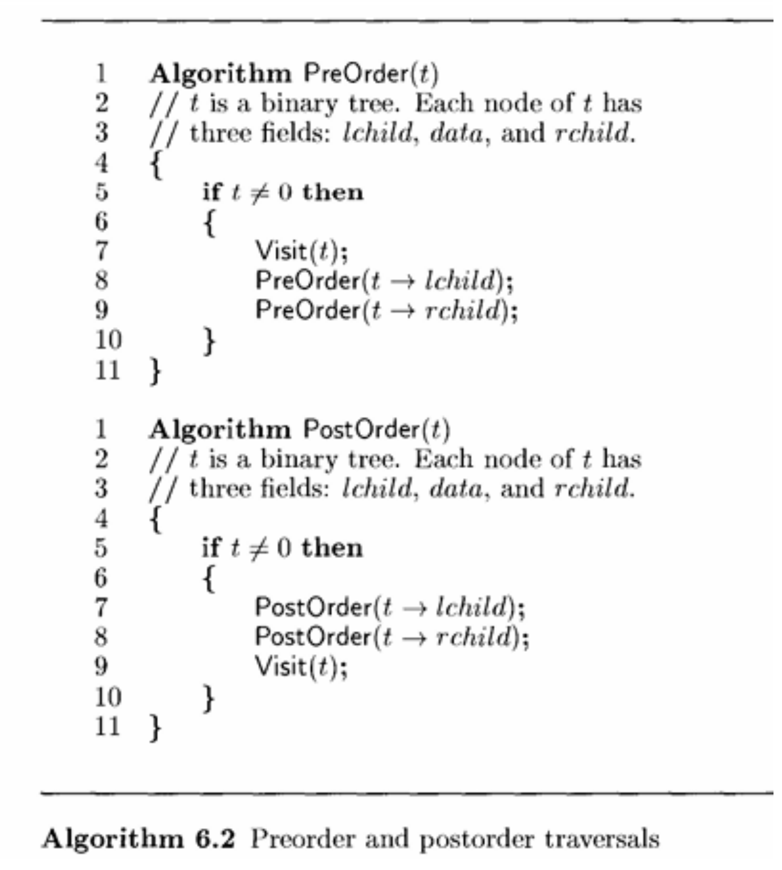
25. Write a C/C++ program to Implement the Tree Traversals.

Algorithm: -



```
treenode = record
{
    Type data; // Type is the data type of data.
    treenode *lchild; treenode *rchild;
}

Algorithm InOrder(t)
// t is a binary tree. Each node of t has
// three fields: lchild, data, and rchild.
{
    if t ≠ 0 then
    {
        InOrder(t → lchild);
        Visit(t);
        InOrder(t → rchild);
    }
}
```

Algorithm 6.1 Recursive formulation of inorder traversal



```
Algorithm PreOrder(t)
// t is a binary tree. Each node of t has
// three fields: lchild, data, and rchild.
{
    if t ≠ 0 then
    {
        Visit(t);
        PreOrder(t → lchild);
        PreOrder(t → rchild);
    }
}

Algorithm PostOrder(t)
// t is a binary tree. Each node of t has
// three fields: lchild, data, and rchild.
{
    if t ≠ 0 then
    {
        PostOrder(t → lchild);
        PostOrder(t → rchild);
        Visit(t);
    }
}
```

Algorithm 6.2 Preorder and postorder traversals

Time Complexity: -

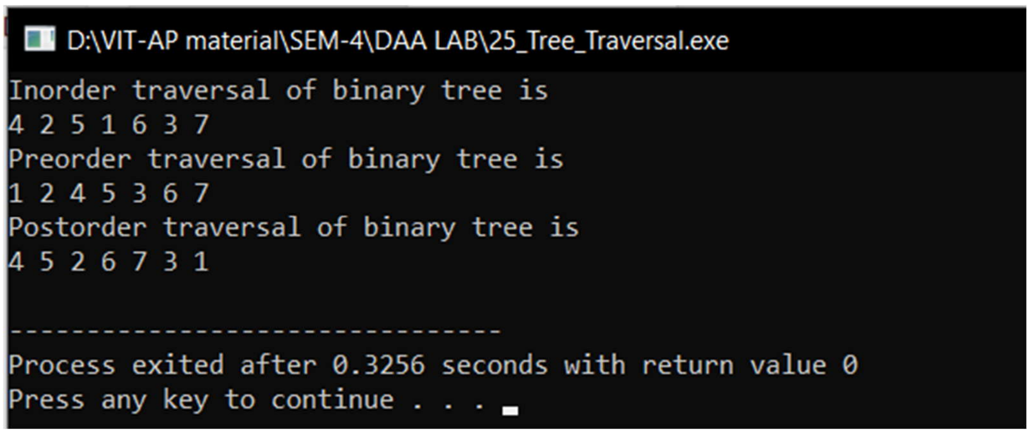
O(n)

Code: -

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

// A binary tree node has data, pointer to left child
// and a pointer to right child
struct Node {
    int data;
    struct Node *left, *right;
};

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Given a binary tree, print its nodes in inorder
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    // First recur on left child
    printInorder(node->left);

    // Then print the data of node
    cout << node->data << " ";

    // Now recur on right child
    printInorder(node->right);
}

// Given a binary tree, print its nodes in preorder
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    // First print data of node
    cout << node->data << " ";

    // Then recur on left subtree
    printPreorder(node->left);

    // Now recur on right subtree
    printPreorder(node->right);
}

// Given a binary tree, print its nodes according to the
// "bottom-up" postorder traversal.
void printPostorder(struct Node* node)
```

```
{
    if (node == NULL)
        return;

    // First recur on left subtree
    printPostorder(node->left);

    // Then recur on right subtree
    printPostorder(node->right);

    // Now deal with the node
    cout << node->data << " ";
}

// Driver code
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    // Function call
    cout << "Inorder traversal of binary tree is \n";
    printInorder(root);
    cout<<endl;
    cout << "Preorder traversal of binary tree is \n";
    printPreorder(root);
    cout<<endl;
    cout << "Postorder traversal of binary tree is \n";
    printPostorder(root);
    cout<<endl;

    return 0;
}
```

Output: -


D:\VIT-AP material\SEM-4\DAA LAB\25_Tree_Traversal.exe

```
Inorder traversal of binary tree is
4 2 5 1 6 3 7
Preorder traversal of binary tree is
1 2 4 5 3 6 7
Postorder traversal of binary tree is
4 5 2 6 7 3 1

--------------------------------
Process exited after 0.3256 seconds with return value 0
Press any key to continue . . .
```

26. Write a C/C++ program to Implement the Topological Sorting.

Algorithm: -

```
topologicalSort()

For(vertex=0; vertex<inputSize; vertex++)

  Find indegree[vertex]

while(node with in-degree zero exists)

{

Find vertex U with in-degree = 0

Remove U and all its edges (U, V) from the graph.

For vertices where edges connected to them were removed.

    in-degree[vertex]=in-degree[vertex]-1

}

if(elements sorted = all elements)

    Return or Print nodes in topologically sorted order

Else

    Return null or Print no topological ordering exists

end topologicalSort()
```

Time Complexity: -

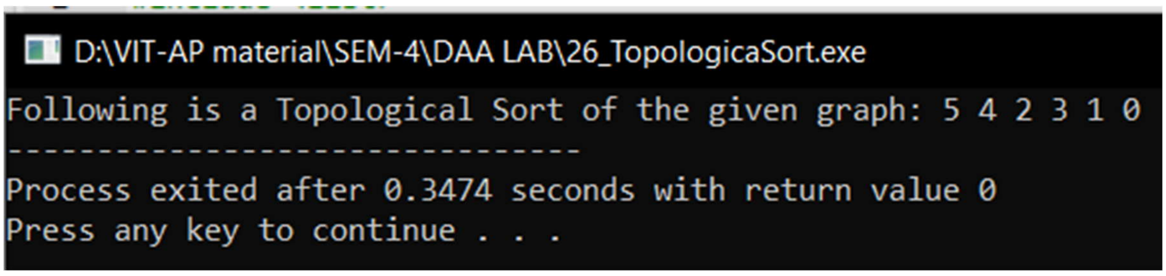
O(V+E)

Code: -

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


class Graph {
    int V;

    list<int>* adj;

    void topologicalSortUtil(int v, bool visited[], stack<int>&
Stack);

public:
    Graph(int V);

    void addEdge(int v, int w);

    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}

void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int>& Stack)
{

    visited[v] = true;

    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    Stack.push(v);
}

void Graph::topologicalSort()
{
    stack<int> Stack;

    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    while (Stack.empty() == false) {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}

int main()
{

    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
```

```
        g.addEdge(3, 1);

        cout << "Following is a Topological Sort of the given graph: ";
        g.topologicalSort();

        return 0;
    }
```

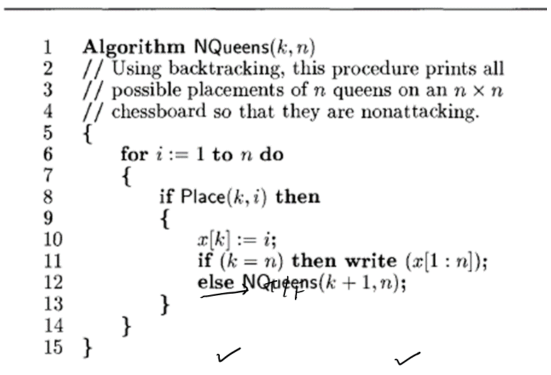
Output: -



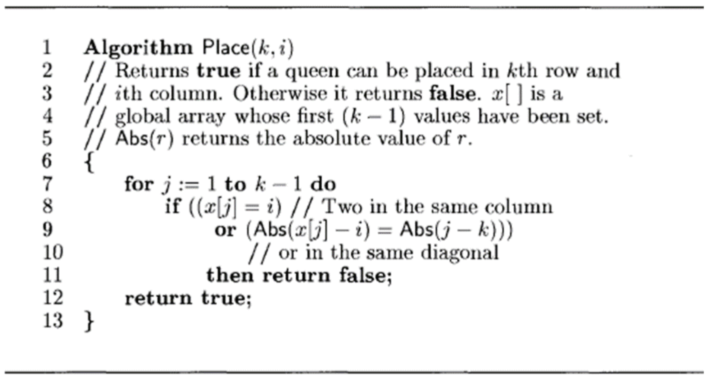
```
D:\VIT-AP material\SEM-4\DAA LAB\26_TopologicaSort.exe
Following is a Topological Sort of the given graph: 5 4 2 3 1 0
--------------------------------
Process exited after 0.3474 seconds with return value 0
Press any key to continue . . .
```

27. Write a C/C++ program to Implement the N-Queens Problem.

Algorithm: -



```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```



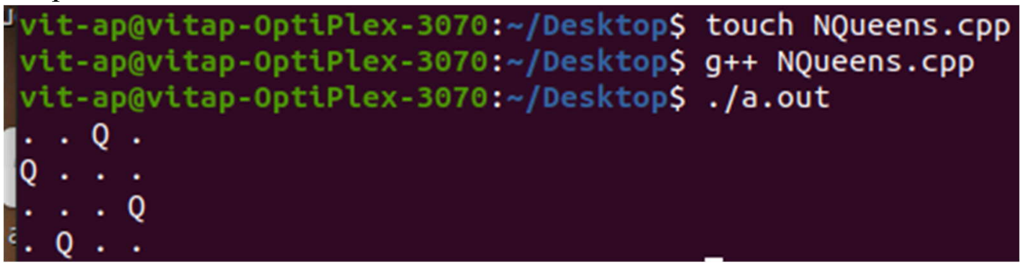
```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k − 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k − 1 do
        if ((x[j] = i) // Two in the same column
            or (Abs(x[j] − i) = Abs(j − k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

Time Complexity: - O(n!)

Code: -

```
#include<iostream>
#define N 4
using namespace std;

void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
        if(board[i][j])
            cout << "Q ";
        else cout<<". ";
        printf("\n");
    }
}

bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

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

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

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

            if (Place(board, col + 1))
                return true;

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

bool NQueens()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (Place(board, 0) == false) {
        cout << "Solution does not exist";
        return false;
    }

    printSolution(board);
    return true;
}

int main()
{
    NQueens();
    return 0;
}
```

Output: -



```
vit-ap@vitap-OptiPlex-3070:~/Desktop$ touch NQueens.cpp
vit-ap@vitap-OptiPlex-3070:~/Desktop$ g++ NQueens.cpp
vit-ap@vitap-OptiPlex-3070:~/Desktop$ ./a.out
. . Q .
Q . . .
. . . Q
. Q . .
```

28. Write a C/C++ program to Implement the Graph Coloring Problem.

Algorithm: -

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for x[k].
        NextValue(k); // Assign to x[k] a legal color.
        if (x[k] = 0) then return; // No new color possible
        if (k = n) then     // At most m colors have been
                            // used to color the n vertices.
            write (x[1 : n]);
        else mColoring(k + 1);
    } until (false);
}
```

```
Algorithm NextValue(k)
// x[1], ..., x[k - 1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been used.
        for j := 1 to n do
        {   // Check if this color is
            // distinct from adjacent colors.
            if ((G[k, j] ≠ 0) and (x[k] = x[j]))
            // If (k, j) is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if (j = n + 1) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

Time Complexity: -

O(m^v)

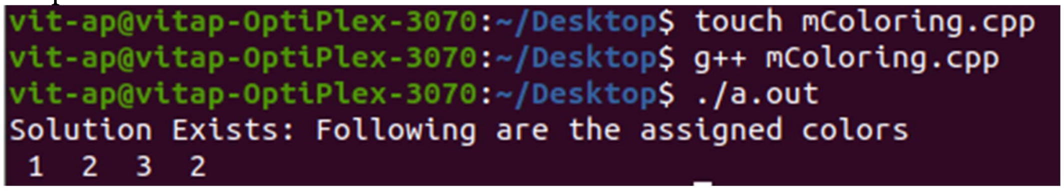
Code: -

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

#define V 4

void printSolution(int color[]);

bool isSafe(int v, bool graph[V][V], int color[], int c)
{
    for (int i = 0; i < V; i++)
        if (graph[v][i] && c == color[i])
            return false;

    return true;
```

```cpp
}

bool graphColoringUtil(bool graph[V][V], int m, int color[],
                       int v)
{

    if (v == V)
        return true;

    for (int c = 1; c <= m; c++) {

        if (isSafe(v, graph, color, c)) {
            color[v] = c;

            if (graphColoringUtil(graph, m, color, v + 1)
                == true)
                return true;

            color[v] = 0;
        }
    }

    return false;
}

bool graphColoring(bool graph[V][V], int m)
{


    int color[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    if (graphColoringUtil(graph, m, color, 0) == false) {
        cout << "Solution does not exist";
        return false;
    }
    printSolution(color);
    return true;
}

void printSolution(int color[])
{
    cout << "Solution Exists:"
         << " Following are the assigned colors"
         << "\n";
    for (int i = 0; i < V; i++)
        cout << " " << color[i] << " ";

    cout << "\n";
```

```
}

int main()
{

    /* Create following graph and test
       whether it is 3 colorable
      (3)---(2)
       |   / |
       |  /  |
       | /   |
      (0)---(1)
    */
    bool graph[V][V] = {
        { 0, 1, 1, 1 },
        { 1, 0, 1, 0 },
        { 1, 1, 0, 1 },
        { 1, 0, 1, 0 },
    };
    int m = 3;
    graphColoring(graph, m);
    return 0;
}
```

Output: -



```
vit-ap@vitap-OptiPlex-3070:~/Desktop$ touch mColoring.cpp
vit-ap@vitap-OptiPlex-3070:~/Desktop$ g++ mColoring.cpp
vit-ap@vitap-OptiPlex-3070:~/Desktop$ ./a.out
Solution Exists: Following are the assigned colors
 1  2  3  2
```

29. Write a C/C++ program to Implement the Graph Coloring Problem.

Algorithm: -

```
Algorithm NextValue($k$)
// $x[1:k-1]$ is a path of $k-1$ distinct vertices. If $x[k]=0$, then
// no vertex has as yet been assigned to $x[k]$. After execution,
// $x[k]$ is assigned to the next highest numbered vertex which
// does not already appear in $x[1:k-1]$ and is connected by
// an edge to $x[k-1]$. Otherwise $x[k]=0$. If $k=n$, then
// in addition $x[k]$ is connected to $x[1]$.
{
    repeat
    {
        $x[k] := (x[k]+1)$ mod $(n+1)$; // Next vertex.
        if $(x[k]=0)$ then return;
        if $(G[x[k-1], x[k]] \neq 0)$ then
        { // Is there an edge?
            for $j := 1$ to $k-1$ do if $(x[j]=x[k])$ then break;
                        // Check for distinctness.
            if $(j=k)$ then // If true, then the vertex is distinct.
                if $((k<n)$ or $((k=n)$ and $G[x[n], x[1]] \neq 0))$
                    then return;
        }
    } until (false);
}
```

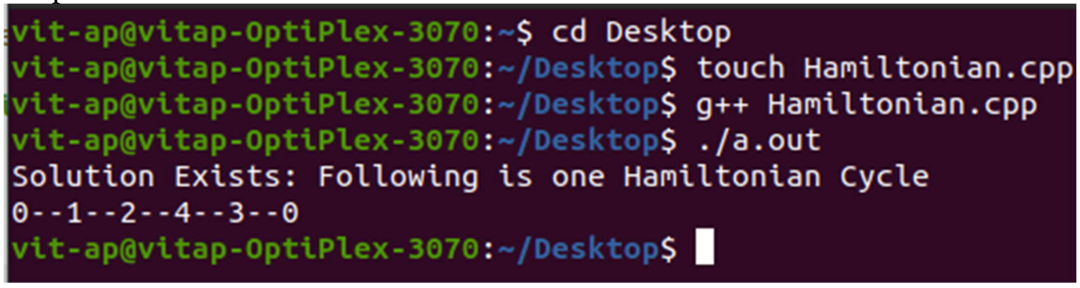
```
Algorithm Hamiltonian($k$)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix $G[1:n, 1:n]$. All cycles begin at node 1.
{
    repeat
    { // Generate values for $x[k]$.
        NextValue($k$); // Assign a legal next value to $x[k]$.
        if $(x[k]=0)$ then return;
        if $(k=n)$ then write $(x[1:n])$;
        else Hamiltonian($k+1$);
    } until (false);
}
```

Time Complexity: - O(N!)

Code: -

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

#define V 5

void printSolution(int path[]){
    cout << "Solution Exists: Following is one Hamiltonian 
Cycle"<<endl;
    for (int i = 0; i < V; i++)
        cout << path[i] << "--";

    cout << path[0] << " "<<endl;

}

bool isSafe(int v, bool graph[V][V], int path[], int pos){
    if (graph [path[pos - 1]][ v ] == 0)
        return false;
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

bool HamCycle(bool graph[V][V], int path[], int pos){
    if (pos == V)
    {
        if (graph[path[pos - 1]][path[0]] == 1)
            return true;
        else
            return false;
    }

    for (int v = 1; v < V; v++)
    {
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            if (HamCycle(graph, path, pos + 1) == true)
                return true;

            path[pos] = -1;
        }
    }
    return false;
}

bool HamCycle(bool graph[V][V]){
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    path[0] = 0;
    if (HamCycle(graph, path, 1) == false )
    {
        cout << "\nSolution does not exist";
        return false;
    }
    printSolution(path);
    return true;
}

int main(){
    bool graph[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                      {1, 1, 0, 0, 1},
                      {0, 1, 1, 1, 0}};

    HamCycle(graph);

    return 0;
}
```

Output: -



```
vit-ap@vitap-OptiPlex-3070:~$ cd Desktop
vit-ap@vitap-OptiPlex-3070:~/Desktop$ touch Hamiltonian.cpp
vit-ap@vitap-OptiPlex-3070:~/Desktop$ g++ Hamiltonian.cpp
vit-ap@vitap-OptiPlex-3070:~/Desktop$ ./a.out
Solution Exists: Following is one Hamiltonian Cycle
0--1--2--4--3--0
vit-ap@vitap-OptiPlex-3070:~/Desktop$
```

Algorithm	Time Complexity			Space Complexity
	Worst Case	Average Case	Best Case	
Sum	O(1)	O(1)	O(1)	O(1)
Multiplication	O(1)	O(1)	O(1)	O(1)
Fibonacci	O(n)	O(n)	O(n)	O(n)
GCD	O(n)	O(logn)	O(1)	O(1)
Factorial	O(n)	O(n)	O(1)	O(1)
Permutation Generator	$O(2^n)$	O(n)	O(n)	$O(2^n)$
Set of Characters	O(n!)	O(n)	O(1)	O(n!)
Linear Search	O(n)	O(n)	O(1)	O(n)
Binary Search	O(logn)	O(logn)	O(1)	O(n)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Travellilng Salesman Problem using Brute Force	$O(2^n*n^2)$	O(n!)	O(n!)	O(n)
0/1 Knapsack Problem using Brute Force	$O(2^n)$	$O(2^n)$	$O(2^n)$	O(n)
Job assignment Problem using Brute Force	O(n!)	O(n!)	O(n!)	O(n)
Fractional Knapsack using Greedy Method	$O(2^n)$	$O(2^n)$	O(n)	O(n)
Job Sequencing with deadlines using Greedy Method	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n)
Dijkstra's Algorithm using Greedy Method	$O(V^2)$	O(V+E)	O(V+E)	O(n)
Minimum Cost Spanning Tree using Greedy Method	$O(n^2)$	O(E*logE)	O(E*logE)	O(2n)
Kruskal's Algorithm using Greedy Method	O(E*logE)	O(E*logE)	O(E*logE)	O(E+V)
TSP using Dynamic Programming	$O(n^2*2^n)$	$O(n^2*2^n)$	$O(n^2)$	O(n)
All Pairs shortest path problem using Dynamic Programming	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n^2)$
Warshall's Algorithm	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n^2)$
Optimal Binary Search Tree	$O(n^3)$	O(logn)	$O(n^2)$	$O(n^2)$
Tree Traversals	O(n)	O(n)	O(n)	O(h)
Topological Sorting	O(V+E)	O(V+E)	O(V)	O(V)
N-Queens Problem	O(n!)	$O(n^2)$	O(1)	$O(n^2)$
Graph Coloring Problem	$O(m^V)$	$O(m^V)$	O(1)	O(V)
Hamiltonian Graph	$O(2^n))$	$O(2^n)$	O(n!)	O(1)