

**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**

|  |  |
| --- | --- |
| S. No | Title |
| 1 | Sum |
| 2 | Multiplication |
| 3 | Fibonacci |
| 4 | GCD |
| 5 | Factorial |
| 6 | Permutation Generator |
| 7 | Set of Characters |
| 8 | Linear Search |
| 9 | Binary Search |
| 10 | Merge Sort |
| 11 | Quick Sort |
| 12 | Travellilng Salesman Problem using Brute Force |
| 13 | 0/1 Knapsack Problem using Brute Force |
| 14 | Job assignment Problem using Brute Force |
| 15 | Fractional Knapsack using Greedy Method |
| 16 | Job Sequencing with deadlines using Greedy Method |
| 17 | Dijkstra’s Algorithm using Greedy Method |
| 18 | Minium Cost Spanning Tree using Greedy Method |
| 19 | Kruskal’s Algorithm using Greedy Method |
| 20 | TSP using Dynamic Programming |
| 21 | All Pairs shortest path problem using Dynamic Programming |
| 22 | Warshall’s Algorithm |
| 23 | Optimal Binary Search Tree |
| 24 | Tree Traversals |
| 25 | Topological Sorting |
| 26 | N-Queens Problem |
| 27 | Graph Coloring Problem |
| 28 | Hamiltonian Graph |
| 29 | Sum |

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

1. Find the Sum of two Positive Integers.

#include<stdio.h>

void main()

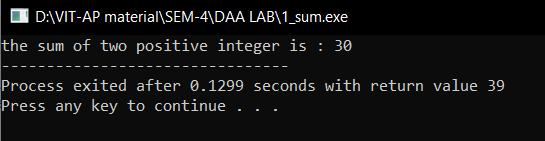
{

    int a=10, b=20;

    printf("the sum of two positive integer is : %d",a+b);

}

* **Output:**

****

1. Find the Multiplication of two Positive Integers.

#include<stdio.h>

void main()

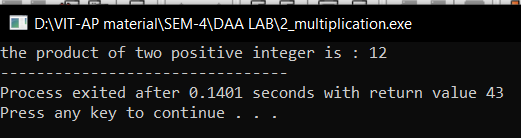
{

    int a=2, b=6;

    printf("the product of two positive integer is : %d",a\*b);

}

* **Output:**



1. 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: -

#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);

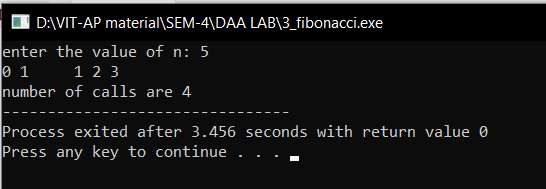
     fibonacci(n-2, &calls);

     printf("\nnumber of calls are %d", calls);

     return 0;

}

* **Output:**



1. Find the GCD of two numbers.

#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()

{

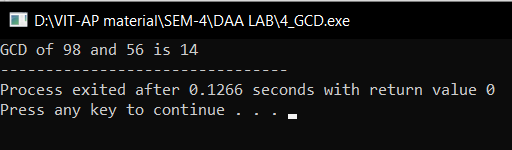
    int a = 98, b = 56;

    printf("GCD of %d and %d is %d ", a, b, gcd(a, b));

    return 0;

}

* **Output:**



1. 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: -

#include <stdio.h>

int factorial(int n)

{

    int result = 1, i;

    for (i = n; i > 0; i--) {

        result \*= i;

    }

    return result;

}

int main()

{

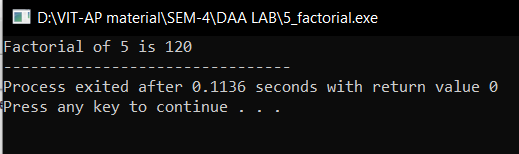
    int num = 5;

    printf("Factorial of %d is %d", num, factorial(num));

    return 0;

}

* **Output:**



1. 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()

{

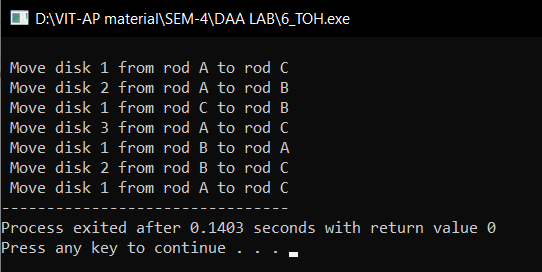
    int n = 3;

    TowerOfHanoi(n, 'A', 'C', 'B');

    return 0;

}

* **Output:**



1. 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.

#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";

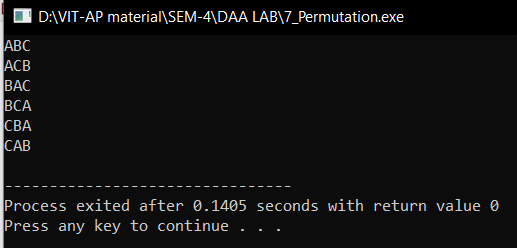
    int n = strlen(str);

    permute(str, 0, n - 1);

    return 0;

}

* **Output:**



1. 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];

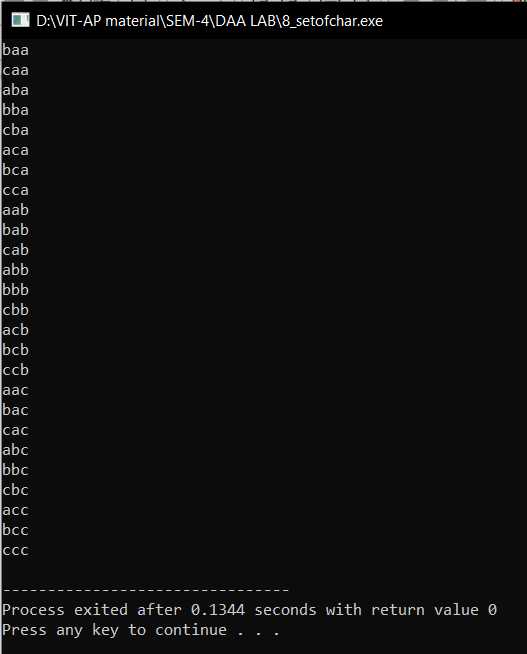
    current[k] = '\0'; // null-terminate the string

    generateStrings(input\_characters, k, current);

    return 0;

}

* **Output:**



1. Write a c/c++ program to implement Linear Search.

Algorithm: -

Procedure search(i, j, x: i, j, x integers, 1≤ i ≤ j ≤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]);

    }

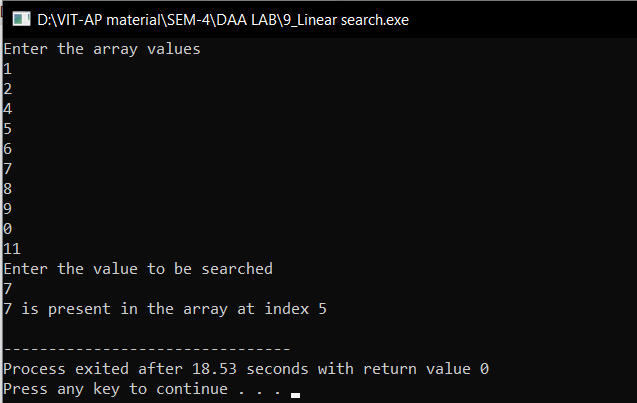
    printf("Enter the value to be searched\n");

    scanf("%d", &x);

    linear\_search(arr, n, x);

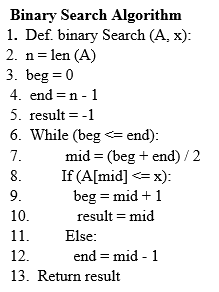
}

* **Output:**

****

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

Algorithm: -



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;

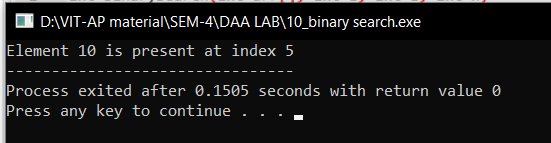
    int result = binarySearch(arr, 0, n - 1, x);

    (result == -1) ? printf("Element is not present in array") : printf("Element %d is present at index %d", x, result);

    return 0;

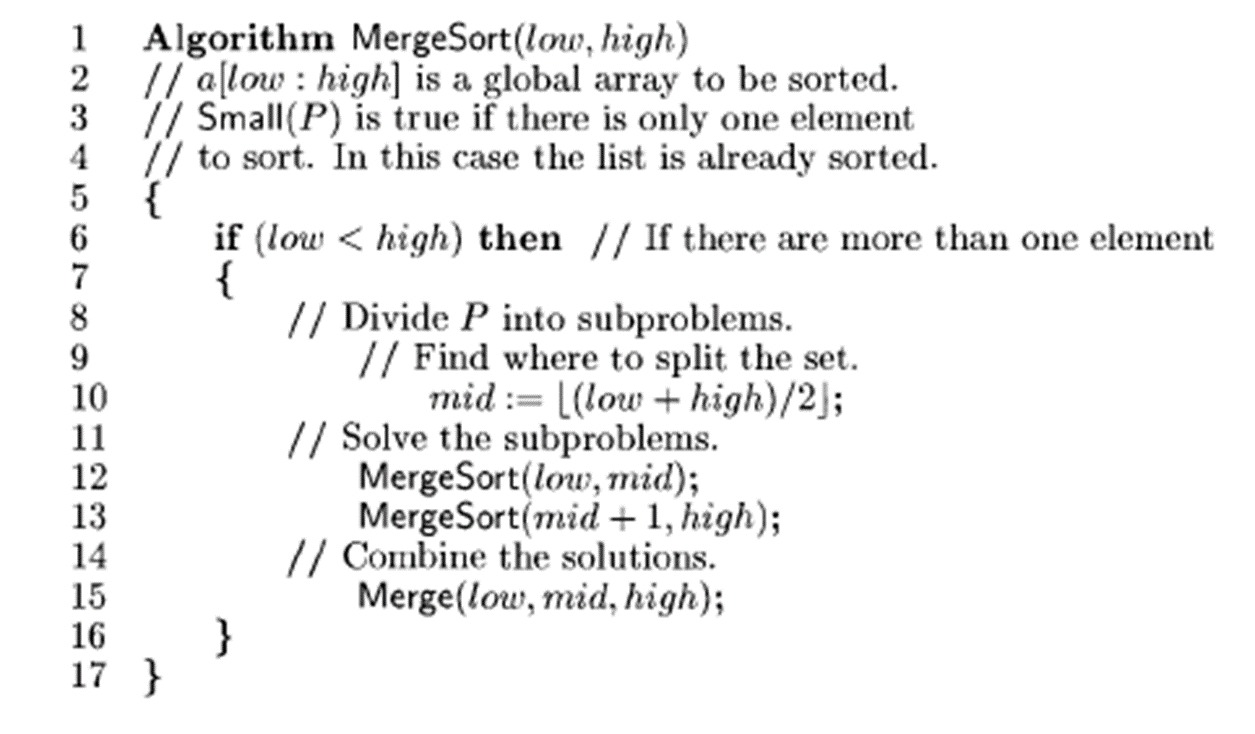
}

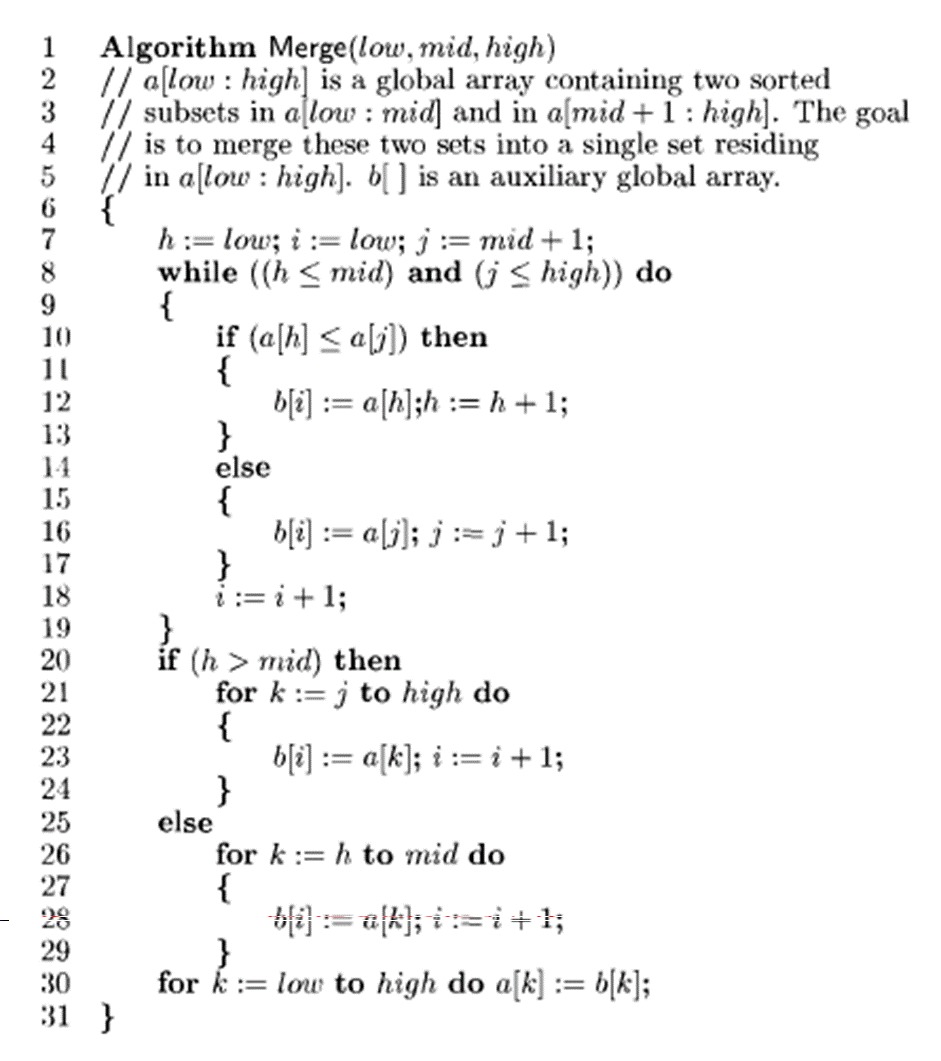
* **Output:**



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

Algorithm: -





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++)

    {

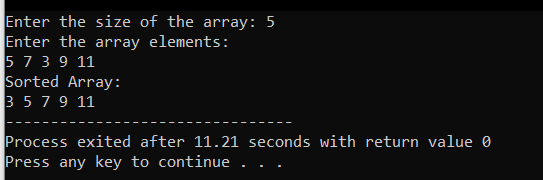
        printf("%d\t", arr[i]);

    }

    return 0;

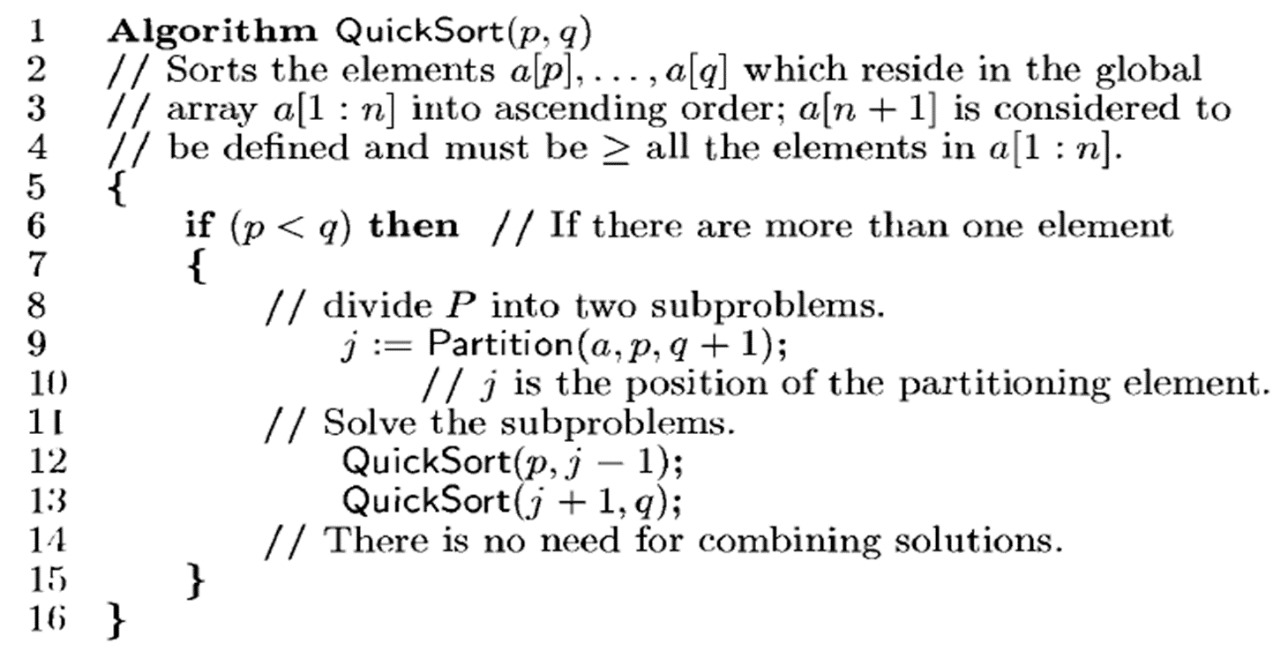
}

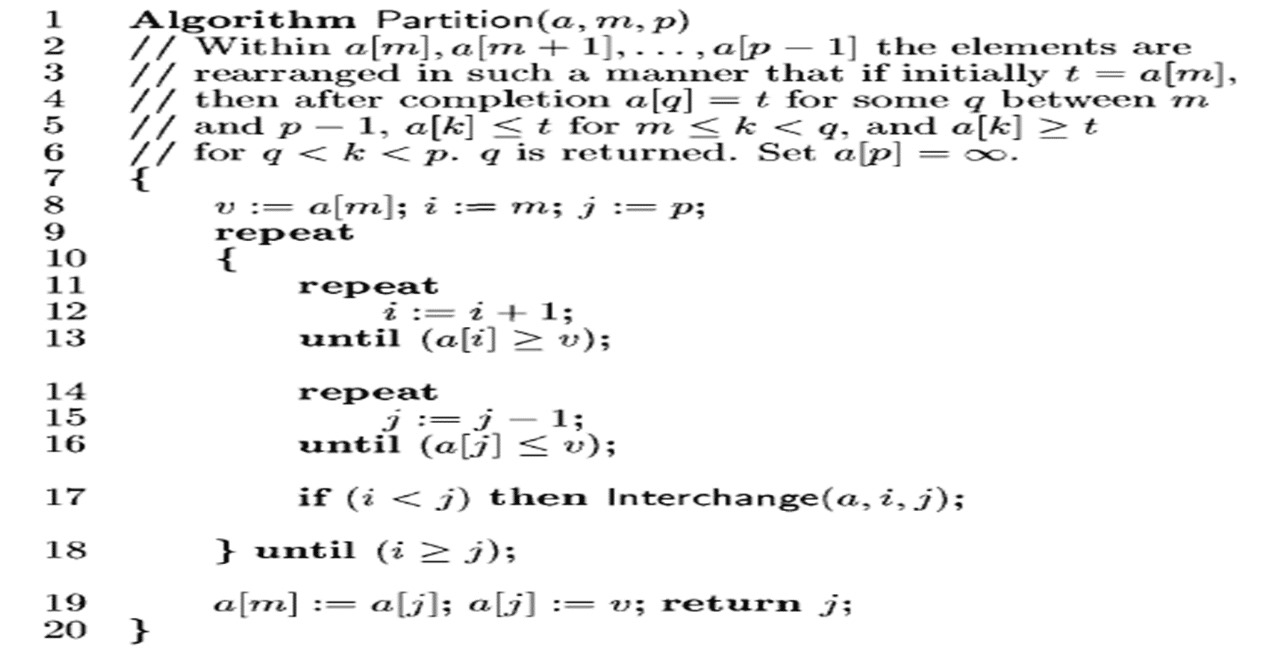
* **Output:**

****

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

Algorithm: -





Time Complexity: -

O(nlogn)

Code: -

#include <stdio.h>

void swap(int \*a, int \*b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

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++) {

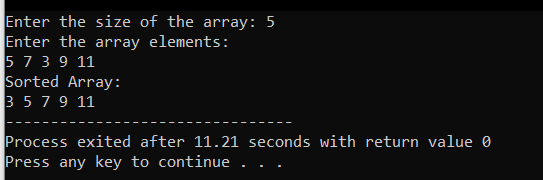
        printf("%d ", arr[i]);

    }

    return 0;

}

* **Output:**



1. 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(2n\*n2)

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}};

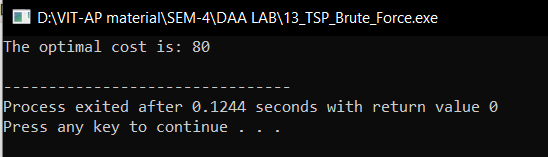
    int s = 0;

    printf("The optimal cost is: %d\n", travllingSalesmanProblem(graph, s));

    return 0;

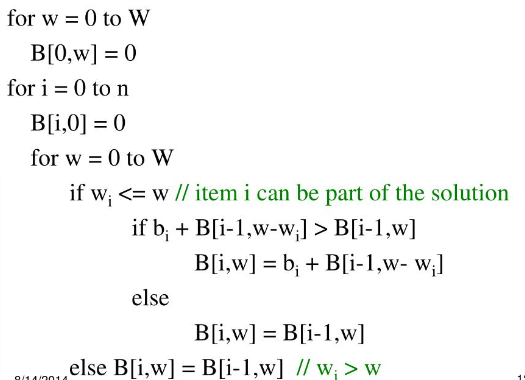
}

Output: -

****

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

Algorithm: -



Time Complexity: -

O(2n)

Code: -

#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;

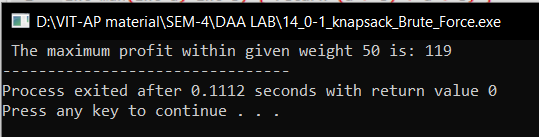
    int n = sizeof(profit) / sizeof(profit[0]);

    printf(" The maximum profit within given weight %d is: %d", W, knapSack(W, weight, profit, n));

    return 0;

}

Output: -



1. 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: -

#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);

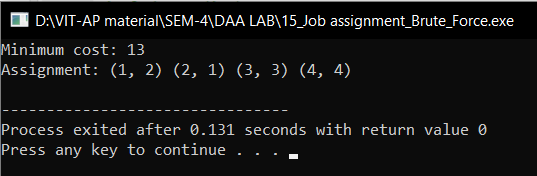
    }

    printf("\n");

    return 0;

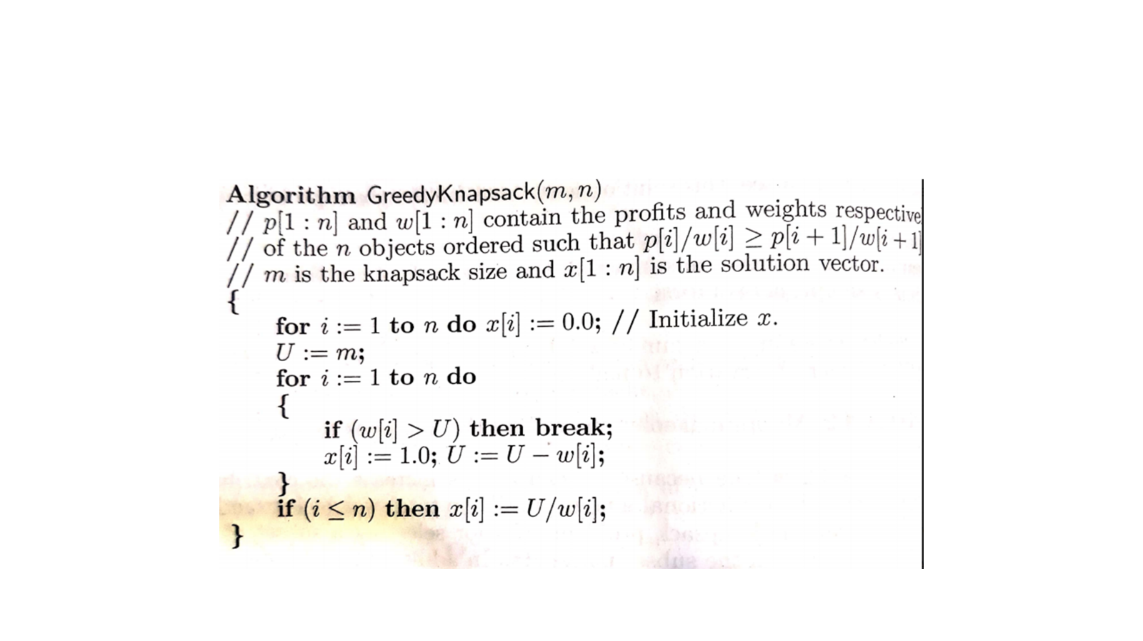
}

Output: -

****

1. Write a C/C++ program to Implement the Fraction Knapsack Problem using Greedy Method.

Algorithm: -



Time Complexity: -

O(2n)

Code: -

#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]);

            }

        }

    }

}

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;

    }

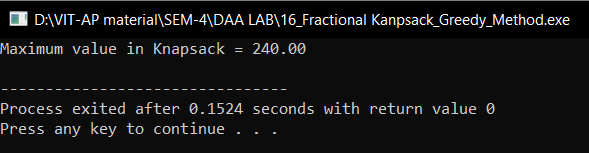
    float totalValue = fractionalKnapsack(capacity, items, n);

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

    return 0;

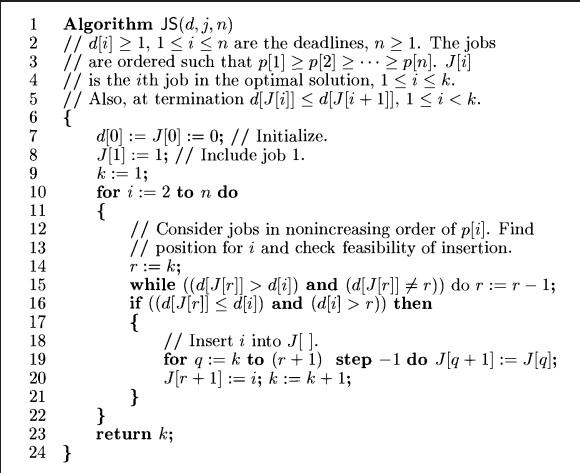
}

Output: -



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

Algorithm: -



Time Complexity: -

O(n2)

Code: -

#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;

}

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(

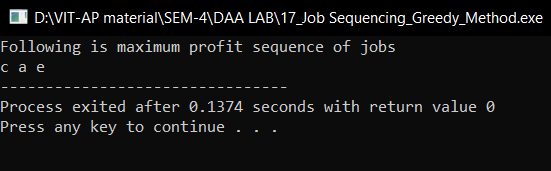
        "Following is maximum profit sequence of jobs \n");

    printJobScheduling(arr, n);

    return 0;

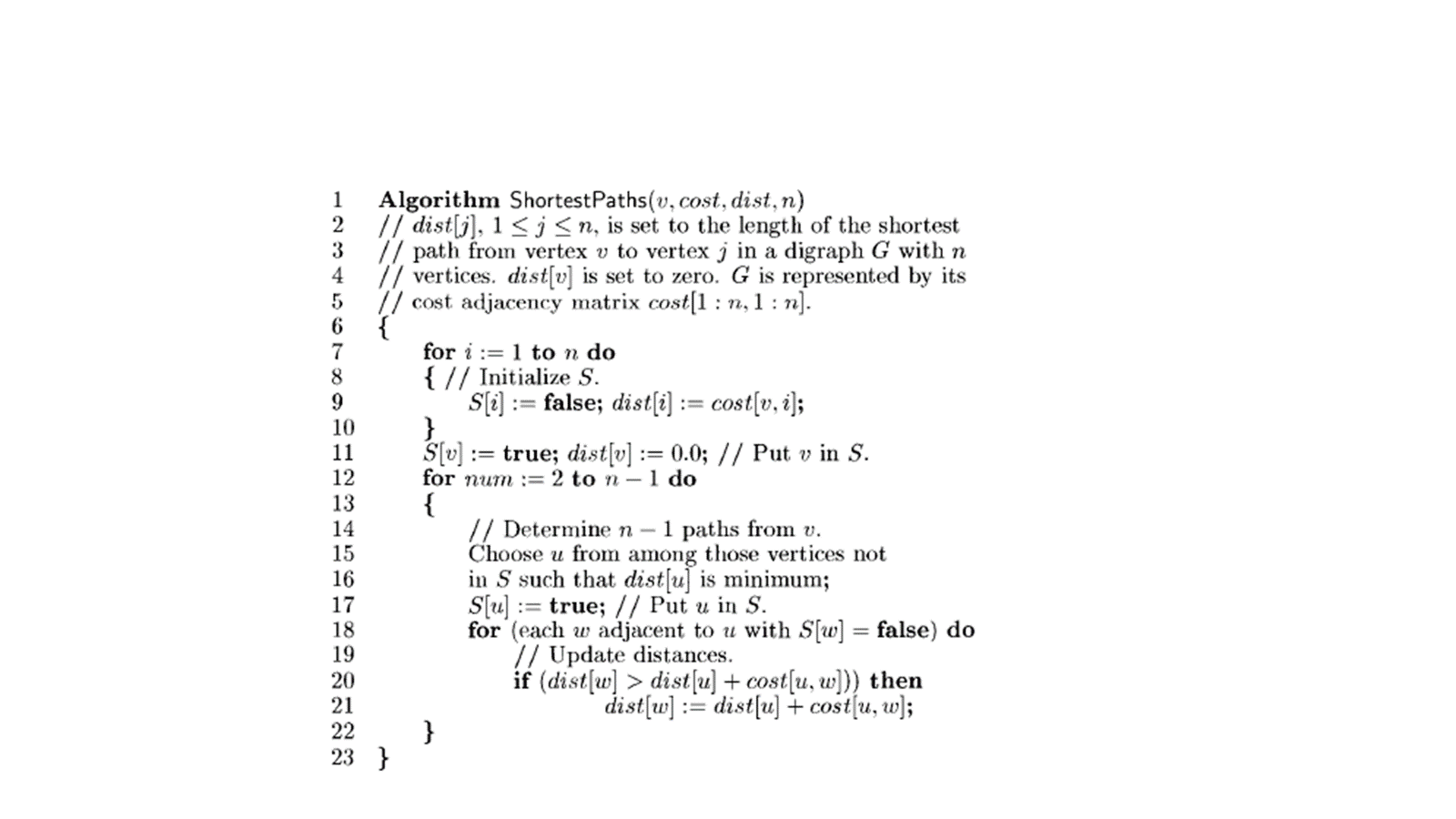
}

Output: -

****

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

Algorithm: -



Time Complexity: -

O(n2)

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},

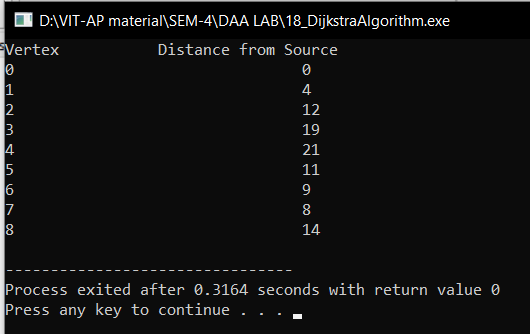
                       {0, 0, 2, 0, 0, 0, 6, 7, 0}};

    dijkstra(graph, 0);

    return 0;

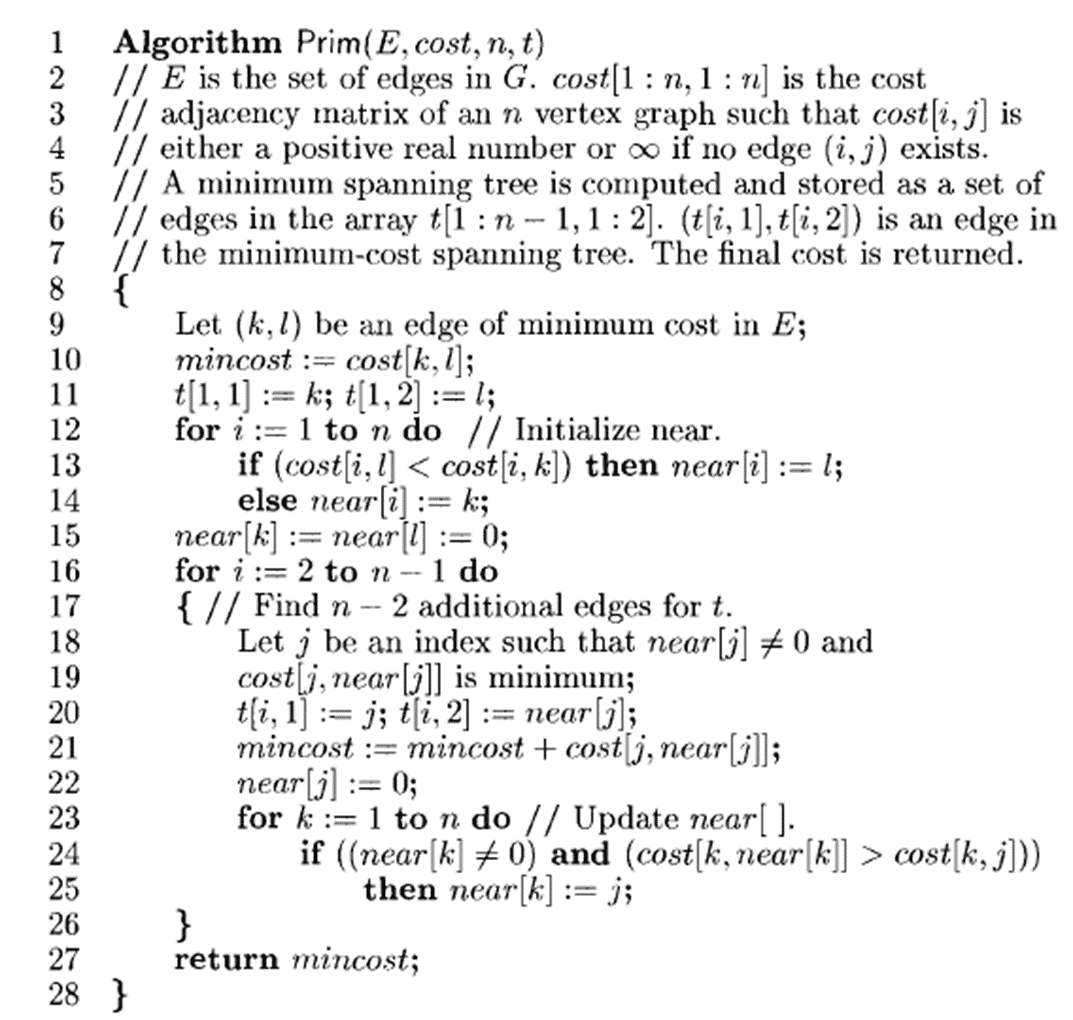
}

Output: -



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

Algorithm: -



Time Complexity: -

O(n2)

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},

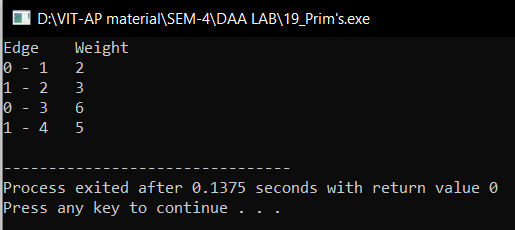
                       {0, 5, 7, 9, 0}};

    primMST(graph);

    return 0;

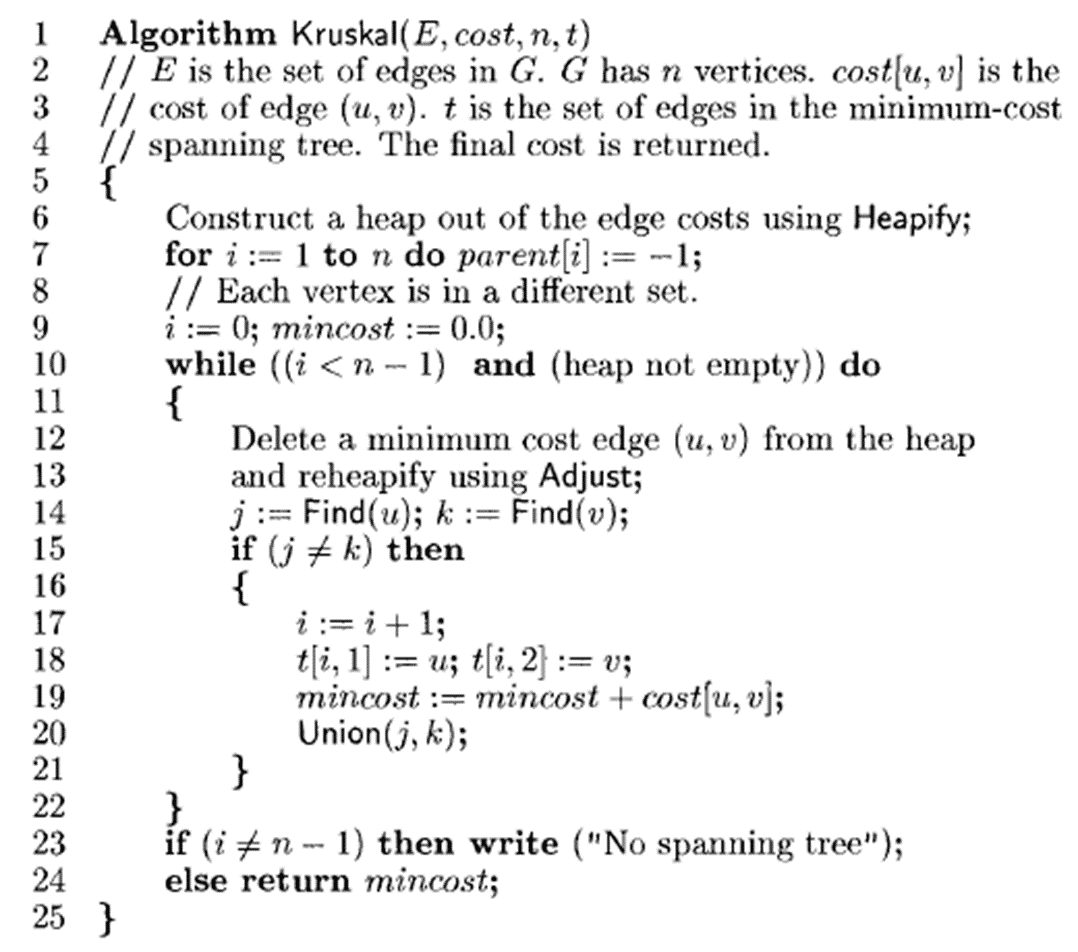
}

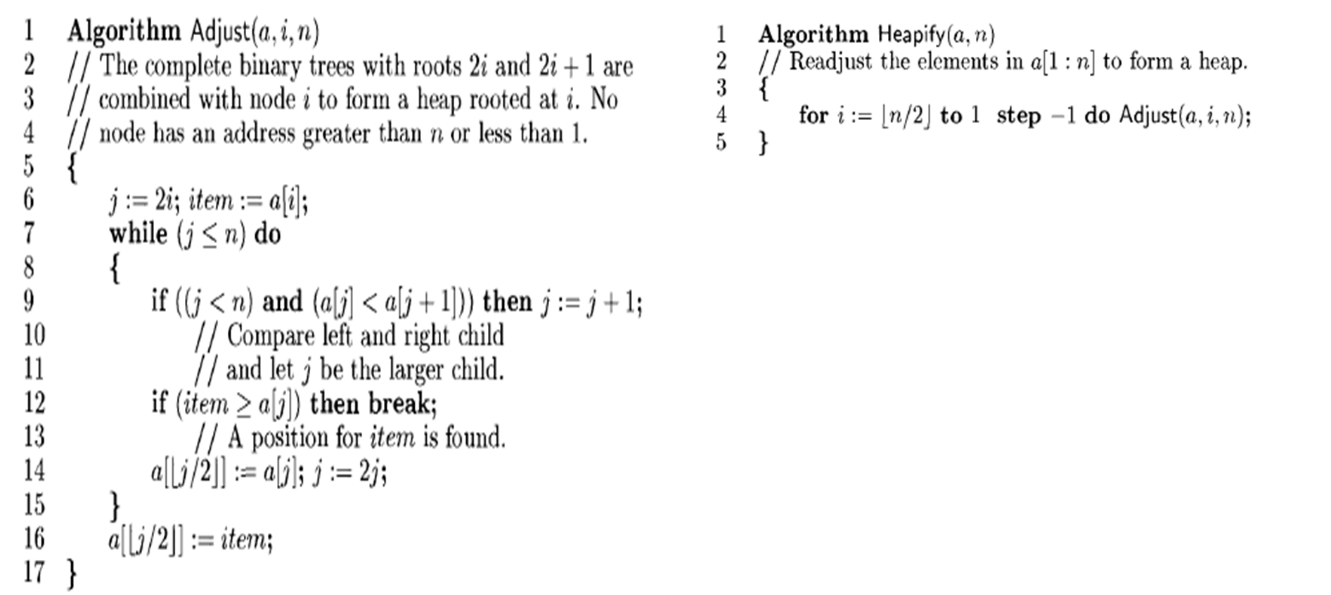
Output: -

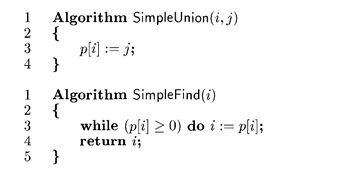


1. Write a C/C++ program to Implement the Kruskal’s Algorithm for the construction of a minimum cost-spanning tree using the Greedy Methodology.

Algorithm: -







Time Complexity: -

O(E\*logE)

Code: -

#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]++;

    }

}

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},

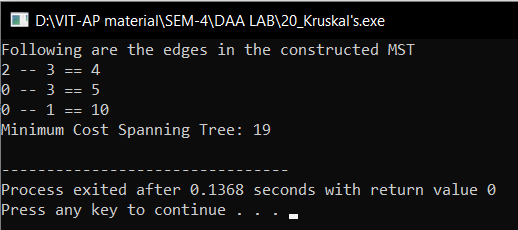
                      {2, 3, 4}};

    kruskalAlgo(5, edge);

    return 0;

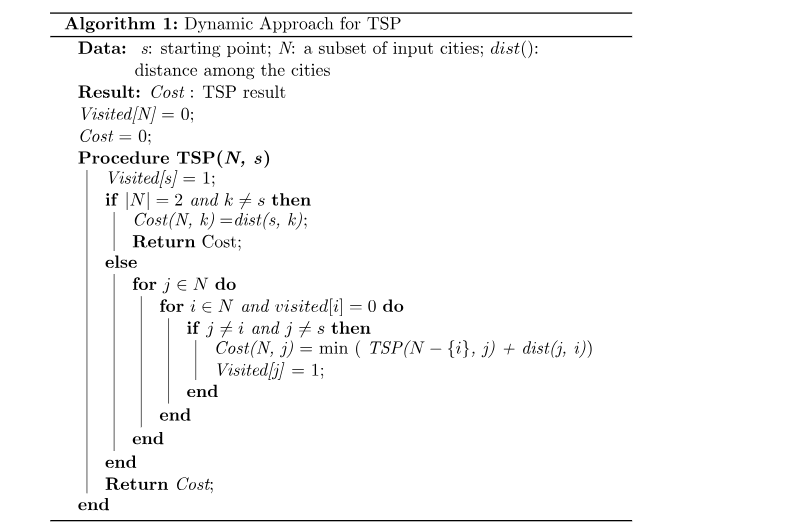
}

Output: -



1. Write a C/C++ program to Implement the Travelling Salesperson (TSP) Problem using Dynamic Programming.

Algorithm: -



Time Complexity: -

O(n2\*2n)

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++)

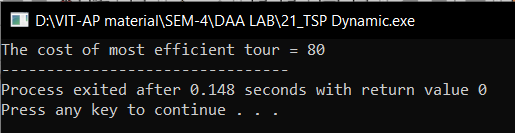
        ans = min(ans, fun(i, (1 << (n + 1)) - 1) + dist[i][1]);

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

    return 0;

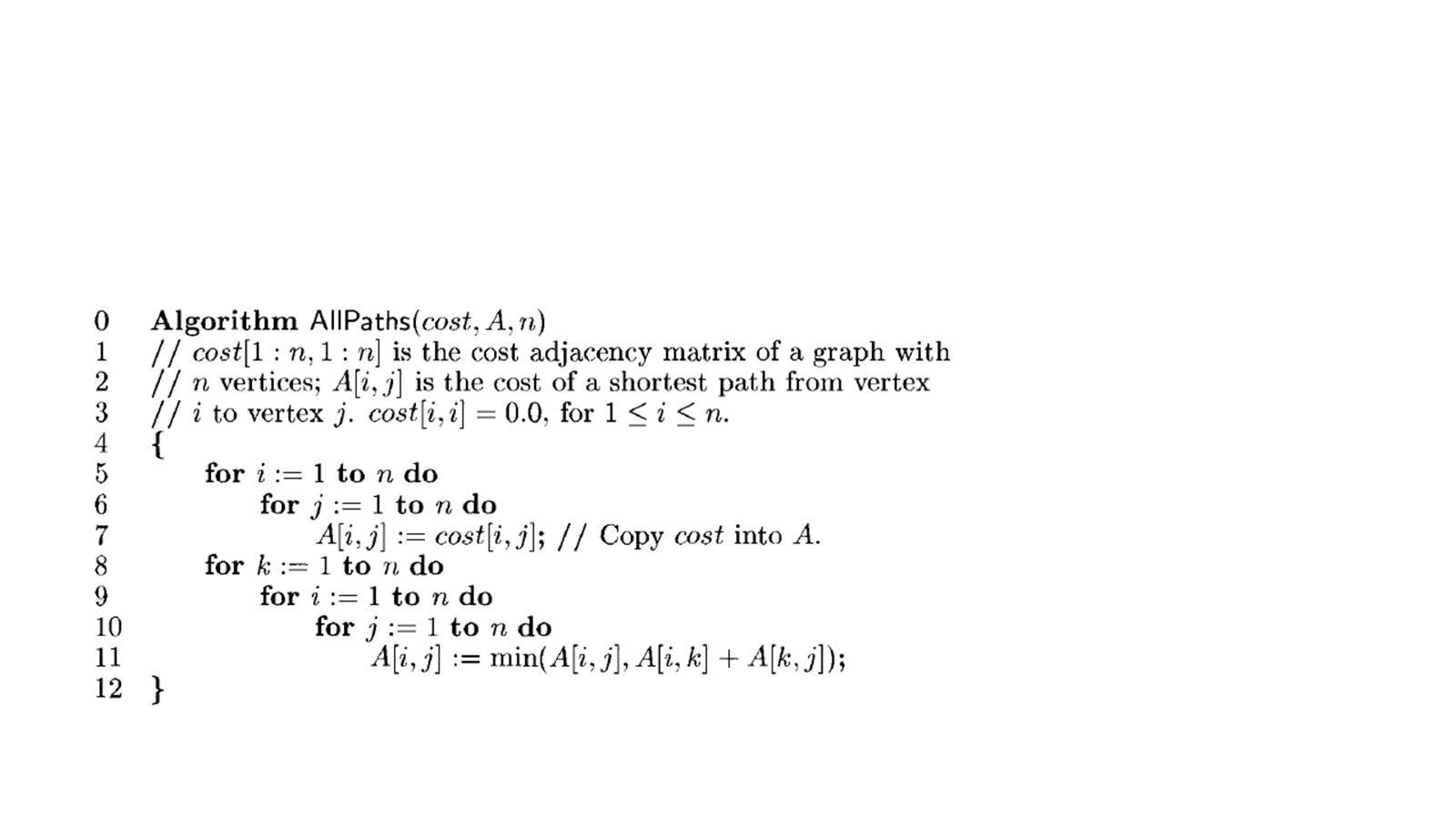
}

Output: -

****

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

Algorithm: -



Time Complexity: -

O(n3)

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},

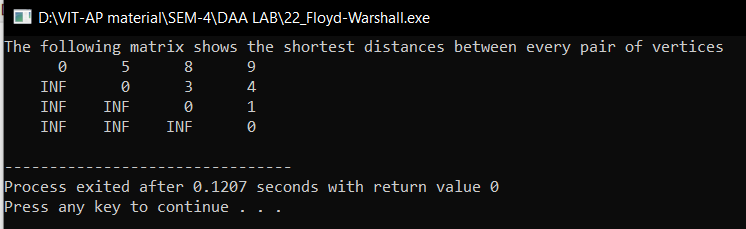
                       {INF, INF, INF, 0}};

    floydWarshall(graph);

    return 0;

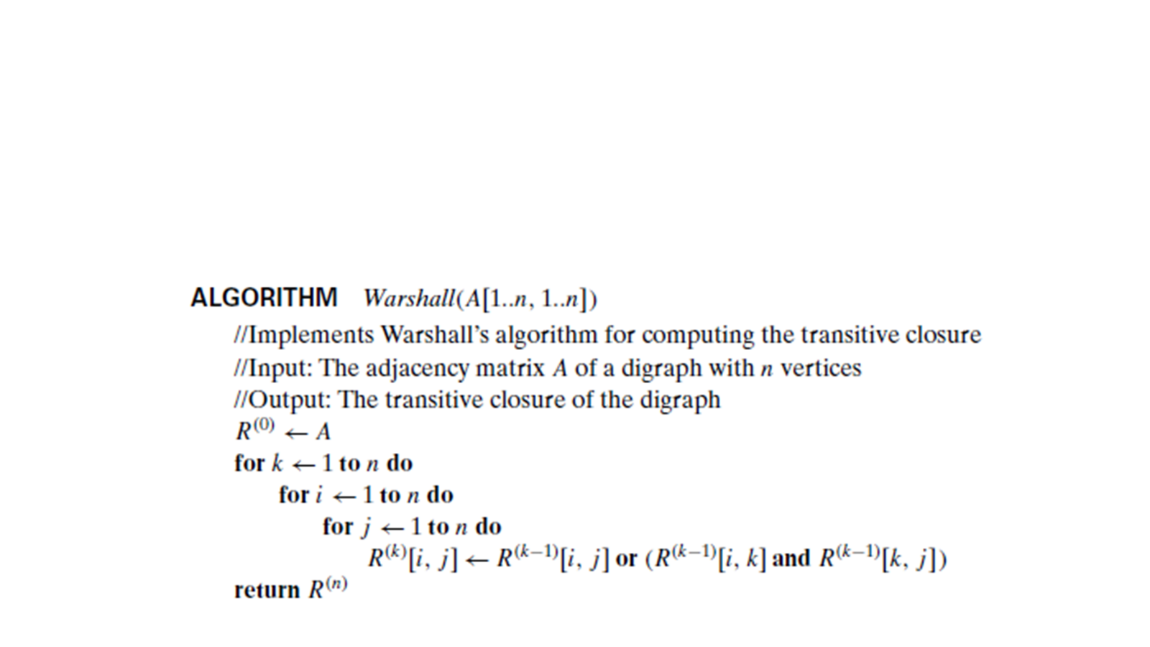
}

Output: -



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

Algorithm: -



Time Complexity: -

O(n3)

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},

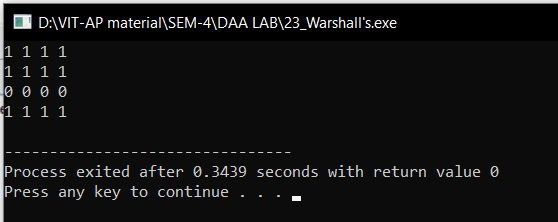
                        {1, 0, 1, 0} };

    transitiveClosure(graph);

    return 0;

}

Output: -

****

1. 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(n3)

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]);

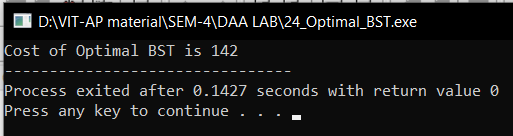
    printf("Cost of Optimal BST is %d ",

            optimalSearchTree(keys, freq, n));

    return 0;

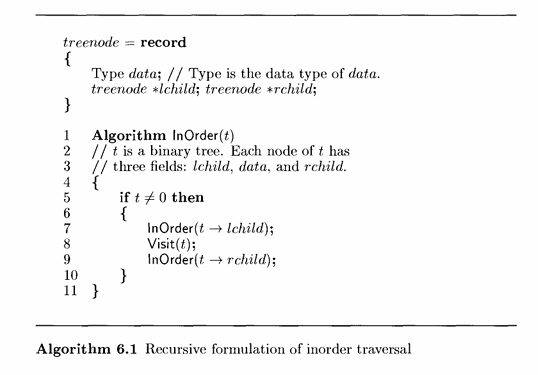
}

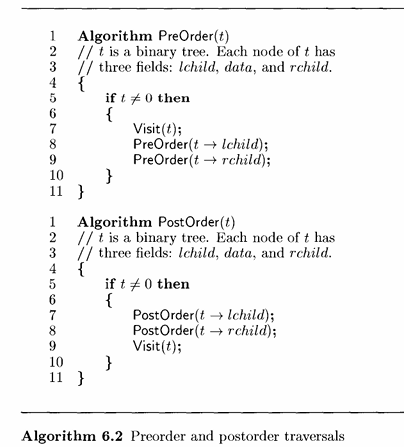
Output: -



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

Algorithm: -





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";

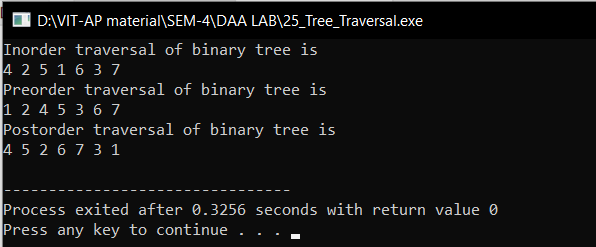
    printPostorder(root);

    cout<<endl;

    return 0;

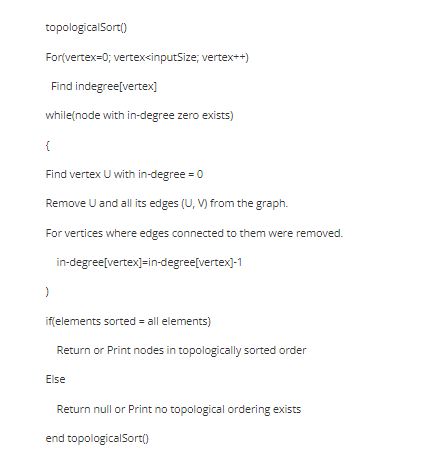
}

Output: -

****

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

Algorithm: -



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);

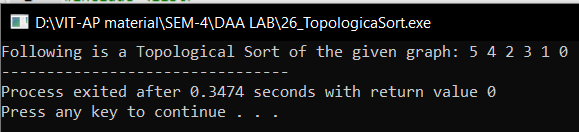
    cout << "Following is a Topological Sort of the given graph: ";

    g.topologicalSort();

    return 0;

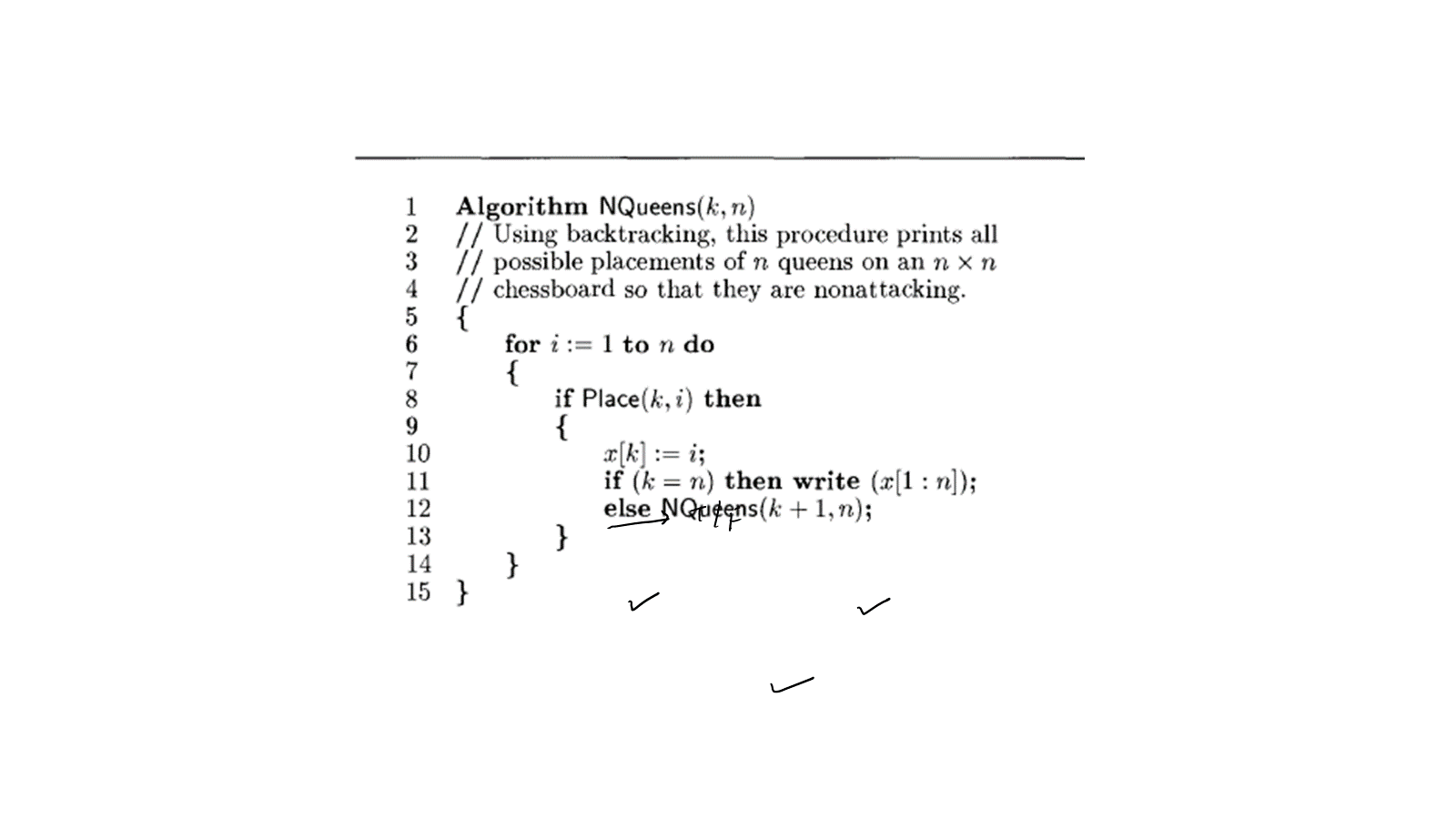
}

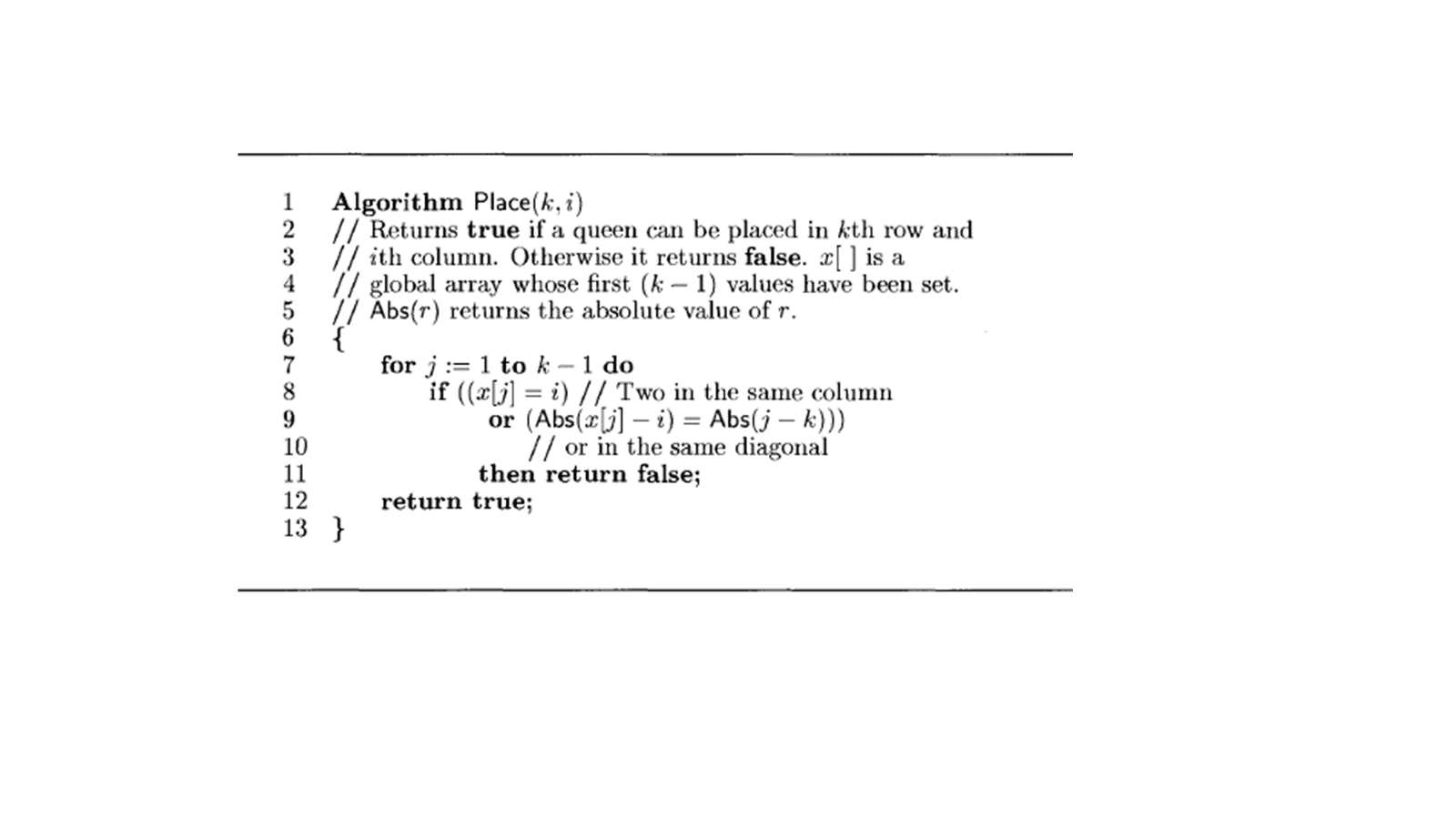
Output: -



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

Algorithm: -





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()

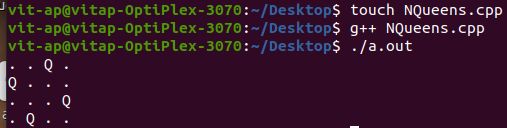
{

    NQueens();

    return 0;

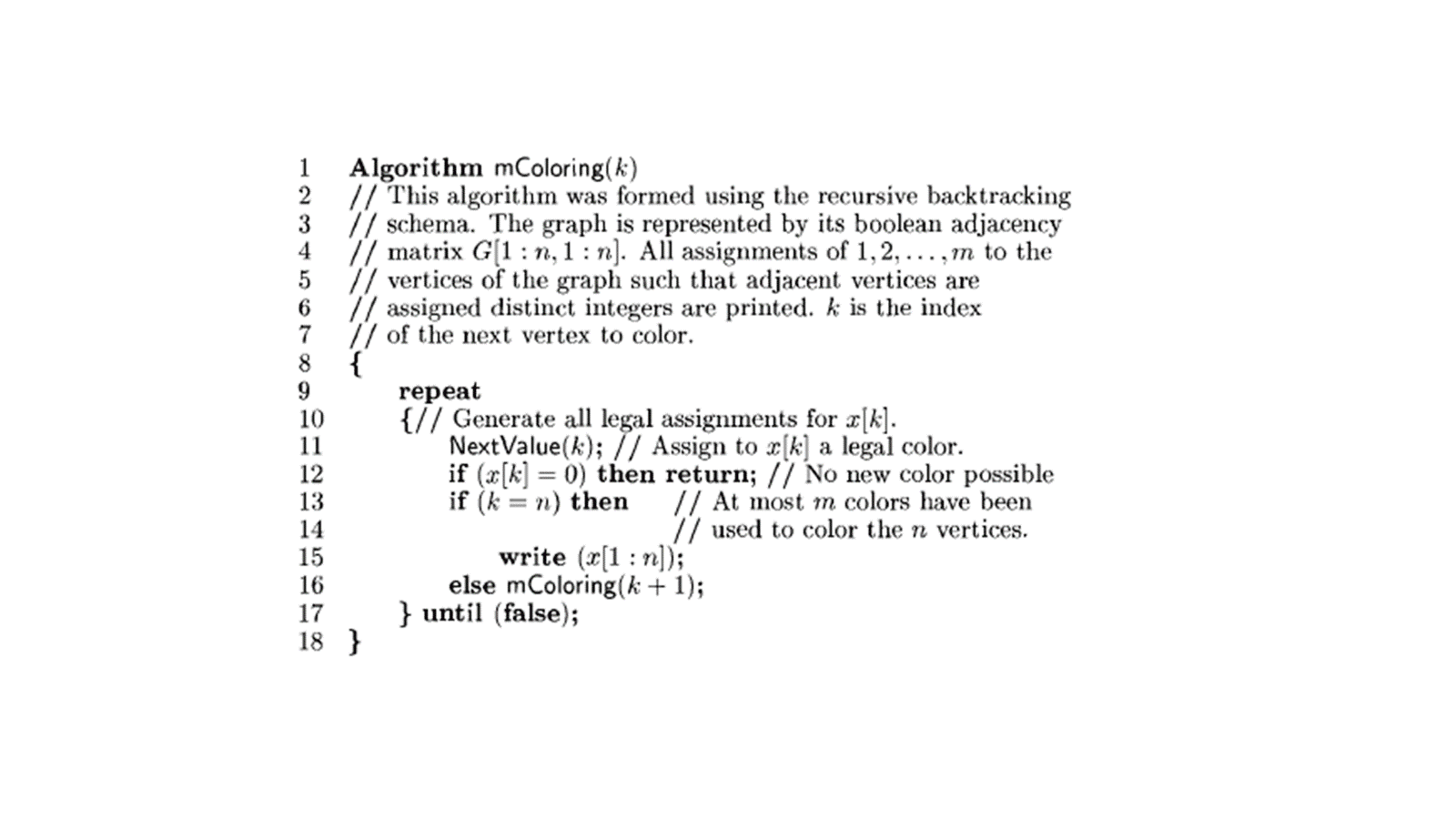
}

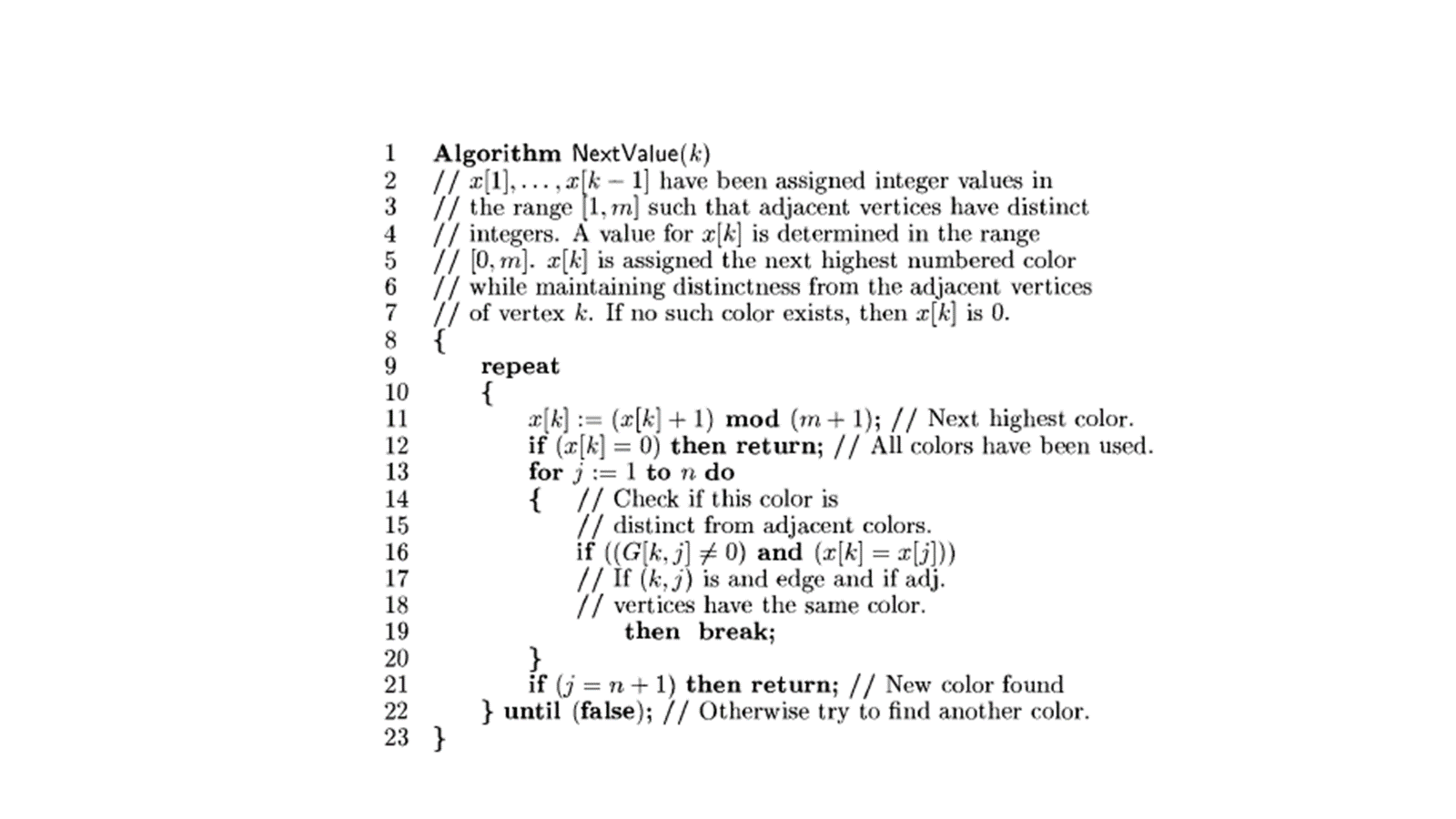
Output: -



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

Algorithm: -





Time Complexity: -

O(mv)

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;

}

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 },

    };

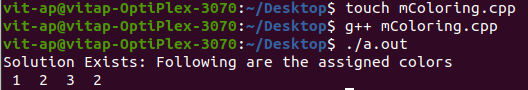
    int m = 3;

    graphColoring(graph, m);

    return 0;

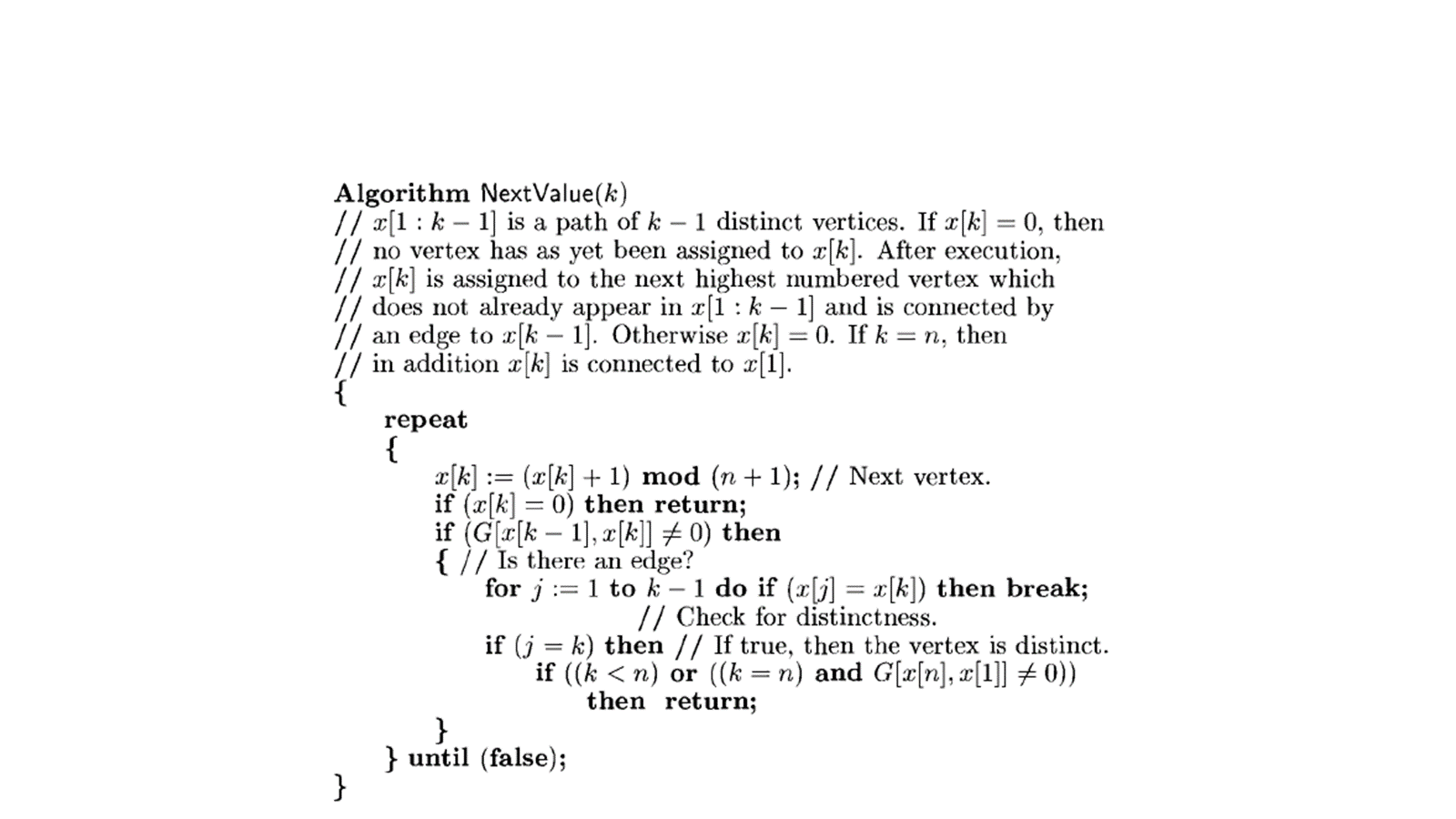
}

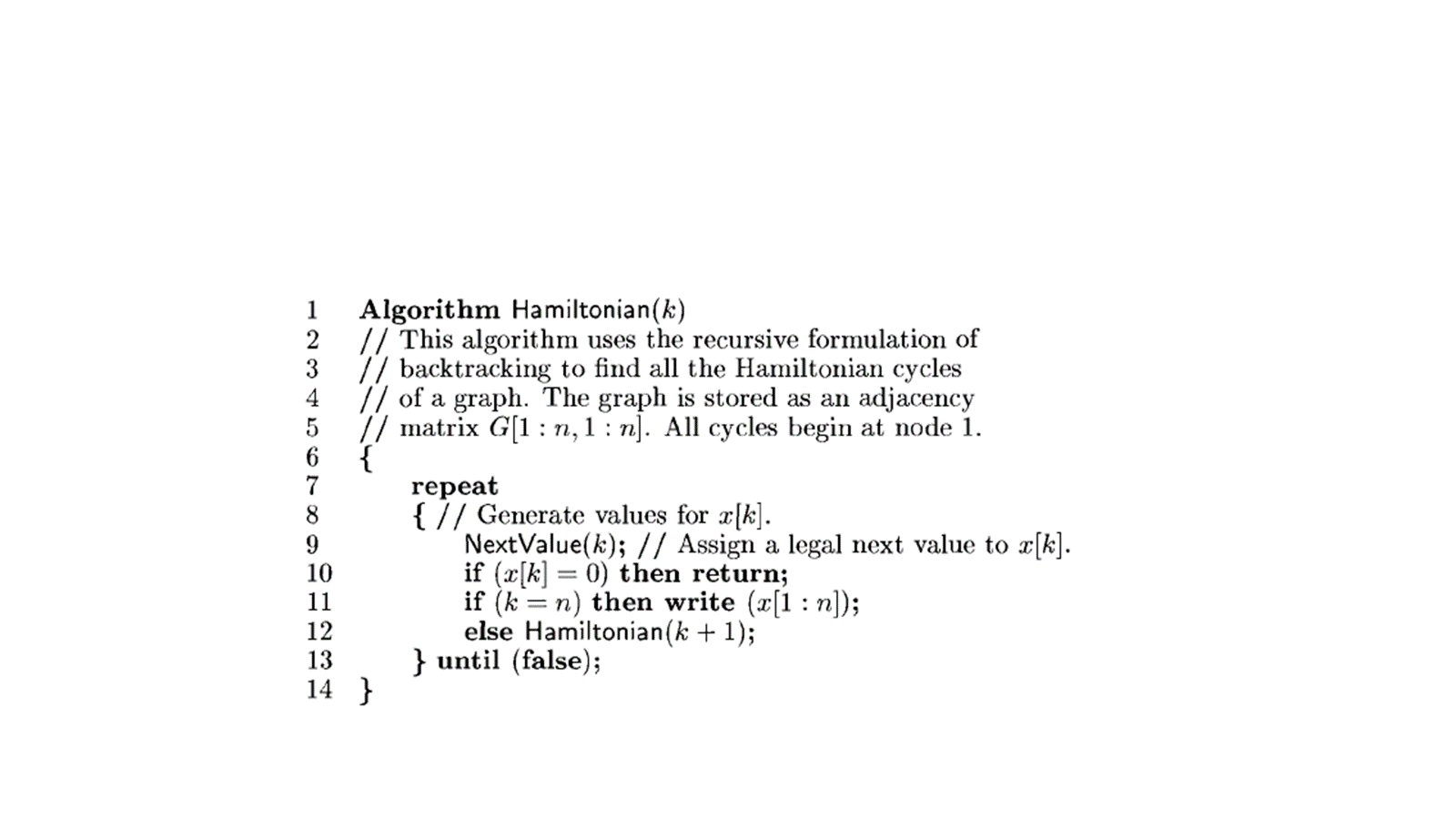
Output: -



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

Algorithm: -





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},

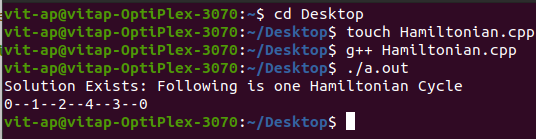
                 {1, 1, 0, 0, 1},

                 {0, 1, 1, 1, 0}};

    HamCycle(graph);

    return 0;

}

Output: -

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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(2n) | O(n) | O(n) | O(2n) |
| 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(n) | O(n) |
| Travellilng Salesman Problem using Brute Force | O(2n\*n2) | O(n!) | O(n!) | O(n) |
|  |
| 0/1 Knapsack Problem using Brute Force | O(2n) | O(2n) | O(2n) | O(n) |  |
| Job assignment Problem using Brute Force | O(n!) | O(n!) | O(n!) | O(n) |  |
|  |
| Fractional Knapsack using Greedy Method | O(2n) | O(2n) | O(n) | O(n) |  |
| Job Sequencing with deadlines using Greedy Method | O(n2) | O(n2) | O(n2) | O(n) |  |
|  |
| Dijkstra’s Algorithm using Greedy Method | O(V2) | O(V+E) | O(V+E) | O(n) |  |
| Minimum Cost Spanning Tree using Greedy Method | O(n2) | 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(n2\*2n) | O(n2\*2n) | O(n2) | O(n) |  |
| All Pairs shortest path problem using Dynamic Programming | O(n3) | O(n3) | O(n3) | O(n2) |  |
|  |
| Warshall’s Algorithm | O(n3) | O(n3) | O(n3) | O(n2) |  |
| Optimal Binary Search Tree | O(n3) | O(logn) | O(n2) | O(n2) |  |
| 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(n2) | O(1) | O(n2) |  |
| Graph Coloring Problem | O(mV) | O(mV) | O(1) | O(V) |  |
|  |
| Hamiltonian Graph | O(2n)) | O(2n) | O(n!) | O(1) |  |