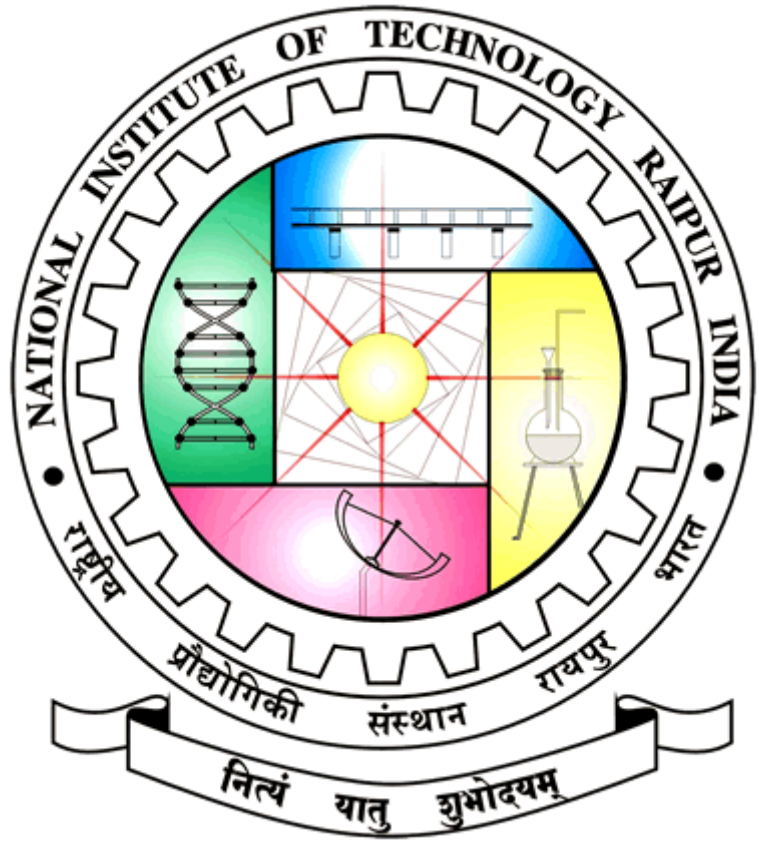
National Institute of Technology, Raipur

Department of Computer Science and Engineering



ANALYSIS AND DESIGN OF ALGORITHMS LAB- Practical Work

COURSE: B.Tech.

SEMESTER: IV

NAME: KUNAL SACHDEVA

ROLL NO. : 19115045

FACULTY IN-CHARGE: Mr. Nilesh Verma

No OF EXPERIMENTS: 15

COURSE OBJECTIVE

CO-1	**To learn how to analyse the complexity of Algorithm**
CO-2	**To compare and evaluate algorithms in terms of time and space complexity**
CO-3	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations **(Design/development of solutions)**.

Q1. Write a program to implement Radix Sort.

Ans. Logic:

i) Find the largest element in array "Max". Let x be no. of digits in the "max". x is calculated as we have to go through all the significant positions.

ii) Now, go through each significant place one by one. We use counting sort to sort the digits at each significant place. So, first sort no.s w.r.t. one's place, then move to ten's place, then to hundred's untill all 'x' places have been sorted.

iii) Print the sorted array.

Time complexity: $O(d(n+b))$ where b is the base of representing no.s, $d = \log_b k$ where k is the max no.

Code:

```
#include <iostream>
using namespace std;
int getMax(int arr[], int n)
{
    int max = arr[0];
    for(int i = 1; i < n; i++)
        if(arr[i] > max)
            max = arr[i];
    return max;
}
```

```
void countSort (int arr[], int n, int exp)
{
    int output[n];
    int i, count[10] = {0};
    for (i=0; i<n; i++)
        count[((arr[i]/exp) %10]++;
    for (i=1; i<10; i++)
        count[i] += count[i-1];
    for (i=n-1; i>=0; i--)
    {
        output[ count[(arr[i]/exp)%10]-1] = arr[i];
        count[(arr[i]/exp) %10]--;
    }
    for (i=0; i<n; i++)
        arr[i] = output[i];
}
void radixsort (int arr[], int n)
{
    int m = getMax(arr, n);
    for (int exp=1; m/exp >0; exp*=10)
        countSort(arr, n, exp);
}
void print (int arr[], int n)
{
    for (int i=0; i<n; i++)
        cout arr[i] << " ";
}
```

```
int main()
{
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr)/sizeof(arr[0]);
    radixSort(arr, n);
    print(arr, n);
    return 0;
}
```

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
2 24 45 66 75 90 170 802
```

Q.2 Write a program to implement Counting sort.

Ans. Logic:

i) Find out the maximum element. Let it be 'max'.

ii) Initialize an array of length max + 1 with all elements 0. This array is used for storing count of elements of the array.

iii) Store the count of each element at their respective index in 'count' array.

iv) Store cummulative sum of the elements of the count array. It helps in placing the elements into correct index of the sorted array.

v) Find the index of each element of the original array in the count array. This gives the cummulative count. Place the element in the index calculated.

vi) After placing each element at its correct position, decrease count by one.

Time complexity: Time complexity for worst case is O(n+k)

Code:

```
#include <bits/stdc++.h>
using namespace std;
void countSort(vector<int>& arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int min = *min_element(arr.begin(), arr.end());
    int range = max - min + 1;
```

```
vector<int> count(range), output(arr.size());
for (int i=0; i<arr.size(); i++)
    count[arr[i]-min]++;
for (int i=1; i<count.size(); i++)
    count[i] += count[i-1];
for (int i = arr.size()-1; i>=0; i--)
{
    output[ count[arr[i]-min]-1 ] = arr[i];
    count[arr[i]-min]--;
}
for (int i=0; i<arr.size(); i++)
    arr[i] = output[i];
}
void printArray(vector<int> &arr)
{
    for (int i=0; i<arr.size(); i++
        cout << arr[i] << " ";
}
int main()
{
    vector<int> arr = {-5,-10,0,-3,8,5,-1,10};
    countSort(arr);
    printArray(arr);
    return 0;
}
```

Output :

stdin copy

```
Standard input is empty
```

stdout copy

```
-10 -5 -3 -1 0 5 8 10
```

Q.3 Write a program to implement quick sort algorithm.

Ans. Logic:

i) Select a pivot: Let it be the first element in the array.

ii) We now partition the number set (or array) around the pivot point.

iii) Once the partitioning is done, the group of elements smaller than the pivot is placed on its left, while larger elements appear on its right.

iv) Now, sort these two groups using recursion.

Time Complexity: O(nlogn)

Code:

```
#include <bits/stdc++.h>
using namespace std;
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    for (j = low; j <= high - 1; 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);
    }
}
void printArray(int arr[], int size)
{
    int i;
    for(i=0; i<size; i++)
        cout<<arr[i]<<" ";
    cout<<endl;
}
int main()
{
    int arr[] = {15,27,1,3,49,10};
    int n = sizeof(arr)/sizeof(arr[0]);
    quicksort(arr, 0, n-1);
    cout<<"Sorted Array: \n";
```

```
    printArray(arr, n);
    return 0;
}
```

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
Sorted array:
1 3 10 15 27 49
```

4. Write a program to implement merge sort.

Ans. Logic: We use the divide and conquer approach.

- Divide: If q is a halfway point between p & r, then we can split the array into two subarrays A[p ... q] & A[q+1 ... r]
- Conquer: We recursively sort both these subarrays. If base case hasn't been reached, split the array again.
- Combine: When the sub-arrays have been solved, we now combine the parts to form a completed sorted array.

Time Complexity : $O(n \log n)$

Code:

```
#include<bits/stdc++.h>
using namespace std;
void merge(int arr[], int l, int m, int r)
{
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[m + 1 + i];
    int i = 0, j = 0, k = l;
```

```
while (i < n1 && j < n2)
{
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
}
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}
while (j < n2){
    arr[k] = R[j];
    j++;
    k++;
}
}
void mergeSort(int arr[], int l, int r){
    if(l >= r) {
        return;
    }
    int m = l + (r - l)/2;
    mergeSort(arr, l, m);
```

```
    mergeSort(arr, m+1, r);
    merge(arr, l, m, r);
}
void printArray(int A[], int size)
{
    for(i=0; i<size; i++)
    { cout<<A[i]<<" "; }
}
int main()
{
    int arr[] = {13,10,12,89,1,3,5,1,56};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    cout<<"Given array is \n";
    printArray(arr, arr_size);
    mergeSort(arr, 0, arr_size-1);
    cout<<"Sorted array is \n";
    return 0;
}
```

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
Given array is
13 10 12 89 1 3 5 1 56
Sorted array is
1 1 3 5 10 12 13 56 89
```

Q.5 Write a program to implement bubble sort.

Ans. Logic: We have the following algorithm for bubble sort:

```
bubblesort(array)
    for i ← 1 to index of last unsorted element - 1
        if left element > right element
            swap left and right element
    end bubble sort.
```

Time Complexity: $O(n^2)$

Code:

```
#include<bits/stdc++.h>
using namespace std;
void swap(int *x, int *y)
{
    int temp = i *x;
    *x = *y
    *y = temp;
}
void bubbleSort(int arr[]
{
    int i, j;
    bool swapped;
    for(int i=0; i< n-1; i++)
    {
        swapped = false;
        for(j=0; j< n-1-i; j++)
        {
            if(arr[j] > arr[j+1])
            {
                swap(&arr[j], &arr[j+1])
                swapped = true;
```

```
        }
    }
    if (swapped == false)
        break;
  }
}
void printArray (int arr[], int size)
{
    for(int i=0 ; i < size ; i++)
        cout << arr[i] << " ";
}

int main()
{
    int arr[] = { 64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
Sorted array:
11 12 22 25 34 64 90 n
```

2.6 Write a program to implement insertion sort.

Ans. Logic: We have the following algorithm

```
For j = 2 to A.length()
    key = A[j]
    i = j - 1
    while i > 0 && A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[j+1] = key
```

Time Complexity: $O(n^2)$

Code:

```
#include <bits/stdc++.h>
using namespace std;
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for(i=1; i<n; i++){
    key = arr[i];
    j = i-1;
    while(j>=0 && arr[j] > key)
    {
        arr[j+1] = arr[j];
        j = j-1;
    }
    arr[j+1] = key;
```

```
    }
}
void printArray(int arr[], int n)
{
    for(int i=0; i<n; i++)
        cout<<arr[i]<<" ";
}
int main()
{
    int arr[] = {12,11,13,5,6};
    int n = sizeof(arr) / sizeof(arr[0]);
    insertionSort(arr, n);
    printArray(arr,n);
    return 0;
}
```

Output :

stdin copy

```
Standard input is empty
```

stdout copy

```
5 6 11 12 13
```

Q-7 Write a program to implement huffman coding

Ans- Logic: We create a priority queue Q consisting of every unique characters & sort them in ascending order of frequences. Now, for all the unique characters:

- Create a new node
- extract minimum value & assign it to left child of new node.
- extract min. value from Q & assign it to right child of new node.
- Calculate sum of these values & assign it to new node.
- insert new node into tree

return root node

Time Complexity: $O(n \log n)$

Code:

```
#include<bits/stdc++.h>
using namespace std;
struct min heap
{
    char data;
    unsigned freq;
    minheap *left, *right;
    minheap(char data, unsigned freq)
    {
        this -> data = data
        left = right = null
        this -> freq = freq;
};
```

```
struct compare {
    bool operator()(minheap* l, minheap* r)
    { return (l->freq, r->freq);}};
void print(struct minheap * root, string str)
{ if(! root) return;
  if(root->data != '$')
  {
    cout << root->data << ":" << str << endl;
     print(root->left, str + "0");
     print(root->right, str + "1");
  }
}
void huffman(char data[], int freq[], int size)
{
  struct minheap* left, * right, * top;
  priority_queue<minheap*, vector<minheap*>, compare> min;
  for(int i=0; i< size; i++)
      min.push(new minheap(data[i], freq[i]));
  while(min.size()!=1)
  {
     left = min.top();
      min.pop();
      right = min.top();
        min.pop();
    top = new minheap('$', left->freq + right->freq);
    top->left = left;
    top->right = right;
```

```
        min.push(top);
    }
    print(min.top(), " ");
}
int main()
{
    char arr[] = {'a','b','c','d','e','f'};
    int freq[] = {5,9,12,13,16,45};
    int size = sizeof(arr)/sizeof(arr[0]);
    huffman(arr, freq, size);
    return 0;
}
```

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Q:8 Write a program to implement job - sequencing problem.

Ans. Logic: Sort all jobs in decreasing order of profit. Now, iterate through them and find a time slot i such that the slot is empty & i < deadline & i is greatest. Put the job in & mark it as filled. If no such slot exists, ignore the job.

Time Complexity: $O(n^2)$

Source Code:

```
#include <bits/stdc++.h>
using namespace std;
struct job {
 char id;
 int dl;
 int profit;
};
bool comparison( job a, job b)
   return( a.profit > b.profit);
void print( job arr[], int n)
{
    sort (arr, arr +n , comparison);
    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].dl) - 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])
            cout << 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]);
    cout << "Following is maximum profit sequence of jobs \n";
    print(arr, n);
    return 0;
}
```

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
a c e
```

Q 9 WAP to implement Warshall's algorithm.

Ans. Logic: This algorithm take two nodes at a time and tries to find the optimal distance between them via points that occur in their path. If a shorter distance is possible, the distance is updated in the adjacency matrix. It uses dynamic programming approach to calculate shortest distance paths by using already stored values.

Time Complexity: $O(n^3)$

Code:

```
#include <bits/stdc++.h>
using namespace std;
#define V 5
#define INF 99999
void printSolution(int dist[][V]);
void floydWarshall(int graph[][V])
{
    int dist[V][V], i, j, k;
    for(i=0; i<V; i++)
        for(j=0; j<V; j++)
    dist[i][j] = graph[i][j];
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]
                dis[i][j]=dist[i][k]+dist[k][j]
        }
}
}
```

```
    printSolution(dist);
}
void printSolution(int dist[][V])
{
    cout << "The following matrix shows the shortest distance
         between every pair of vertices \n";
    for (i=0; i<V; i++)
    {
        for (int j=0; j<V; j++){
            if (dist[i][j]==INF)
                cout << "INF" << "  ";
            else
                cout << dist[i][j] << "  ";
        }
        cout << endl;
    }
}
int main()
{
        int graph[V][V] = {{ 0, INF, 2, INF, INF },
                           { 10, 0, INF, INF, 3 },
                           { INF, 1, 0, 7, INF },
                           { INF, 5, INF, 0, INF },
                           { INF, INF, INF, 11, 0 }};

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

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
The following matrix shows the shortest distances between every pair of vertices
0    3   2   9   6
10   0   12  14  3
11   1   0   7   4
15   5   17  0   8
26   16  28  11  0
```

Q.10 WAP to implement Dijkstra's algorithm.

Ans. ~~Code~~ Logic: Create Vertex set q
for each vertex v in graph
dist[v] = ∞
add v to q (build heap)
dist[source] = 0
while q is not empty
u = exact minimum [q]
for each neighbour v of u
relax (u, v);

Time Complexity: O(E log V)

Code:
```
#include <bits/stdc++.h>
using namespace std;
# define V 5
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[])
{
    cout << " Vertex    Distance From Source \n";
    for (int i = 0; i < V; i ++)
        cout << i << "    " << dist[i] << endl;
}
```

```
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]= distance[u] + graph[u][v];
    }
    printSolution (dist);
}
int main ()
{
    int graph[V][V] = {{ 0, 10, 2, 0, 0},
                       {10, 0, 1, 5, 3},
                       { 2, 1, 0, 7, 0},
                       { 0, 5, 7, 0, 11},
                       { 0, 3, 0, 11, 0}};
```

dist dijkstra (graph, 0);

return 0;

}

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
Vertex      Distance from Source
0                   0
1                   3
2                   2
3                   8
4                   6
```

Q11 Write a program to implement Prism's algorithm.

Ans. Logic: i) Initialize the minimum Spanning tree with a vertex chosen at random.

ii) Find all the edges that connect the tree to new vertices, find the minimum & add it to the tree.

iii) Keep repeating step 2 untill we get a minimum spanning tree.

Time Complexity: O(E log V)

Code:

```
#include<bits/stdc++.h>
using namespace std;
#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;
}
void primMST(int parent[], int graph[V][V])
{
    cout << "Edge \t Weight \n";
    for(int i = 1; i < V; i++)
    cout << parent[i] << " - " << i << " \t"
```

```
        << graph[i][parent[i]] << " \n";
}
void primMST(int graph[V][V])
{
    int parent[V], 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] = {{0,10,2,0,0},
                       {10,0,0,5,3},
                       {2,0,0,7,0},
                       {0,5,7,0,11},
                       {0,3,0,11,0}};
    primMST(graph);
    return 0;
}
Output;
```

stdin copy

```
Standard input is empty
```

stdout copy

```
Edge    Weight
3 - 1   5
0 - 2   2
2 - 3   7
1 - 4   3
```

Q.12 WAP to implement Kruskkal's algorithm.

Ans. Logic: i) Sort all the edges from low weight to high.

ii) Take the edge with lowest weight & add it to spanning tree as long as it does not create a cycle.

iii) Keep adding edges until all vertices are included.

Time Complexity: O(E log E)

Code:

```
#include <bits/stdc++.h>
using namespace std;
class Edge {
  public:
      int src, dest, weight;
};
class Graph {
  public:
     int V, E;
     Edge* edge;
};
Graph * createGraph (int V, int E)
{
     Graph * graph = new Graph;
     graph -> V = V;
     graph -> E = E;
     graph -> edge = new Edge [E];
```

```
    return graph;
}
class subset {
public:
    int parent, rank;
};
int find (subset subsets[], int i)
{
    if (subsets[i].parent != i)
        subsets[i].parent = find (subsets, subsets[i].
                                  parent);
    return subsets[i].parent;
}
void Union (subset subsets[], int x, int y)
{
    int xroot = find (subsets, x);
    int yroot = find (subsets, y);
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}
```

```
int myComp (const void * a, const void * b)
{
    Edge * a1 = (Edge *) a;
    Edge * b1 = (Edge *) b;
    return a1->weight > b1->weight;
}
void KruskalMST(Graph * graph)
{
    int V = graph->V;
    Edge result[V];
    int e = 0, i = 0;
    sort(graph->edge, graph->E, sizeof(graph->edge[0]),
                                myComp);
    subset * subsets = new subset[(V * sizeof(subset))];
    for (int v=0; v<V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }
    while (e < V-1 && i < graph->E)
    {
        Edge next_edge = graph->egde[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y) {
            result[e++] = next_edge;
```

```
        Union (subsets, x, y);
    }
}
cout << "Following are the edges in the constructed MST
int minimumCost = 0;
for (i = 0; i < e; ++i)
{
    cout << result[i].src << " -- " << result[i].dest <<
            result[i].weight << endl;
    minimumCost = minimumCost + result[i].weight;
}
cout << "Minimum Cost Spanning Tree: " << minimumCost
                                    << endl;

}
int main()
{
    int V = 5;
    int E = 6;
    Graph* graph = createGraph(V, E);
    graph->edge[0].src = 0
    graph->edge[0].dist = 1
    graph->weight[0].weight = 10;
    graph->edge[1].src = 0;
    graph->edge[1].dist = 2;
    graph->edge[1].weight = 2;
    graph->edge[2].src = 1;
    graph->edge[2].dist = 3;
    graph->edge[2].weight = 5;
```

```
graph → edge [3]. src = 1;
graph → edge [3]. dist =
graph → edge [3]. weight =
graph → edge [4]. src =
graph → edge [4]. dist =
graph → edge [4]. weight =
Kruskal MST(graph);
return 0;
}
```

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
Following are the edges in the constructed MST
0 -- 2 == 2
1 -- 4 == 3
1 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 20
```

Q.13 WAP to implement BFS & DFS.

Ans. Logic : (BFS)

i) Start by putting any one of the graph's vertices at the back of a queue.

ii) Take the front item of that ~~vertex~~ queue & add it to the visited list.

iii) Created a list of ~~all~~ that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

iv) Repeat Steps 2 & 3 until queue is empty.

Time Complexity : $O(V+E)$

Code :

```
#include <bits/stdc++.h>
using namespace std;
class Graph
{
    int V;
    list<int> *adj;
public:
    Graph(int V)
    {
        this->V = V;
        adj = new list<int>[V];
    }
    void addEdge(int v, int w)
    {
        adj[v].push_back(w);
```

```
void BFS(int s)
{
    bool * visited = new bool[V];
    for (int i=0; i<V; i++)
        visited[i] = false;
    list <int> queue;
    visited[s] = true;
    queue.push_back(s);
    list <int>:: iterator i;
    while (!queue.empty())
    {
        s. queue.front();
        cout << s << " ";
        queue.pop_front();
        for (i = adj[s].begin(); i != adj[s].end(); i++)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }}}};
int main()
{
    Graph g(4);
    g.addEgde(0,1);
```

```
    g.addEdge(0,2);
    g.addEdge(1,2);
    g.addEdge(2,0);
    g.addEdge(2,3);
    g.addEdge(3,3);
    cout << "Following is Breadth First Traversal (starting from
                                    vertex 2) \n";
    g.BFS(2);
    return 0;
}
```

DFS

Logic: i) Start by putting any one of the graph's vertices on top of a stack.

ii) Take the top item of the stack & add it to the visited list.

iii) Create a list of that vertices adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

iv) Keep repeating steps 2 & 3 until stack is empty.

Time Complexity: $O(E+V)$

Code:

```
#include<bits/stdc++.h>
using namespace std;
class Graph {
```

```
public:
        map<int, bool> visited;
        map<int, list<int>> adj;
        void addEdge(int v, int w)
            adj[v].push_back(w);
        void DFS(int v)
        {
            visited[v] = true;
            cout << v << " ";
            list<int>::iterator i;
            for(i = adj.begin(); i != adj[v].end(); i++)
                if(!visited[*i])
                    DFS(*i);
        }
};
int main()
{
    Graph g;
    g.addEdge(0,1);
    g.addEdge(0,9);
    g.addEdge(1,2);
    g.addEdge(2,0);
    g.addEdge(2,3);
    g.addEdge(9,3);
    cout << "Following is DFS (starting from vertex 2)\n";
    g.DFS(2);
    return 0;
}
```

Output:

Success #stdin #stdout 0s 5344KB comments (0)

stdin copy

```
Standard input is empty
```

stdout copy

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

stdin copy

```
Standard input is empty
```

stdout copy

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 9 3
```

Q.14 WAP to implement travelling salesperson problem.

Ans Logic: $C(\{1\}, 1) = 0$

for s = 2 to n do

for all subsets $S \in \{1, 2, 3, \ldots, n\}$ of size s and containing 1

$C(S, 1) = \infty$

for all $j \in S$ and $j \neq 1$

$C(S, j) = \min\{C(S - \{j\}, i) + d(i, j) \text{ for } i \in S \& i \neq j\}$

return $\min_j C(\{1, 2, 3, \ldots, n\}, j) + d(j, i)$

Time Complexity: $O(2^n \cdot n^2)$

Code:

```
#include <bits/stdc++.h>
using namespace std;
#define V 4
void tsp(int graph[][V], vector<bool> &v, int
    currPos, int n, int count, int cost, int &ans)
{
    if (count == n && graph[currPos][0])
    {
        ans = min(ans, cost + graph[currPos][0]);
        return;
    }
    for (int i = 0; i < n; i++)
    {
        if (!v[i] && graph[currPos][i])
        {
```

```
                v[i] = true;
                tsp(graph, v, i, n, count + 1, cost + graph[curlos][
                                                    ans);
                v[i] = false;
            }
        }
    };
    int main ()
    {
      int n = 4;
      int graph [][V] = {{0, 10, 15, 20}, {10, 0, 35, 25},
                {15, 35, 0, 30}, {20, 25, 30, 0}};
      vector <bool> v(n);
      for (int i = 0; i < n; i++)
          v[i] = false;
      v[0] = true;
      int ans = INT_MAX;
      tsp (graph, v, 0, n, 1, 0, ans);
      cout << ans;
      return 0;
    }
```

Output:

stdin copy

```
Standard input is empty
```

stdout copy

```
80
```

Q.15 N- queens problem

Ans. Logic : i) Start in the leftmost column

ii) if all queens are placed
return true.

iii) Try all rows in the current Column, do the following
a) if queen can be placed safely in this row then mark this [row, column] as part of the solution & recursively check if placing queen here leads to a solution.
b) if placing the queen in [row, column] leads to a solution then return true.
c) if placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) & go to step (a) to try the other rows.

iv) If all rows have been tried & nothing worked, return false & trigger ~~loop~~ backtracking.

Time Complexity: $O(2^n)$

Code:
```
#define N 5
#include <bits/stdc++.h>
using namespace std;
void print(int board[N][N]){
for(int i=0; i<N; i++){
    for(int j=0; j<N; j++)
        cout << board[i][j] << endl;
}}
bool issafe(int board[N][N], int rows, int col)
{ int i, j;
```

```
for (i = 0; i < col; i++)
    if (board[row][i])
        return false;
for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
    if (board[i][j])
        return false;
for (i = row, j = col; j >= 0 && i < N; i++, j--)
    if (board[i][j])
        return false;
return true;
}
bool solveNqutil (int board[N][N], int col)
{
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++) {
        if (issafe (board, i, col)) {
            board[i][col] = 1;
            if (solveNqutil(board, col+1))
                return true;
            board[i][col] = 0;
        }
    }
    return false;
}
bool solveNq()
{
    int board[N][N] = {{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},
                       {0,0,0,0,0},{0,0,0,0,0}};
    if (solveNqutil(board, 0) == false {
        cout << "Solution does not exist"; return false;
    }
    print(board); return true; }
int main
{ solveNq(); return 0; }
```

stdin copy

```
Standard input is empty
```

stdout copy

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