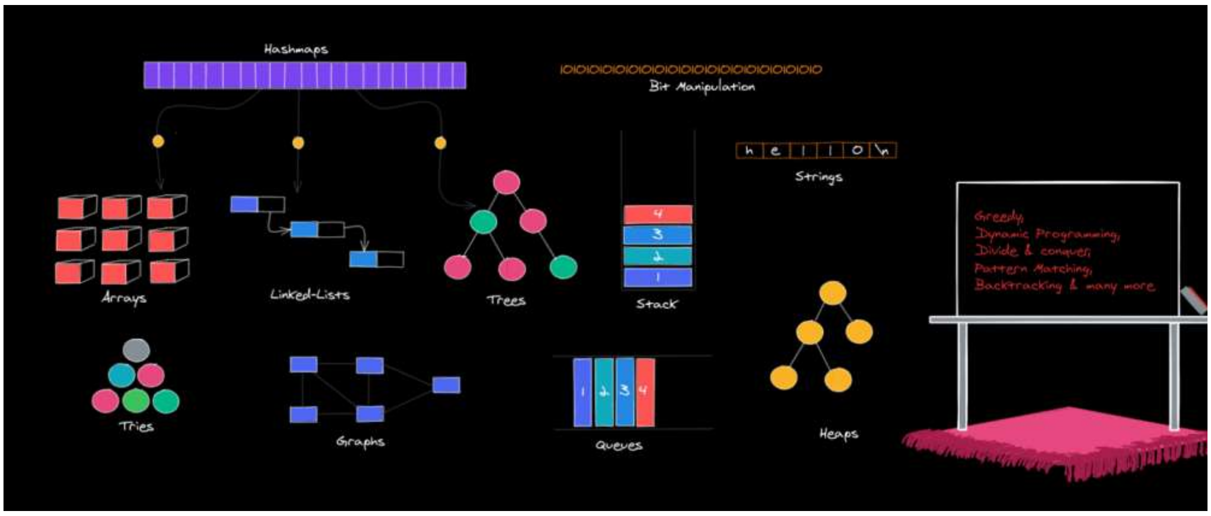
Data Structures and Algorithms Practical File


Hashmaps
Bit Manipulation
Arrays
Linked-Lists
Trees
Stack
Strings
Greedy, Dynamic Programming, Divide & conquer, Pattern Matching, Backtracking & many more
Tries
Graphs
Queues
Heaps

Submitted to: Mrs. Archana Gahlaut

Submitted by: Amit Pal

Roll No. : 88015

Course: B.Sc.(Hons)

Computer Science 2nd Year

Q1)(i) Implement Insertion Sort (The program should report the number of comparisons)

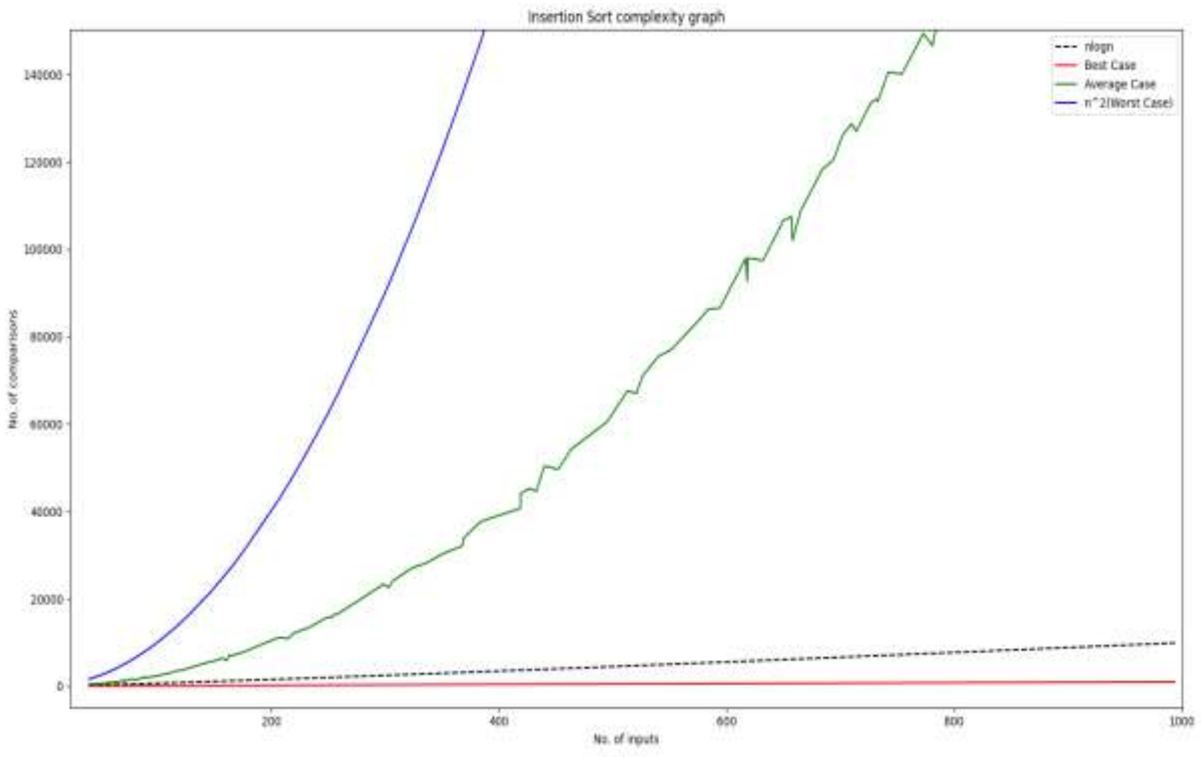
Test run the algorithm on 100 different inputs of sizes varying from 30 to 1000. Count the number of comparisons and draw the graph. Compare it with a graph of "nlogn"

Code:-

```cpp
#include <bits/stdc++.h>
using namespace std;

int insertionSort(int arr[], int size)
{
        int cnt2 = 0;
        for (int i = 1; i < size; i++)
        {
                int key = arr[i];
                int j = i - 1;
                while (j >= 0 && arr[j] > key)
                {
                        cnt2++;
                        arr[j + 1] = arr[j];
                        j = j - 1;
                }
                arr[j + 1] = key;
        }

        return cnt2 + 1;
}

int main()
{
        int size;
        ofstream fout("MyExcel.csv");
        fout << "Size"
             << ","
             << "Comparisons" << endl;
        for (int i = 0; i < 100; i++)
        {
                size = rand() % 971 + 30;
                int Array[size] = {0};
                for (int j = 0; j < size; j++)
                {
                        Array[j] = rand() % 10000;
                }

                // cout << "Unsorted array" << endl;
                // for (int i = 0; i < size; i++)
                // {
```

```
                //          cout << Array[i] << "  ";
                // }

                fout << size << "," << insertionSort(Array, size) << endl;

                // cout << "Sorted array" << endl;
                // for (int i = 0; i < size; i++)
                // {
                //          cout << Array[i] << "  ";
                // }
        }

        return 0;
}
```

Output graph:-



Zoomed-in view of graph

Q1)(ii) Implement Merge Sort (The program should report the number of comparisons)

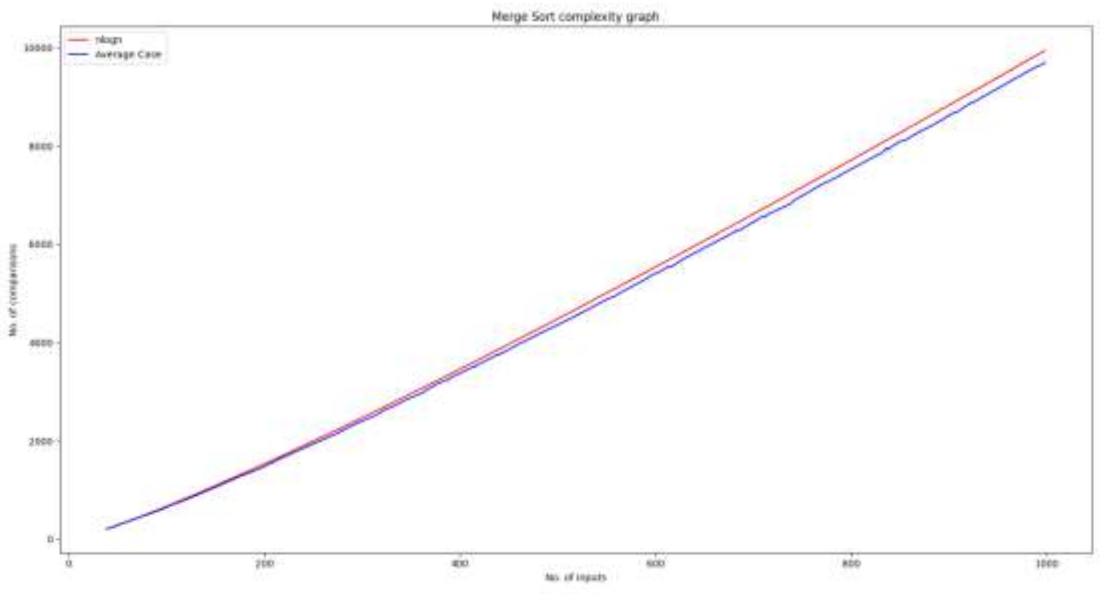
Test run the algorithm on 100 different inputs of sizes varying from 30 to 1000. Count the number of comparisons and draw the graph. Compare it with a graph of “nlogn”

Code:-

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;
int mcount = 0;

int merge(int arr[], int l, int mid, int r)
{
        int c = 0;
        int n1 = mid - l + 1;
        int n2 = r - mid;

        int arr1[n1];
        int arr2[n2];

        for (int i = 0; i < n1; i++)
        {
                arr1[i] = arr[l + i];
        }

        for (int i = 0; i < n2; i++)
        {
                arr2[i] = arr[mid + 1 + i];
        }

        int i = 0;
        int j = 0;
        int k = l;

        while (i < n1 && j < n2)
        {
                c++;
                if (arr1[i] < arr2[j])
                {
                        arr[k] = arr1[i];
                        i++;
                        k++;
                }
                else
                {
                        arr[k] = arr2[j];
                        j++;
                        k++;
                }
        }

        while (i < n1)
        {
                arr[k] = arr1[i];
                i++;
                k++;
        }

        while (j < n2)
        {
                arr[k] = arr2[j];
                j++;
                k++;
        }

        return c;
}

void mergeSort(int arr[], int l, int r)
{
        mcount++;
        if (l < r)
        {
                int mid = (l + r) / 2;
                mergeSort(arr, l, mid);
                mergeSort(arr, mid + 1, r);

                mcount = mcount + merge(arr, l, mid, r) - 1;
        }
}

int main()
{
        int size;
        ofstream fout("MyExcel.csv");
        fout << "Size"
             << ","
             << "Comparisons" << endl;
        srand(time(0));
        for (int i = 0; i < 100; i++)
        {
                size = rand() % 971 + 30;
                int Array[size] = {0};
                for (int j = 0; j < size; j++)
                {
                        Array[j] = rand() % 10000;
                }

                // cout << "Unsorted array" << endl;
                // for (int i = 0; i < size; i++)
                // {
                //         cout << Array[i] << "  ";
                // }
                mergeSort(Array, 0, size);
                fout << size << "," << mcount << endl;
                mcount = 0;

                // cout << "Sorted array" << endl;
                // for (int i = 0; i < size; i++)
                // {
                //         cout << Array[i] << "  ";
                // }
        }

        return 0;
}
```

Output graph:-



Q2) Implement Heap Sort (The program should report the number of comparisons)

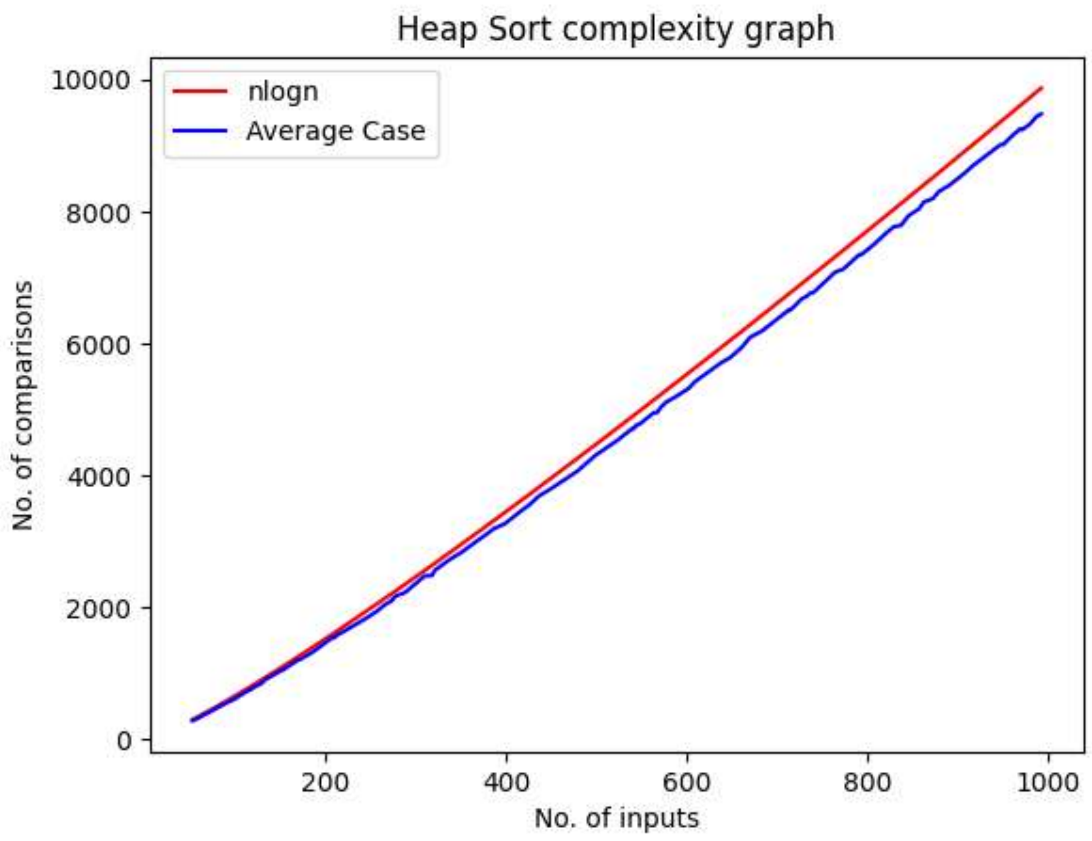
Test run the algorithm on 100 different inputs of sizes varying from 30 to 1000. Count the number of comparisons and draw the graph. Compare it with a graph of “nlogn”

Code:-

```
#include <bits/stdc++.h>
using namespace std;
int cnt = 0;

void heapify(int arr[], int n, int i)
{
        cnt++;
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && arr[left] > arr[largest])
                largest = left;

        if (right < n && arr[right] > arr[largest])
                largest = right;

        if (largest != i)
        {
                swap(arr[i], arr[largest]);
                heapify(arr, n, largest);
        }
}

void heapsort(int arr[], int n)
{
        // Building max-heap
        for (int i = n / 2 - 1; i >= 0; i--)
                heapify(arr, n, i);

        // heap sort
        for (int i = n - 1; i >= 0; i--)
        {
                swap(arr[0], arr[i]);

                // heapify root element
                heapify(arr, i, 0);
        }
}

int main()
{
        int size;
        ofstream fout("MyExcel.csv");
        fout << "Size"
             << ","
             << "Comparisons" << endl;
        srand(time(0));
        for (int i = 0; i < 100; i++)
        {
                size = rand() % 971 + 30;
                int Array[size] = {0};
                for (int j = 0; j < size; j++)
                {
                        Array[j] = rand() % 10000;
```

```cpp
                }

                // cout << "Unsorted array" << endl;
                // for (int i = 0; i < size; i++)
                // {
                //          cout << Array[i] << "  ";
                // }

                heapsort(Array, size);
                fout << size << "," << cnt << endl;
                cnt = 0;

                // cout << "Sorted array" << endl;
                // for (int i = 0; i < size; i++)
                // {
                //          cout << Array[i] << "  ";
                // }
        }

        return 0;
}
```

Output:-



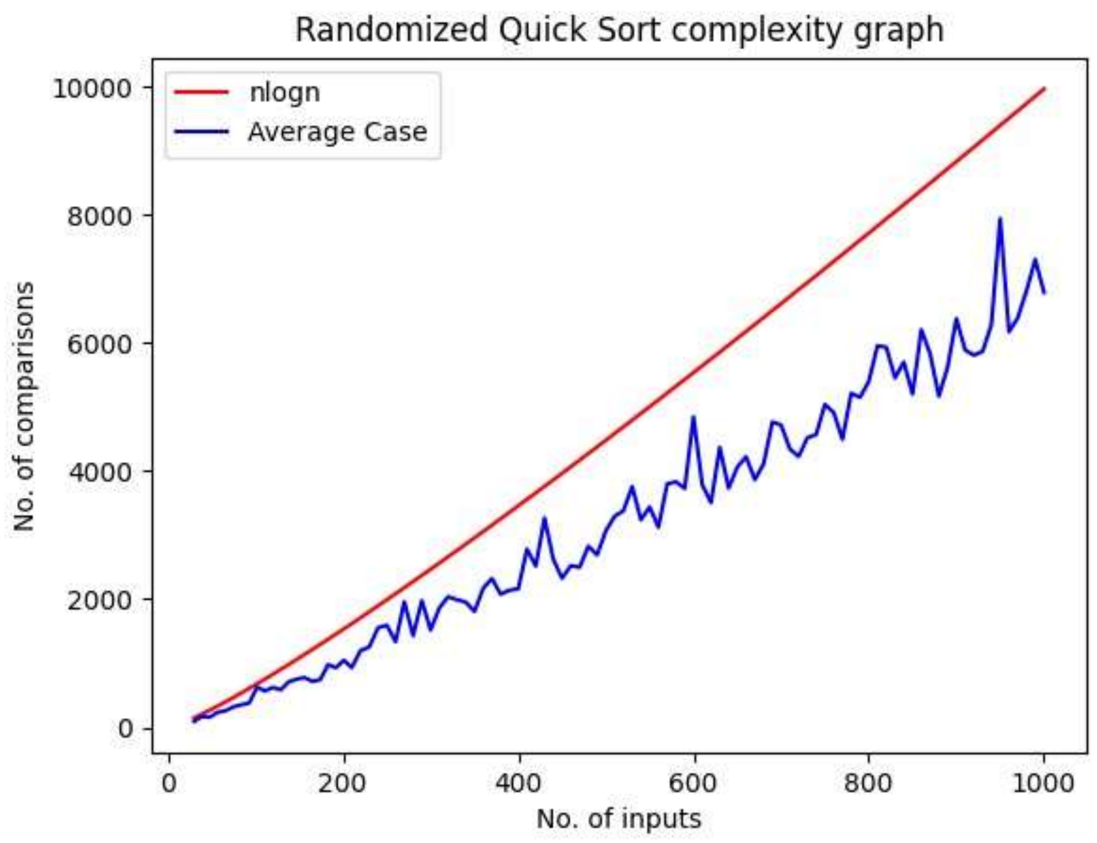
Q3) Implement Randomized Quick Sort (The program should report the number of comparisons)

Test run the algorithm on 100 different inputs of sizes varying from 30 to 1000. Count the number of comparisons and draw the graph. Compare it with a graph of “nlogn”

Code:-

```
#include <bits/stdc++.h>
using namespace std;
int cnt = 0;

int Partition(int arr[], int low, int high)
{
        int pivot = arr[high];
        int i = (low - 1);

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

int partitionRandom(int arr[], int low, int high)
{
        srand(time(NULL));
        int random = low + rand() % (high - low);
        swap(arr[random], arr[high]);
        cnt++;

        return Partition(arr, low, high);
}

void quickSort(int arr[], int low, int high)
{
        if (low < high)
        {
                int pi = partitionRandom(arr, low, high);

                quickSort(arr, low, pi - 1);
                quickSort(arr, pi + 1, high);
        }
}

int main()
{
        int size;
        ofstream fout("MyExcel.csv");
        fout << "Size"
             << ","
             << "Comparisons" << endl;
        // srand(time(0));
        size = 30;
        for (int i = 0; i < 100; i++)
        {
                int Array[size] = {0};
                for (int j = 0; j < size; j++)
                {
                        Array[j] = rand() % 10000;
                }

                // cout << "Unsorted array" << endl;
                // for (int i = 0; i < size; i++)
                // {
                //         cout << Array[i] << "  ";
                // }

                quickSort(Array, 0, size - 1);
                fout << size << "," << cnt << endl;
                // cout << "Size: " << size << " cnt: " << cnt << endl;
                cnt = 0;

                // cout << "Sorted array" << endl;
                // for (int i = 0; i < size; i++)
                // {
                //         cout << Array[i] << "  ";
                // }
                if (i < 20)
                        size += 9;
                else
                        size += 10;
        }

        return 0;
}
```

Output:-

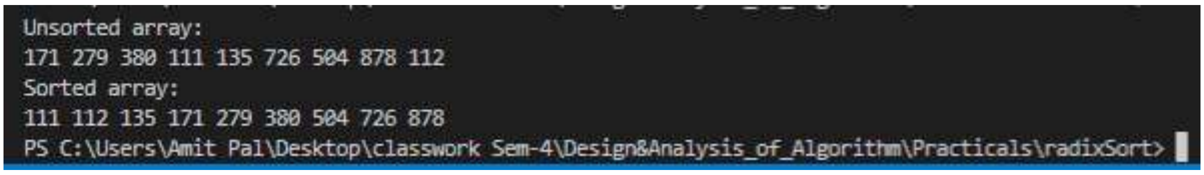


Q4) Implement Radix Sort.

Code:

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

int getMax(int a[], int n)
{
        int max = a[1];
        for (int i = 0; i < n; i++)
        {
                if (a[i] > max)
                        max = a[i];
        }

        return max;
}

void countingSort(int a[], int n, int place)
{
        int output[n + 1];
        int count[10] = {0};
        // Calculating count of elements
        for (int i = 0; i < n; i++)
                count[(a[i] / place) % 10]++;

        // Calculating cumulative frequency
        for (int i = 1; i < 10; i++)
                count[i] += count[i - 1];

        // Place the elements in sorted order
        for (int i = n - 1; i >= 0; i--)
        {
                output[count[(a[i] / place) % 10] - 1] = a[i];
                count[(a[i] / place) % 10]--;
        }

        for (int i = 0; i < n; i++)
                a[i] = output[i];
}

void radixSort(int a[], int n)
{
        int max = getMax(a, n);

        for (int place = 1; max / place > 0; place *= 10)
                countingSort(a, n, place);
}

int main()
{
        int a[] = {171, 279, 380, 111, 135, 726, 504, 878, 112};

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

        cout << "Unsorted array: \n";
        for (int i = 0; i < n; i++)
                cout << a[i] << " ";
        cout << "\n";

        radixSort(a, n);

        cout << "Sorted array: \n";
        for (int i = 0; i < n; i++)
                cout << a[i] << " ";
        cout << "\n";
}
```

Output:



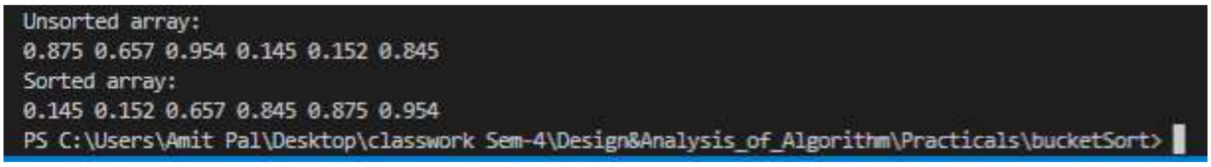
```
Unsorted array:
171 279 380 111 135 726 504 878 112
Sorted array:
111 112 135 171 279 380 504 726 878
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals\radixSort>
```

Q5) Implement Bucket Sort.

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

void bucketSort(float arr[], int n)
{
        // creating n empty buckets
        vector<float> b[n];

        // putting elements in bucket
        for (int i = 0; i < n; i++)
        {
                int bi = n * arr[i];
                b[bi].push_back(arr[i]);
        }

        // sorting each individual bucket
        for (int i = 0; i < n; i++)
                sort(b[i].begin(), b[i].end());

        // concatinating all buckets into an array
        int index = 0;
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < b[i].size(); j++)
                        arr[index++] = b[i][j];
        }
}

int main()
{
        float a[] = {0.875, 0.657, 0.954, 0.145, 0.152, 0.845};

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

        cout << "Unsorted array: \n";
        for (int i = 0; i < n; i++)
                cout << a[i] << " ";
        cout << "\n";
        bucketSort(a, n);

        cout << "Sorted array: \n";
        for (int i = 0; i < n; i++)
                cout << a[i] << " ";
        cout << "\n";
}
```

Output:



```
Unsorted array:
0.875 0.657 0.954 0.145 0.152 0.845
Sorted array:
0.145 0.152 0.657 0.845 0.875 0.954
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals\bucketSort>
```

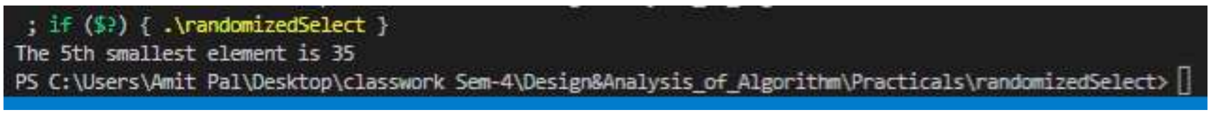
Q6) Implement randomized select.

Code:

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

int Partition(int arr[], int low, int high)
{
        int pivot = arr[high];
        int i = (low - 1);

        for (int 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);
}

int partitionRandom(int arr[], int low, int high)
{
        srand(time(NULL));
        int random = low + rand() % (high - low);
        swap(arr[random], arr[high]);

        return Partition(arr, low, high);
}

int randomSelection(int arr[], int low, int high, int index)
```

```
{
        if (low == high)
                return arr[low];

        int q = partitionRandom(arr, low, high);
        int k = q - low + 1;

        if (k == index)
                return arr[q];

        if (index < k)
                return randomSelection(arr, low, q - 1, index);
        else
                return randomSelection(arr, q + 1, high, index - k);
}

int main()
{
        int arr[] = {12, 15, 74, 65, 12, 35, 89, 15, 36};

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

        cout << "The 5th smallest element is " << randomSelection(arr, 0, n -
1, 5);
        return 0;
}
```

Output:



```
 ; if ($?) { .\randomizedSelect }
The 5th smallest element is 35
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals\randomizedSelect> 
```

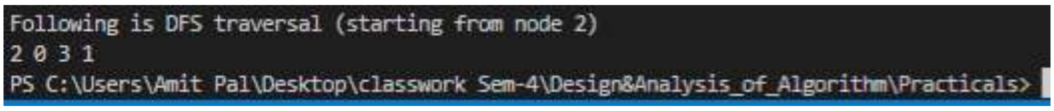
Q7) Implement Breadth first search in graph.

Code:

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

class Graph
{
        int V;

        list<int> *adj;

public:
        Graph(int V)
```

```cpp
        {
                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.addEdge(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 DFS traversal (starting from node 2)\n";

        g.BFS(2);

        return 0;
}
```

Output:



```
Following is DFS traversal (starting from node 2)
2 0 3 1
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals>
```

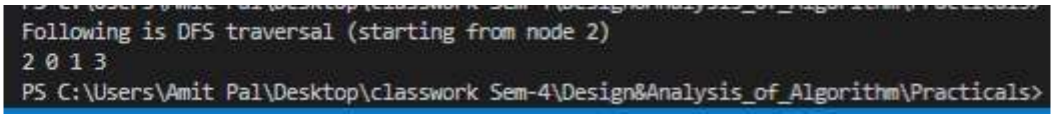
Q8) Implement Depth-first search.

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[v].begin(); i != adj[v].end(); ++i)
                {
                        if (!visited[*i])
                                DFS(*i);
                }
        }
};
```

```cpp
int main()
{
        Graph g;
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(1, 3);
        g.addEdge(3, 3);

        cout << "Following is DFS traversal (starting from node 2)\n";

        g.DFS(2);

        return 0;
}
```

Output:



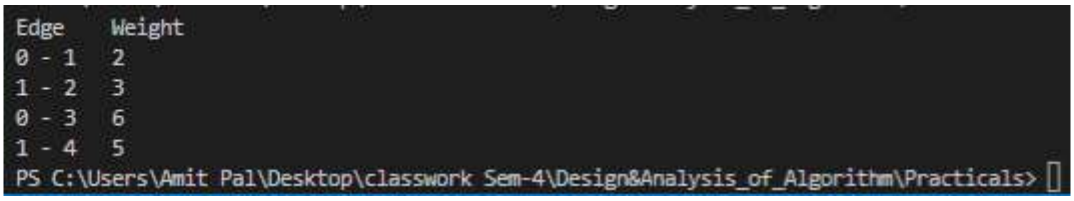
```
Following is DFS traversal (starting from node 2)
2 0 1 3
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals>
```

Q9) Write a program to determine minimum spanning tree

(i) Prim's Algo:
Code:

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

void primsMST(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}};
        primsMST(graph);

        return 0;
}
```

Output:



```
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals> 
```

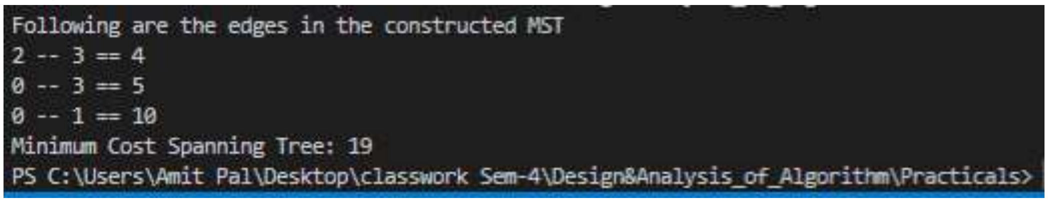
(ii) Krushkal's algo:

Code:

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

class DSU
{
        int *parent;
        int *rank;

public:
        DSU(int n)
        {
                parent = new int[n];
                rank = new int[n];

                for (int i = 0; i < n; i++)
                {
                        parent[i] = -1;
                        rank[i] = 1;
                }
        }

        // Find function
        int find(int i)
        {
                if (parent[i] == -1)
                        return i;

                return parent[i] = find(parent[i]);
        }
        // union function
        void unite(int x, int y)
        {
                int s1 = find(x);
                int s2 = find(y);

                if (s1 != s2)
                {
                        if (rank[s1] < rank[s2])
                        {
                                parent[s1] = s2;
                                rank[s2] += rank[s1];
                        }
                        else
                        {
                                parent[s2] = s1;
                                rank[s1] += rank[s2];
                        }
                }
        }
};

class Graph
{
        vector<vector<int>> edgelist;
        int V;

public:
        Graph(int V) { this->V = V; }

        void addEdge(int x, int y, int w)
        {
                edgelist.push_back({w, x, y});
        }

        void kruskals_mst()
        {

                sort(edgelist.begin(), edgelist.end());

                DSU s(V);
                int ans = 0;
                cout << "Following are the edges in the "
                        "constructed MST"
                     << endl;
                for (auto edge : edgelist)
                {
                        int w = edge[0];
                        int x = edge[1];
                        int y = edge[2];
```

```
            if (s.find(x) != s.find(y))
            {
                s.unite(x, y);
                ans += w;
                cout << x << " -- " << y << " == " << w
                    << endl;
            }
        }
        cout << "Minimum Cost Spanning Tree: " << ans;
    }
};
int main()
{
    Graph g(4);
    g.addEdge(0, 1, 10);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);
    g.addEdge(2, 0, 6);
    g.addEdge(0, 3, 5);

    g.kruskals_mst();
    return 0;
}
```

Output:



```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals>
```

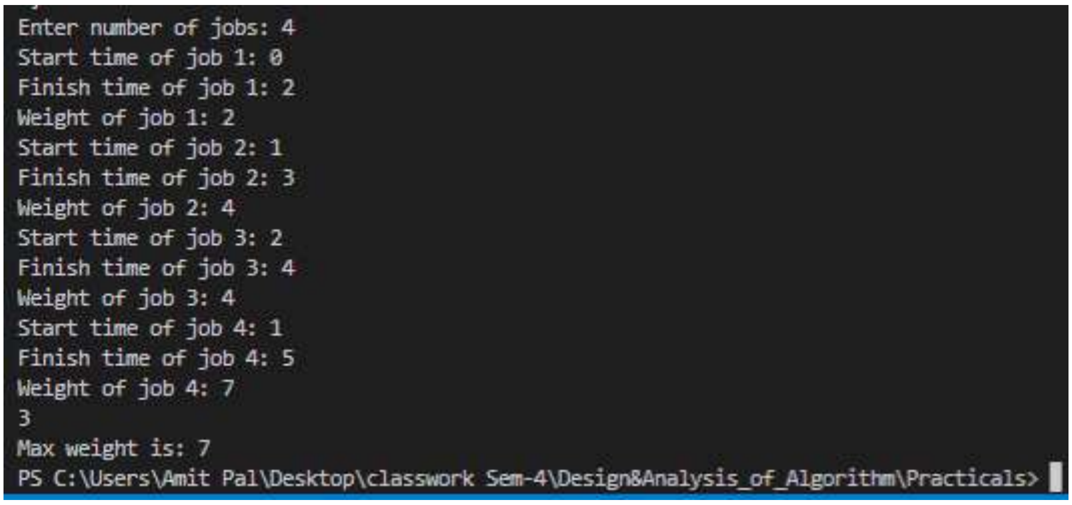
Q10) Write a program to solve weighted interval scheduling problem.

Code:

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

struct Job
{
    int start, finish, weight;
};

bool jobComparator(Job s1, Job s2)
{
    return (s1.finish < s2.finish);
}
```

```cpp
int getMax(int arr[], int n)
{
        int max = 0;
        for (int i = 0; i < n; i++)
        {
                if (arr[i] > arr[max])
                        max = i;
        }

        return max;
}

void weightedShedule(Job arr[], int n)
{
        int wtarr[n];
        for (int i = 0; i < n; i++)
        {
                wtarr[i] = arr[i].weight;
        }

        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < i; j++)
                {
                        if (arr[i].finish <= arr[j].start)
                                wtarr[i] += wtarr[j];
                }
        }

        int maxIndex = getMax(wtarr, n);

        vector<int> schedule;

        for (int i = 0; i < n; i++)
        {
                if (arr[i].finish <= arr[maxIndex].start)
                        schedule.push_back(i);
        }
        schedule.push_back(maxIndex);

        for (int i = 0; i < schedule.size(); i++)
        {
                cout << schedule[i] << " ";
        }
        cout<<"\n";

        for (int i = 0; i < schedule.size(); i++)
        {
                for (int j = i + 1; j < schedule.size(); j++)
                {
                        if (arr[schedule[i]].finish > arr[schedule[j]].start)
                                if (arr[schedule[i]].weight >
arr[schedule[j]].weight)
                                {
                                        vector<int>::iterator it =
schedule.begin() + j;
                                        schedule.erase(it);
                                }
                                else
                                {
                                        vector<int>::iterator it =
schedule.begin() + i;
                                        schedule.erase(it);
                                }
                }
        }

        for (int i = 0; i < schedule.size(); i++)
        {
                cout << schedule[i] << " ";
        }
        cout<<"\n";

        cout << "Max weight is: " << arr[maxIndex].weight;
}

int main()
{
        int a;
        cout << "Enter number of jobs: ";
        cin >> a;

        Job arr[a];

        for (int i = 0; i < a; i++)
        {
                cout << "Start time of job " << (i + 1) << ": ";
                cin >> arr[i].start;
                cout << "Finish time of job " << (i + 1) << ": ";
                cin >> arr[i].finish;
                cout << "Weight of job " << (i + 1) << ": ";
                cin >> arr[i].weight;
        }

        sort(arr, arr + a, jobComparator);
        weightedShedule(arr, a);
}
```

Output:



```
Enter number of jobs: 4
Start time of job 1: 0
Finish time of job 1: 2
Weight of job 1: 2
Start time of job 2: 1
Finish time of job 2: 3
Weight of job 2: 4
Start time of job 3: 2
Finish time of job 3: 4
Weight of job 3: 4
Start time of job 4: 1
Finish time of job 4: 5
Weight of job 4: 7
3
Max weight is: 7
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals>
```

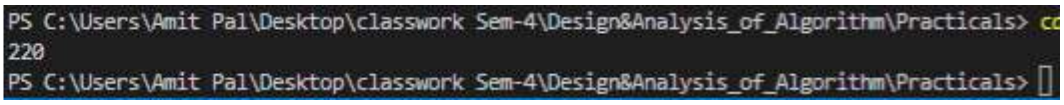
Q11) Write a program to solve 0-1 knapsack.

Code:

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

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 val[] = {60, 100, 120};
        int wt[] = {10, 20, 30};
        int W = 50;
        int n = sizeof(val) / sizeof(val[0]);
        cout << knapsack(W, wt, val, n);
        return 0;
}
```

Output:



```
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals> co
220
PS C:\Users\Amit Pal\Desktop\classwork Sem-4\Design&Analysis_of_Algorithm\Practicals> 
```