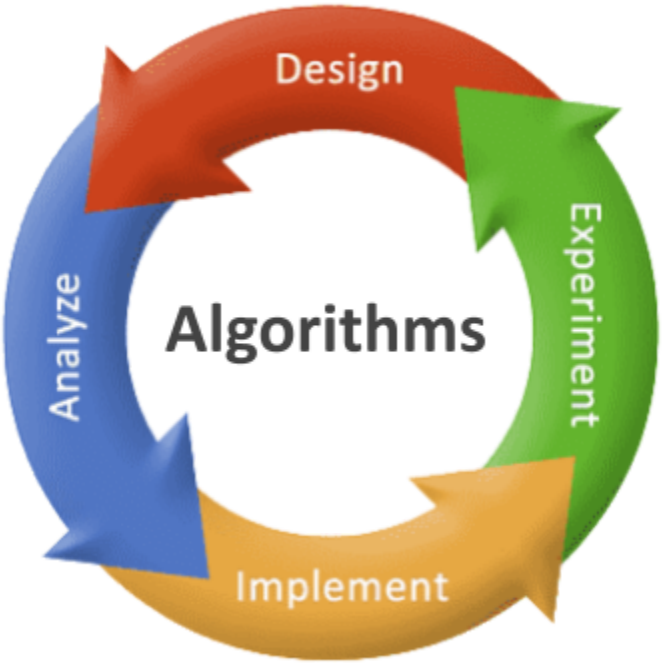
Design & Analysis of Algorithms



Practical File

Name : **Avinash Gautam**

Course : **B.Sc. (Hons.) Computer Science**

Exam Roll No. : **20020570009**

For Basic Sorting Algorithms :
Algo.hpp

```
#ifndef _algo_hpp_
#define _algo_hpp_

#pragma once

#include <iostream>
#include <cstdlib>

using namespace std;

namespace algo
{
  template <class T>
  void printArray(T *array, const uint &len)
  {
     cout << "\n { ";
     for (uint i = 0; i < len - 1; i++)
        cout << array[i] << ", ";
     cout << array[len - 1] << " }\n";
  }

  template <class T>
  void swap(T *a, T *b)
  {
     T tmp = *a;
```

```
    *a = *b;
    *b = tmp;
}


/*      Insertion Sort      */
template <class Tp, class _Comp = less<Tp>>
uint insertionSort(Tp *array, const uint &len,
                                    const _Comp &cmp = _Comp())
{
    uint count = 0;
    Tp key;
    if (len > 1)
        for (int i = 1, j = 0; i < len; i++)
        {
            j = i - 1;
            key = array[i];
            while ( j >= 0 && cmp ( key, array[ j ] ))
            {
                array[ j + 1] = array[ j ];
                j--;
                count++;
            }
            array[ j + 1] = key;
            count++;
        }
    return count;
}
```

```
/*     Merge Sort     */
template <typename T, class _comp>
uint merge(T *arr, const uint &l, const uint &m,
                              const uint &r, const _comp &cmp)
{
   uint n1 = m - l + 1, n2 = r - m, i, j, k;
   T L[n1], R[n2];

   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[ j] = arr[m + 1 + j];

   i =  j = 0;
   k = l;

   while (i < n1 && j < n2)
   {
      if (cmp(L[i], R[j]))
         arr[k] = L[i++];

      else
         arr[k] = R[j++];

      k++;
   }
```

```
    while (i < n1)
        arr[k++] = L[i++];

    while (j < n2)
        arr[k++] = R[ j++];

    return k - l - 1;
}

template <typename T, class _comp = less<T>>
uint mergeSort(T *arr, const uint &begin, const uint &end,
                                    const _comp &cmp = _comp())
{
    if (begin < end)
    {
        uint mid = (begin + end) / 2;
        return mergeSort(arr, begin, mid, cmp) +
                    mergeSort(arr, mid + 1, end, cmp) +
                            merge(arr, begin, mid, end, cmp);
    }
    return 1;
}
```

```
/*      Heap Sort      */
template <class T, class _Comp>
void heapify(T array[], const int &i, const uint &len,
                              const _Comp &cmp, uint &count)
{
   uint left, right, top = i;
   left = 2 * i + 1;
   right = 2 * i + 2;

   if (left < len && cmp(array[top], array[left]))
      count++, top = left;

   if (right < len && cmp(array[top], array[right]))
      count++, top = right;

   if (top != i)
   {
      swap(array + i, array + top);
      count++;
      heapify(array, top, len, cmp, count);
   }
}
template <class Tp, class _Comp = less<Tp>>
uint heapSort(Tp *array, const uint &len,
                                 const _Comp &cmp = _Comp())
{
   uint count = 0; // Counter
```

```
    // Build a Heap
    for (int i = len / 2 - 1; i >= 0; i--)
        heapify(array, i, len, cmp, count);

    // One by one extract an element from heap
    for (uint i = len - 1; i > 0; i--)
    {
        // Move current root to end
        swap(array, array + i);

        // call heapify on the reduced heap
        heapify(array, 0, i, cmp, count);
    }
    return count;
}

/*          Quick Sort          */
template <class T, class _Comp>
int partition(T *array, const int &start, const int &end,
                                        const _Comp &cmp)
{
    T x = array[end];
    int i = start;

    for (int j = start; j < end; j++)
        if (cmp(array[j], x))
        {
```

```
            swap(array + j, array + i);
            i++;
        }
    swap(&array[i], array + end);
    return i;
}
template <class T, class _Comp>
int random_partition(T *array, const int &start,
                                const int &end, const _Comp &cmp)
{
    srand(time(NULL));
    swap(&array[start + rand() % (end - start + 1)], &array[end]);
    return partition(array, start, end, cmp);
}
template <class T, class _Comp = less<T>>
uint random_quickSort(T *array, const int &start,
                    const int &end, const _Comp &cmp = _Comp())
{
    if (start < end)
    {
        int piv = random_partition(array, start, end, cmp);
        return (piv - start) +
              random_quickSort(array, start, piv - 1, cmp) +
              random_quickSort(array, piv + 1, end, cmp);
    }
    return 1;
}
```

```
  /*      Randomized Select      */
  template <typename tp, class _Comp = less<tp>>
  tp randomizedSelect(tp *array, const int &start,
                                 const int &end, const int &i,
                                 const _Comp &cmp = _Comp())
  {
     if (i <= 0)
        __throw_invalid_argument("Index Error : Index starts with 1
not from 0 or negative !!!");

     if (start == end)
        return array[start];

     int piv = random_partition(array, start, end, cmp);
     int k = piv - start + 1;

     if (i == k)
        return array[piv];

     else if (i < k)
        return randomizedSelect(array, start, piv - 1, i, cmp);

     else
        return randomizedSelect(array, piv + 1, end, i - k, cmp);
  }
}
#endif
```

Practical Questions

Q.1.

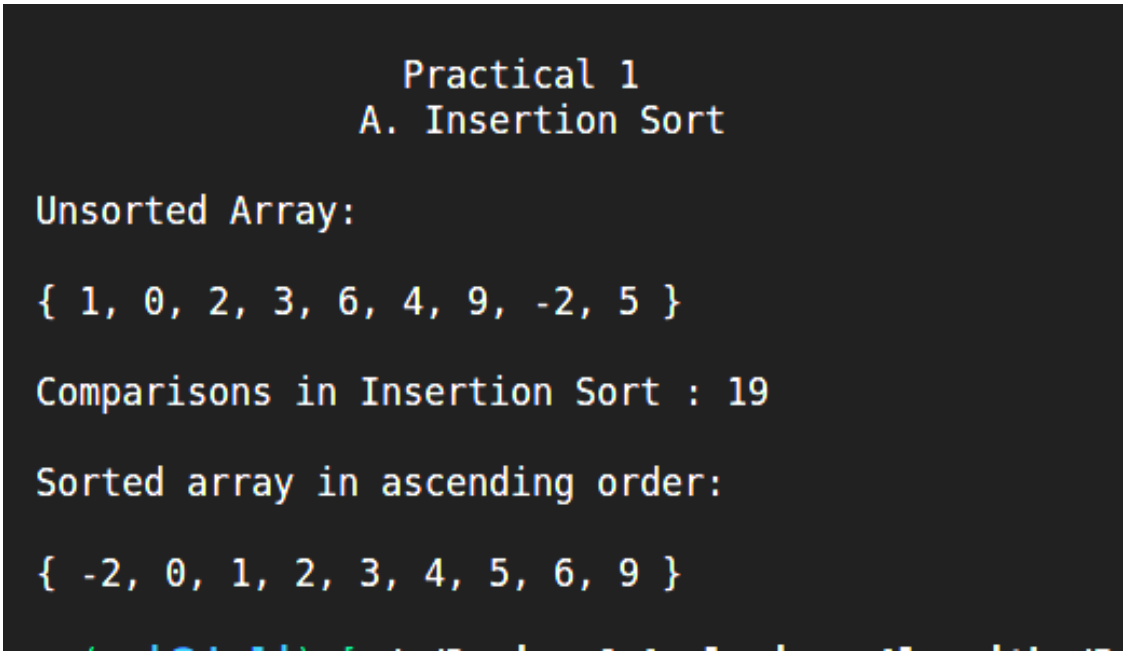
i. Implement Insertion Sort. (The program should report the number of comparisons)

Ans.

Code

```
#include "algo.hpp"
using namespace algo;

int main()
{
  int a[] = {1, 0, 2, 3, 6, 4, 9, -2, 5};
  int n = sizeof(a) / sizeof(a[0]);

  cout << "\n\t\t  Practical 1\n\t\tA. Insertion Sort\n";

  cout << "\n Unsorted Array: \n";
  printArray(a, n);

  cout << "\n Comparisons in Insertion Sort : "
       << insertionSort(a, n) << endl;

  cout << "\n Sorted array in ascending order: \n";
  printArray(a, n);
}
```

Output



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

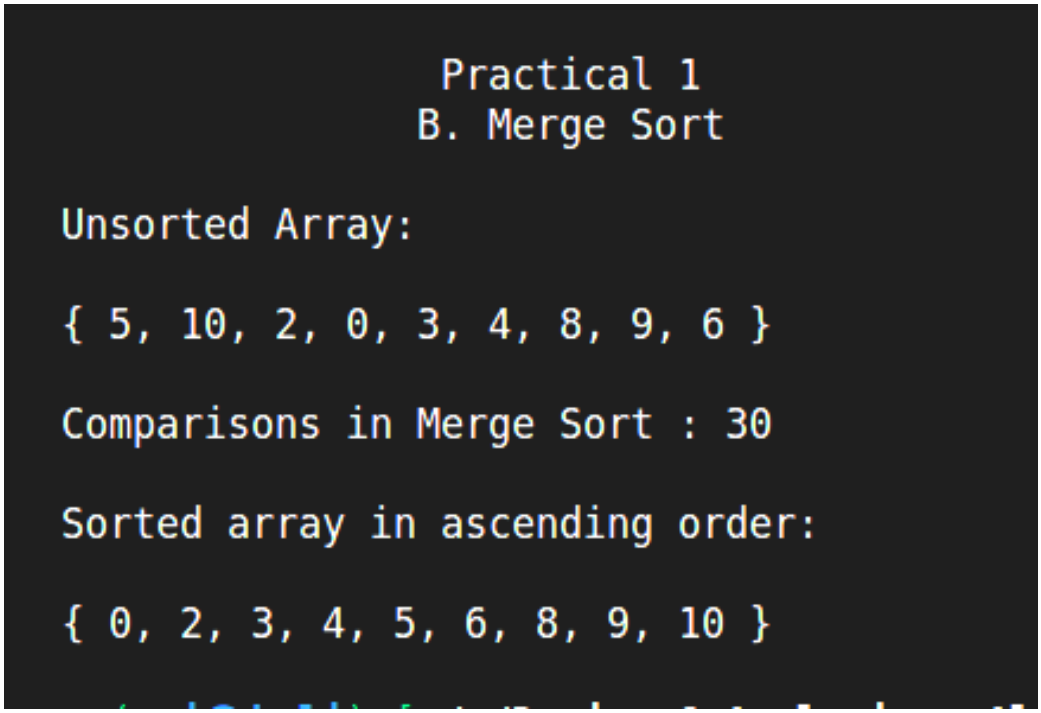
Ans.

Code

```
#include "algo.hpp"
using namespace algo;

int main()
{
  int a[] = {5, 10, 2, 0, 3, 4, 8, 9, 6};
  int n = sizeof(a) / sizeof(a[0]);


  cout << "\n\t\t Practical 1\n\t\tB. Merge Sort\n";
```

```
    cout << "\n Unsorted Array: \n";
    printArray(a, n);

    cout << "\n Comparisons in Merge Sort : "
         << mergeSort(a, 0, n - 1) << endl;

    cout << "\n Sorted array in ascending order: \n";
    printArray(a, n);

}
```

Output



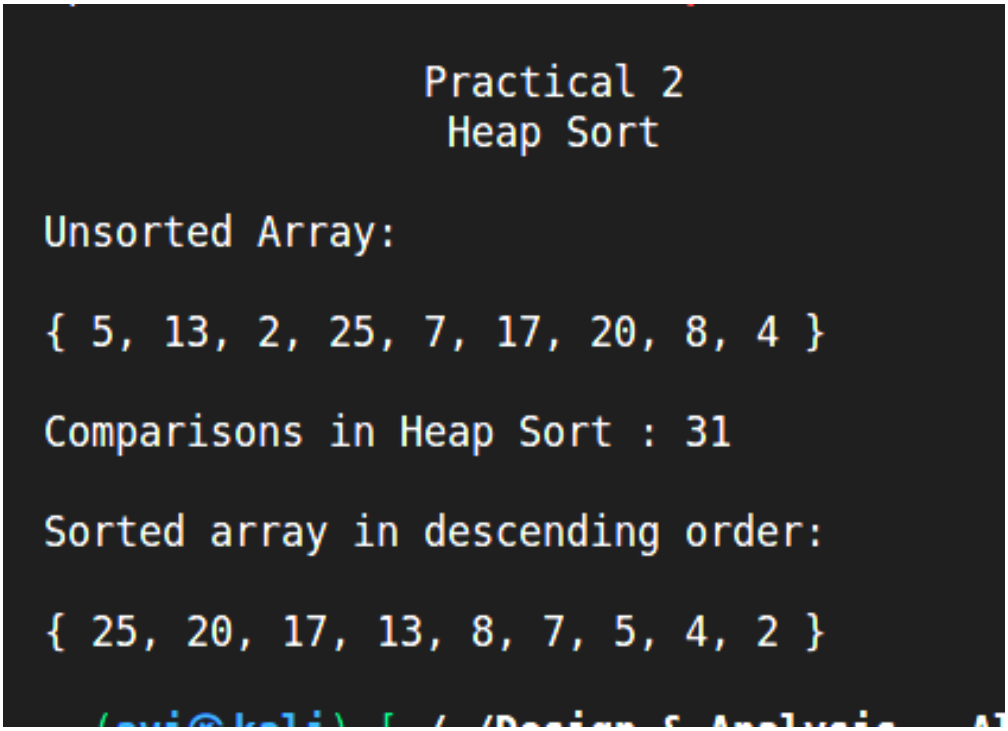
Q.2. Implement Heap Sort (The program should report the number of comparisons)

Ans.

Code

```
#include "algo.hpp"

using namespace algo;

int main()
{
  int a[] = {5,13,2,25,7,17,20,8,4};
  int n = sizeof(a)/sizeof(a[0]);

  cout << "\n\t\t Practical 2\n\t\t  Heap Sort\n";

  cout << "\n Unsorted Array: \n";
  printArray(a, n);

  cout << "\n Comparisons in Heap Sort : "
       << heapSort(a, n, greater<int>()) << endl;

  cout << "\n Sorted array in descending order: \n";
  printArray(a, n);
}
```

Output



Q.3. Implement Randomized Quick sort (The program should report the number of comparisons)

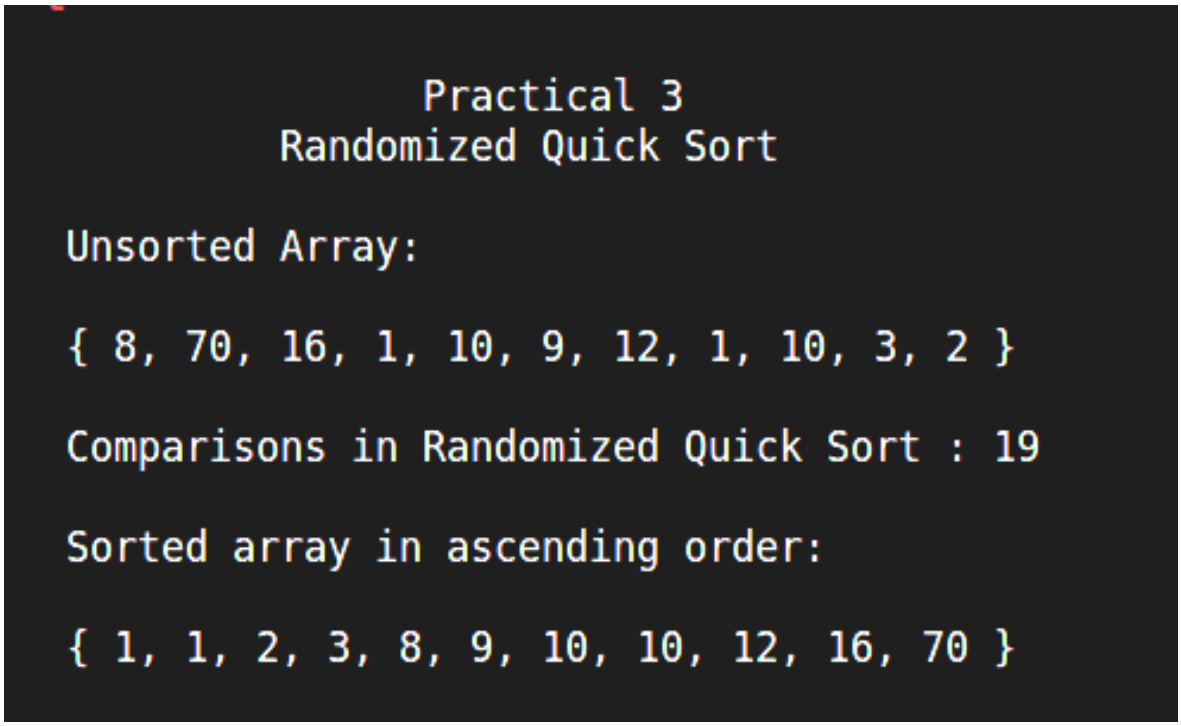
Ans.

Code

```
#include "algo.hpp"
using namespace algo;

int main()
{
  int data[] = {8, 70, 16, 1, 10, 9, 12, 1, 10, 3, 2};
  int n = sizeof(data) / sizeof(data[0]);
```

```
 cout << "\n\t\tPractical 3\n\t  Randomized Quick Sort\n";
 cout << "\n Unsorted Array: \n";
 printArray(data, n);

 cout  << "\n Comparisons in Randomized Quick Sort : "
        << random_quickSort(data, 0, n - 1) << endl;

 cout << "\n Sorted array in ascending order: \n";
 printArray(data, n);
}
```

Output



Q.4. Implement Radix Sort.

Ans.

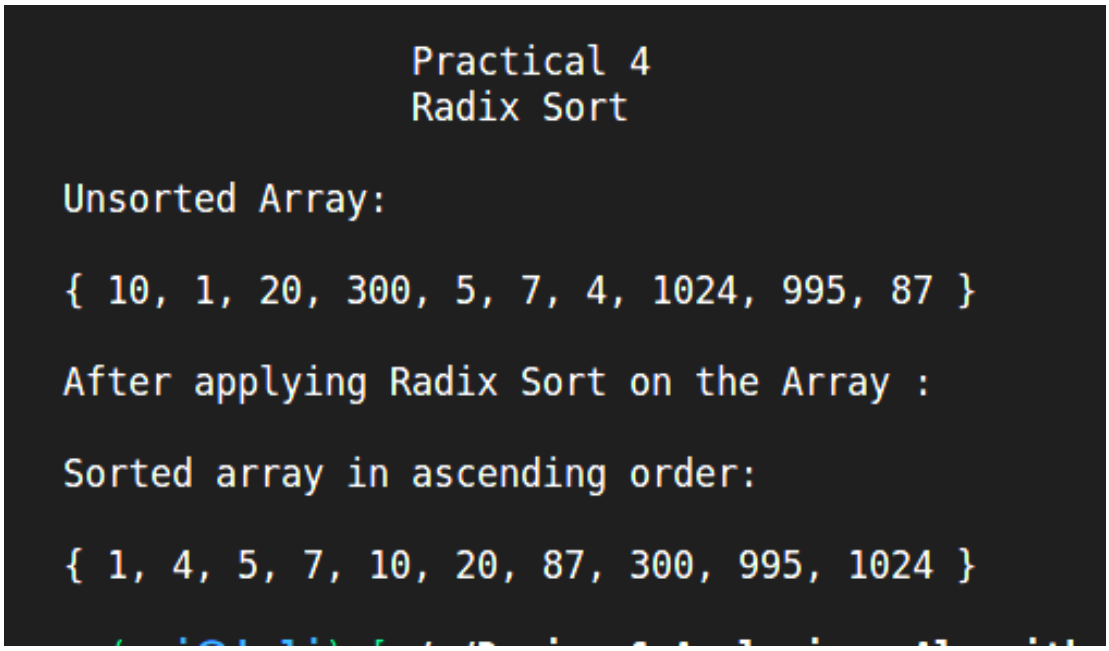
Code

```
#include "algo.hpp"
using namespace algo;

int getMax(int *array, const uint &len)
{
  int max = array[0];
  for (uint i = 1; i < len; i++)
     if (max < array[i])
        max = array[i];
  return max;
}

void countSort(int *array, const uint &len, const uint &exp)
{
  int i, *count = new int[10]{0}, *out = new int[len];

  for (i = 0; i < len; i++)
     count[(array[i] / exp) % 10]++;

  for (i = 1; i < 10; i++)
     count[i] += count[i - 1];

  for (i = len - 1; i >= 0; i--)
  {
     out[count[(array[i] / exp) % 10] - 1] = array[i];
```

```
    count[(array[i] / exp) % 10]--;
  }

  for (i = 0; i < len; i++)
    array[i] = out[i];

  delete out;
  delete count;
}

void radixSort(int *array, const uint &len)
{
  uint max = getMax(array, len);
  for (uint i = 1; (max / i) > 0; i *= 10)
    countSort(array, len, i);
}

int main()
{
  int a[] = {10, 1, 20, 300, 5, 7, 4, 1024, 995, 87};
  int n = sizeof(a) / sizeof(a[0]);

  cout << "\n\t\t Practical 4\n\t\t Radix Sort\n";

  cout << "\n Unsorted Array: \n";
  printArray(a, n);

  radixSort(a, n);
  cout << "\n After applying Radix Sort on the Array :\n";
```

```
  cout << "\n Sorted array in ascending order: \n";
  printArray(a, n);

}
```

Output



Q.5. Implement Bucket Sort.

Ans.

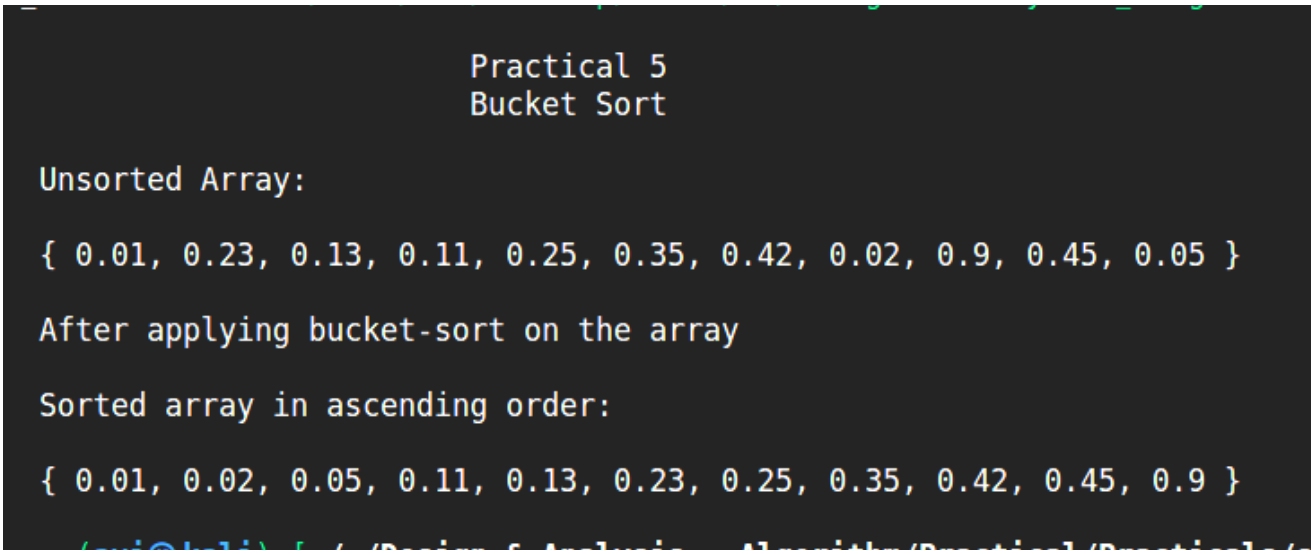
Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void printArray(float *array, const uint &len)
```

```
{
   cout << "\n { ";
   for (uint i = 0; i < len - 1; i++)
      cout << array[i] << ", ";
   cout << array[len - 1] << " }\n";
}

int main()
{
   vector<float> hash[10];
   float a[] = {0.01, 0.23, 0.13, 0.11, 0.25, 0.35, 0.42, 0.02, 0.9, 0.45, 0.05};
   int n = sizeof(a) / sizeof(a[0]), j = 0;

   cout << "\n\t\t\t Practical 5\n\t\t\t Bucket Sort\n";

   cout << "\n Unsorted Array: \n";
   printArray(a, n);

   // Bucket Sort
   for (float &i : a)
      hash[int(i * 10)].push_back(i);

   for (vector<float> &v : hash)
         sort(v.begin(), v.end());

   for (vector<float> &v : hash)
      for (float &f : v)
         a[j++] = f;
```

```
  cout << "\n After applying bucket-sort on the array\n"
       << "\n Sorted array in ascending order: \n";
  printArray(a, n);
}
```

Output



Q.6. Implement Randomized Select.

Ans.

Code

```
#include "algo.hpp"
using namespace algo;

int main()
{
  int a[] = {5, 3, 2, 7, 1, 0, 8, 4};
  int n = sizeof(a) / sizeof(a[0]), loc;
```

```
  cout << "\n\t\t Practical 6\n\t Randomized Select Algorithm\n";
  cout << "\n Unsorted Array: \n";
  printArray(a, n);

  loc = randomizedSelect(a, 0, n - 1, 1);
  cout << "\n Finding minimum element at position 1 : "
       << loc << "\n\n Now Array is : \n";
  printArray(a, n);

  loc = randomizedSelect(a, 0, n - 1, 7);
  cout << "\n Finding minimum element at position 7 : "
       << loc << "\n\n Now Array is : \n";
  printArray(a, n);
}
```

Output

```
                Practical 6
        Randomized Select Algorithm

Unsorted Array:

{ 5, 3, 2, 7, 1, 0, 8, 4 }

Finding minimum element at position 1 : 0

Now Array is :

{ 0, 1, 2, 7, 5, 3, 8, 4 }

Finding minimum element at position 7 : 7

Now Array is :

{ 0, 1, 2, 3, 4, 5, 7, 8 }
```

Q.7. Implement Breadth-First Search in a graph.

Ans.

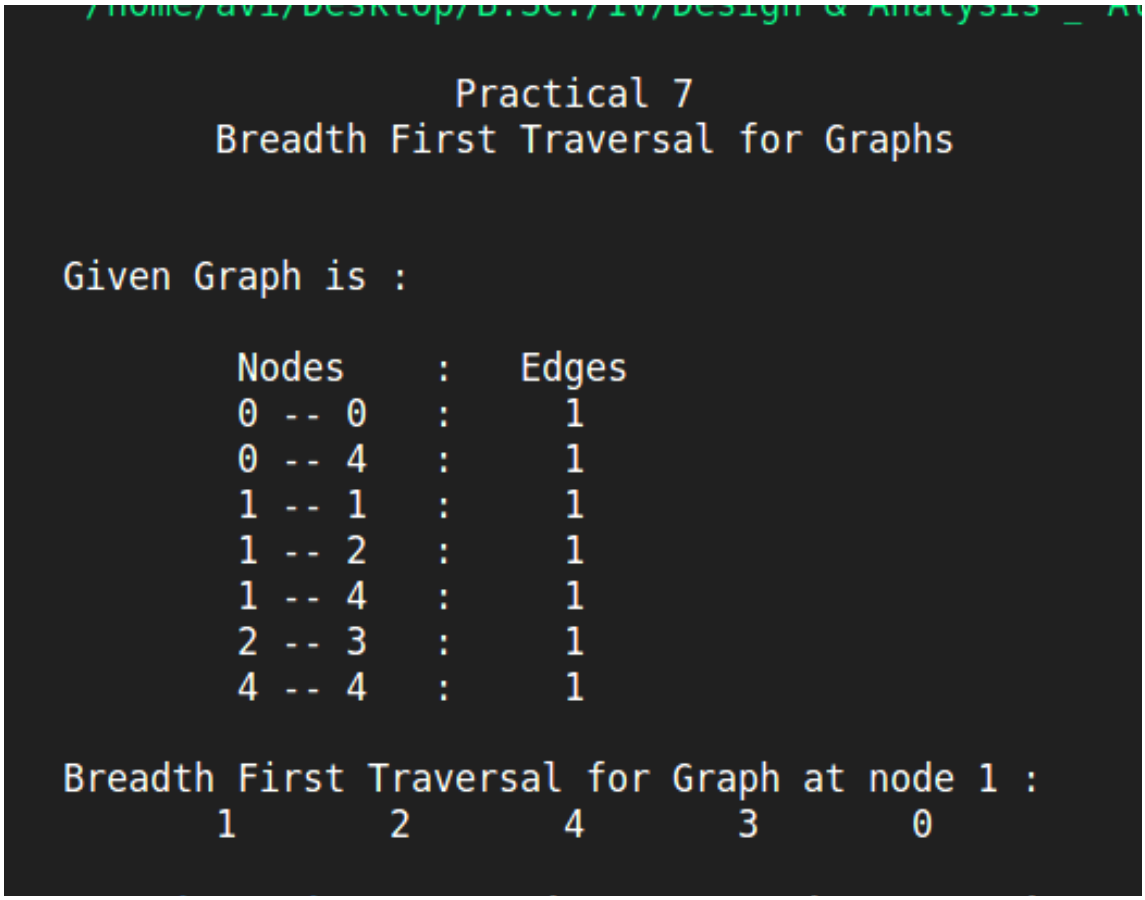
Code

```
#include <iostream>
#include <queue>

using namespace std;

#define v 5

int Graph[v][v] = {{1, 0, 0, 0, 1},
                   {0, 1, 1, 0, 1},
                   {0, 1, 0, 1, 0},
                   {0, 0, 1, 0, 0},
                   {1, 1, 0, 0, 1}};

bool visited[v] = {false};
queue<int> q;

// Breadth First Search for Graphs
void BFS(int start)
{
  if (start < 0 || start > v - 1)
     return;

  int begin;
  q.push(start);
  visited[start] = true;
```

```
  while (!q.empty())
  {
     begin = q.front();
     q.pop();

     cout << '\t' << begin;

     for (int i = 0; i < v; i++)
        if (Graph[begin][i] && !visited[i])
        {
           visited[i] = true;
           q.push(i);
        }
  }
}

int main()
{
  cout << "\n\t\t   Practical 7\n\tBreadth First Traversal for Graphs\n";

  cout << "\n\n Given Graph is :\n\n\t Nodes    :   Edges\n";
  for (int i = 0; i < v; i++)
     for (int j = 0; j < v; j++)
        if (Graph[i][j] && !(visited[j]))
        {
           visited[i] = true;
           cout << "\t " << i << " -- " << j << "   :\t" << Graph[i][j] << endl;
        }
```

```
  fill(visited, visited + v, false);
  cout << "\n Breadth First Traversal for Graph at node 1 :\n";
  BFS(1);
}
```

Output



Q.8. Implement Depth-First Search in a graph.

Ans.

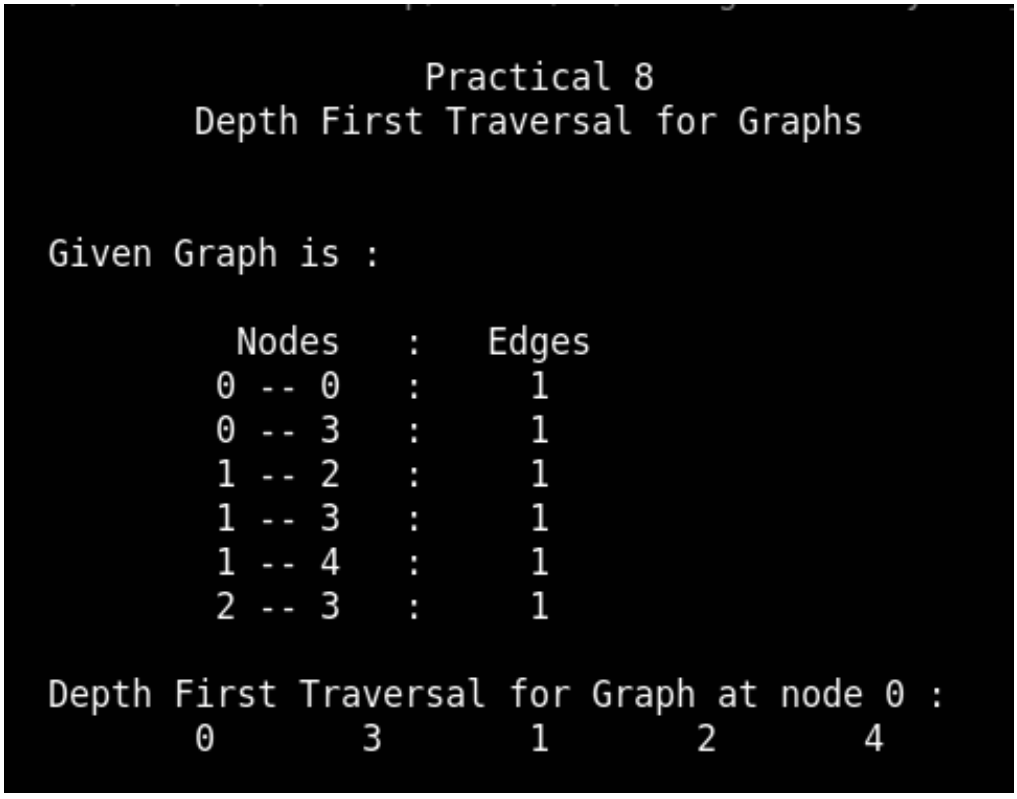
Code

```
#include <iostream>
```

```
using namespace std;

#define v 5

int Graph[v][v] = {{1, 0, 0, 1, 0},
                   {0, 0, 1, 1, 1},
                   {0, 1, 0, 1, 0},
                   {1, 1, 1, 0, 0},
                   {0, 1, 0, 0, 0}};

bool visited[v] = {false};

// Depth First Search for Graphs
void DFS(int start)
{
  if (start < 0 || start > v-1)
     return;

  visited[start] = true;
  cout << '\t' << start ;

  for(int i = 0; i < v; i++)
     if(Graph[start][i] && !visited[i])
        DFS(i);
}

int main(){
  cout << "\n\t\t   Practical 8\n\tDepth First Traversal for Graphs\n";
```

```
  cout << "\n\n Given Graph is :\n\n\t  Nodes   :   Edges\n";
  for (int i = 0; i < v; i++)
     for (int j = 0; j < v; j++)
        if (Graph[i][ j] && !(visited[ j]))
        {
           visited[i] = true;
           cout << "\t " << i << " -- " << j << "   :\t" << Graph[i][ j] << endl;
        }
  fill(visited, visited + v, false);
  cout << "\n Depth First Traversal for Graph at node 0 :\n";
  DFS(0);
}
```

Output



Q.9. Write a program to determine the minimum spanning tree of a graph using both Prims and Kruskal's algorithm.

Ans.

Code

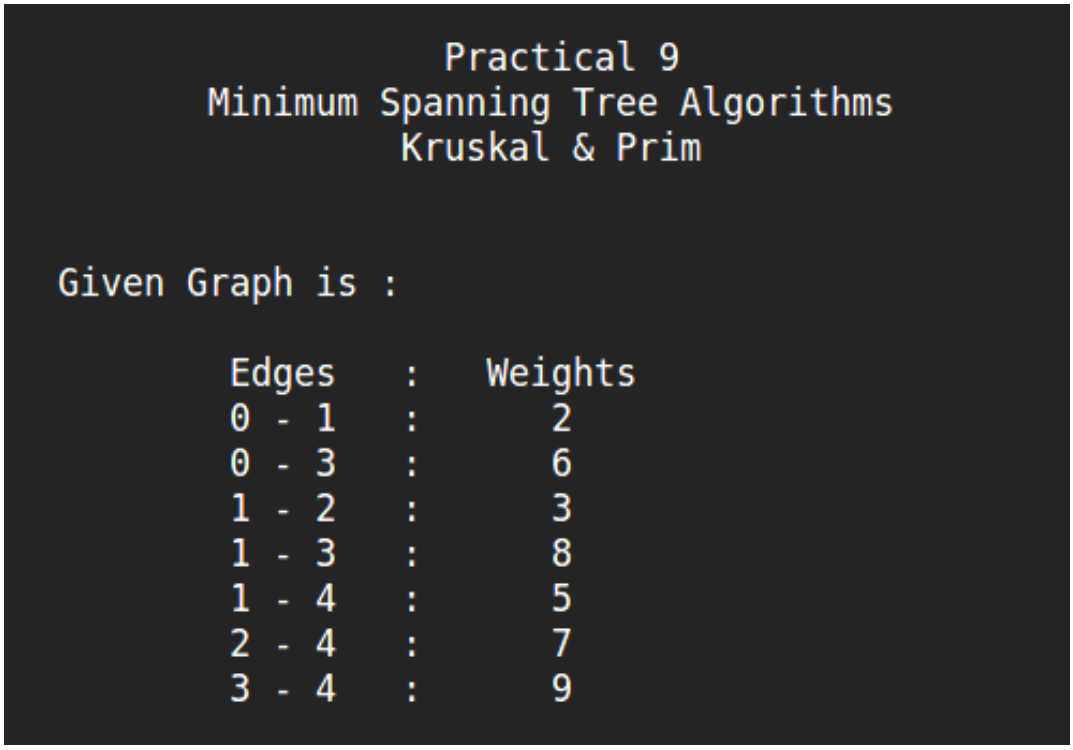
```
#include <iostream>
#include <climits>
using namespace std;

#define v 5

int parent[v];
bool visited[v] = {0};

/******      Kruskal's Algorithm      *******/
int find(int i)
{
  while (parent[i] != i)
     i = parent[i];
  return i;
}

void union1(int i, int j)
{
  int a = find(i), b = find(j);
  parent[a] = b;
}

int kruskal_mst(int cost[][v])
{
  int mincost = 0;
```

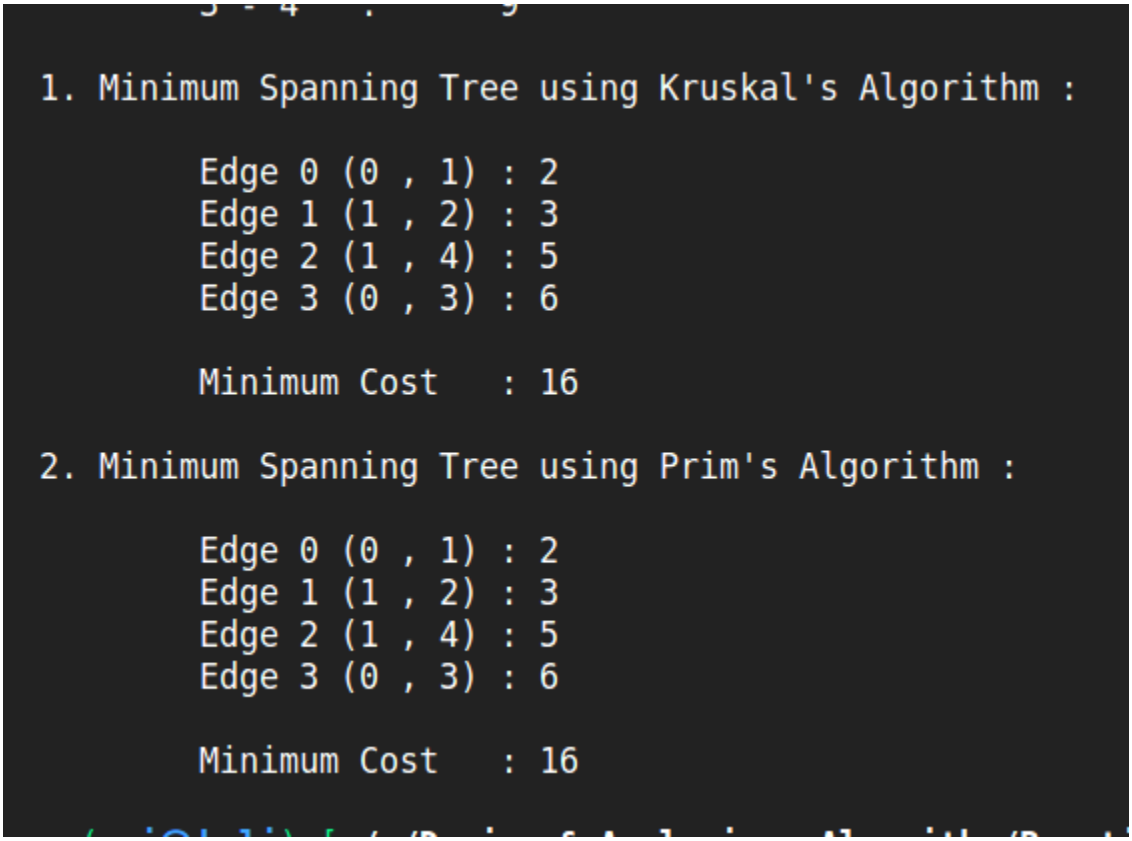
```
  int edge_count = 0;

  for (int i = 0; i < v; i++)
     parent[i] = i;

  while (edge_count < v - 1)
  {
     int min = INT_MAX, a = -1, b = -1;

     for (int i = 0; i < v; i++)
        for (int j = 0; j < v; j++)
           if (find(i) != find( j) &&
              cost[i][ j] < min)
           {
              min = cost[i][ j];
              a = i;
              b = j;
           }

     union1(a, b);
     mincost += min;
     cout << "\n\t Edge " << edge_count++ << " ("
          << a << " , " << b << ") : " << min;
  }
  return mincost;
}
/******      Prim's Algorithm      *******/
bool validEdge(int a, int b)
{
  if (a == b || visited[a] == visited[b])
       return false;
```

```
  return true;
}

int prim_mst(int cost[][v])
{
  int mincost = 0, edge_count = 0;
  fill(visited, visited + v, false);
  visited[0] = true;

  while (edge_count < v - 1)
  {
    int min = INT_MAX, a = -1, b = -1;

    for (int i = 0; i < v; i++)
      for (int j = 0; j < v; j++)
        if (cost[i][ j] < min &&
          validEdge(i, j))
        {
          min = cost[i][ j];
          a = i;
          b = j;
        }

    visited[a] = visited[b] = true;
    mincost += min;
    cout << "\n\t Edge " << edge_count++ << " ("
        << a << " , " << b << ") : " << min;
  }
  return mincost;
}
/*****************************************/
```

```
int main()
{
  int Graph[][v] = {
     {INT_MAX, 2, INT_MAX, 6, INT_MAX},
     {2, INT_MAX, 3, 8, 5},
     {INT_MAX, 3, INT_MAX, INT_MAX, 7},
     {6, 8, INT_MAX, INT_MAX, 9},
     {INT_MAX, 5, 7, 9, INT_MAX},
  };

  cout << "\n\t\t   Practical 9\n\tMinimum Spanning Tree
               Algorithms\n\t\t Kruskal & Prim\n";

  cout << "\n\n Given Graph is :\n\n\t Edges   :   Weights\n";
  for (int i = 0; i < v; i++)
     for (int j = 0; j < v; j++)
        if (Graph[i][j] != INT_MAX && !(visited[j]))
        {
           visited[i] = true;
           cout << "\t " << i << " - " << j << "   :\t" << Graph[i][j] << endl;
        }

  cout << "\n 1. Minimum Spanning Tree using Kruskal's Algorithm : \n";
  int k = kruskal_mst(Graph);
  cout << "\n\n\t Minimum Cost\t: " << k << endl;

  cout << "\n 2. Minimum Spanning Tree using Prim's Algorithm : \n";
  k = prim_mst(Graph);
  cout << "\n\n\t Minimum Cost\t: " << k << endl;
}
```

Output


Practical 9
Minimum Spanning Tree Algorithms
Kruskal & Prim
Given Graph is :
Edges : Weights
0 - 1 : 2
0 - 3 : 6
1 - 2 : 3
1 - 3 : 8
1 - 4 : 5
2 - 4 : 7
3 - 4 : 9


1. Minimum Spanning Tree using Kruskal's Algorithm :
Edge 0 (0 , 1) : 2
Edge 1 (1 , 2) : 3
Edge 2 (1 , 4) : 5
Edge 3 (0 , 3) : 6
Minimum Cost : 16
2. Minimum Spanning Tree using Prim's Algorithm :
Edge 0 (0 , 1) : 2
Edge 1 (1 , 2) : 3
Edge 2 (1 , 4) : 5
Edge 3 (0 , 3) : 6
Minimum Cost : 16

Q.10. Write a program to solve the weighted interval scheduling problem.

Ans.

Code

```
// Weighted Interval Scheduling
#include <iostream>
#include <algorithm>

using namespace std;

struct job
{
  uint start, finish, profit;

  // Constructor
  job(const uint &s = 0, const uint &f = 0, const uint &p = 0)
  {
     start = s;
     finish = f;
     profit = p;
  }
};

bool cmp(const job &a, const job &b)
{
  return a.finish < b.finish;
}
```

```
int check_Overloop(const job *j, const uint &n)
{
  for (int i = n - 2; i > -1; i--)
     if (j[n - 1].start >= j[i].finish)
        return i;
  return -1;
}

uint wi_sch(const job *j, const uint &n)
{
  if (n == 0)
     return j[0].profit;

  else
  {
     int i = check_Overloop(j, n);
     int incl = j[n - 1].profit;

     if (i != -1)
        incl += wi_sch(j, i + 1);

     int excl = wi_sch(j, n - 1);
     return (incl > excl ? incl : excl);
  }
}

uint max_profit(job *j, const uint &n)
{
  sort(j, j + n, cmp);
```

```
  cout << "\n\nSorted jobs according to respective finish time are :\n\n"
       << "S.N.\t   Start-time\t   Finish-time\t\tProfit\n";
  for (int i = 0; i < n; i++)
    cout << "\n " << i + 1 << "\t\t" << j[i].start << "\t\t"
         << j[i].finish << "\t\t" << j[i].profit;
  return wi_sch(j, n);
}

int main()
{
  job j[] = {{3, 10, 20}, {1, 24, 50}, {6, 19, 100}, {2, 100, 20}};
  int n = sizeof(j) / sizeof(j[0]);

  cout << "\n\t\t Practical 10 \n\t Weighted Interval Scheduling \n";

  cout << "\nGiven jobs are :\n\nS.N.\t   Start-time\t
          Finish-time\t\tProfit\n";

  for (int i = 0; i < n; i++)
    cout << "\n " << i + 1 << "\t\t" << j[i].start << "\t\t"
         << j[i].finish << "\t\t" << j[i].profit;

  int mx = max_profit(j, n);
  cout << "\n\nThe Maximum Optimal Profit is : " << mx << endl;
}
```

Output

```
                Practical 10
          Weighted Interval Schedueling

Given jobs are :

S.N.        Start-time      Finish-time          Profit

 1              3               10               20
 2              1               24               50
 3              6               19               100
 4              2               100              20

Sorted jobs according to respective finish time are :

S.N.        Start-time      Finish-time          Profit

 1              3               10               20
 2              6               19               100
 3              1               24               50
 4              2               100              20

The Maximum Optimal Profit is : 100
```

Q.11. Write a program to solve the 0-1 knapsack problem.

Ans.

Code

```
#include <iostream>
#include <algorithm>

using namespace std;
```

```
#define max(a, b) a > b ? a : b;

struct Item
{
  int val, wt;

  // Constructor
  Item(int value, int weight)
  {
     this->val = value;
     this->wt = weight;
  }
};

int knapsack_dynamic(const uint &n, const uint &W, const Item arr[])
{
  int dp[W + 1] = {0};

  for (int i = 1; i <= n; i++)
     for (int w = W; w >= 0; w--)
        if (arr[i - 1].wt <= w)
        {
           dp[w] = max(dp[w],
                                dp[w - arr[i - 1].wt] + arr[i - 1].val);
        }
  return dp[W];
}
```

```
double knapsack_greedy(const uint &n, uint W, Item arr[])
{
  sort(arr, arr + n, [](const Item &a, const Item &b){
            return (a.val * 1.0 / a.wt) > (b.val * 1.0 / b.wt);
        });

  double finalVal = 0;

  for (int i = 0; i < n; i++)
  {
     if (arr[i].wt <= W)
     {
        W -= arr[i].wt;
        finalVal += arr[i].val;
     }
     else
     {
        finalVal += arr[i].val * W * 1.0 / arr[i].wt;
        break;
     }
  }
  return finalVal;
}

int main()
{
  int W = 50; //    Weight of knapsack
  Item arr[] = {{60, 10}, {100, 20}, {120, 30}};
  int n = sizeof(arr) / sizeof(arr[0]);
```

```
    cout << "\n\t\t   Practical 11 \n\t\tKnapSack Algorithm \n";

    cout << "\n Given Items are :\n\n S.N.\tValues\tWeight\n";
    for (int i = 0; i < n; i++)
        cout << "\n  " << i + 1 << "\t" << arr[i].val << '\t' << arr[i].wt;

    cout << "\n\n Dynamic Approach \n Obtained maximum value : "
         << knapsack_dynamic(n, W, arr) << endl;

    cout << "\n Greedy Approach \n Obtained maximum value : "
         << knapsack_greedy(n, W, arr) << endl;
}
```

Output

```
                    Practical 11
                KnapSack Algorithm

Given Items are :

S.N.    Values  Weight

 1      60      10
 2      100     20
 3      120     30

Dynamic Approach
Obtained maximum value : 220

Greedy Approach
Obtained maximum value : 240
```

For the algorithms at S.No 1 to 3 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.

Ans.

Code

```
#include "algo.hpp"
#include <iomanip>
#include "pbPlots.hpp"
#include "supportLib.hpp"
using namespace algo;

bool drawGraph(vector<double> &xs, vector<double> &ys,
                                        const string &file_name)
{
  bool success;
  StringReference *errorMessage = new StringReference();
  RGBABitmapImageReference *imageReference =
                                        CreateRGBABitmapImageReference()
                                        ;

  ScatterPlotSeries *series = GetDefaultScatterPlotSeriesSettings();
  series->lineThickness = 2.0;
  series->xs = &xs;
  series->ys = &ys;

  ScatterPlotSettings *settings = GetDefaultScatterPlotSettings();
```

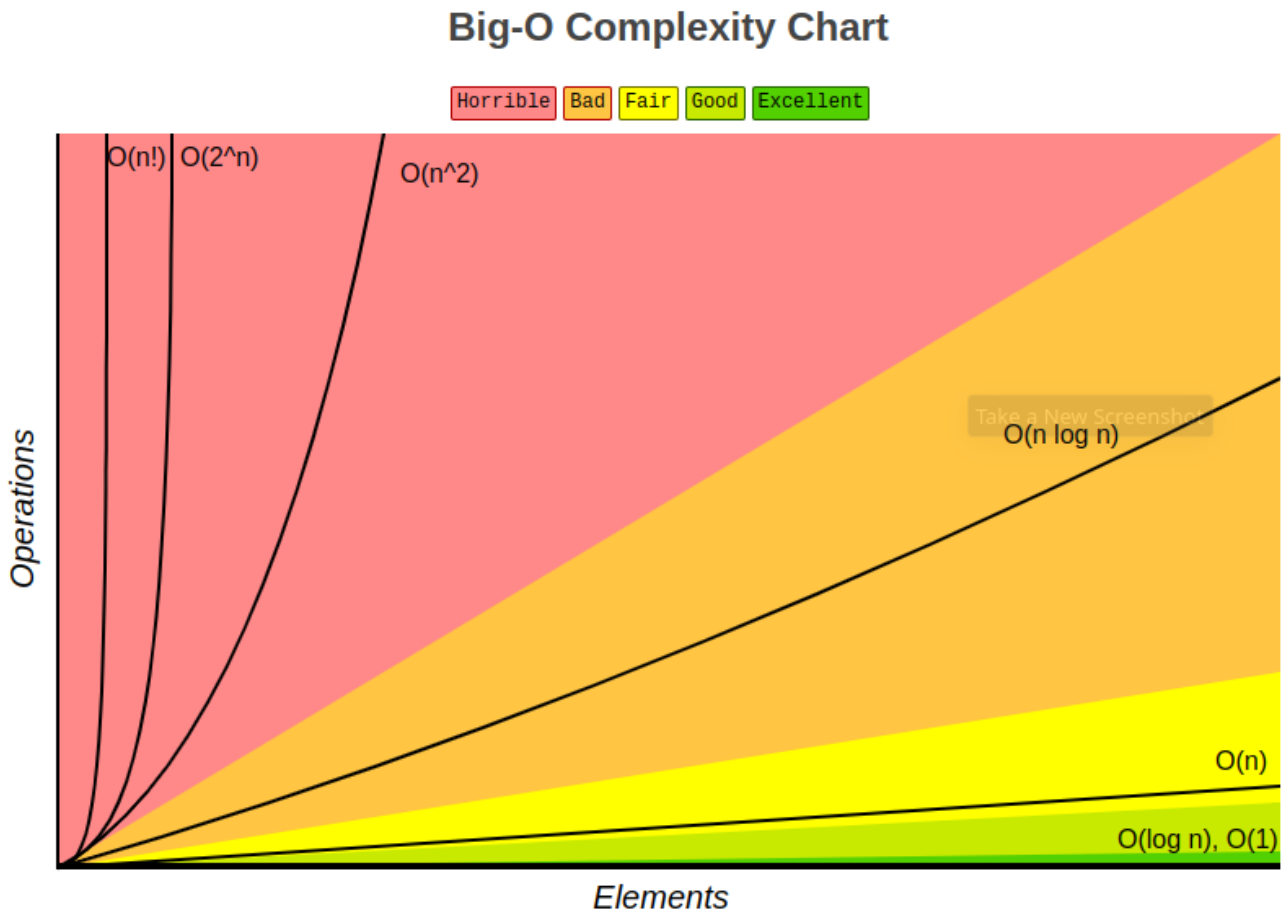
```
settings->width = 800;
settings->height = 600;
settings->autoBoundaries = false;
settings->xMax = GetMaximum(&xs);
settings->yMax = GetMaximum(&ys);
settings->autoPadding = true;
settings->title = toVector(L"Array Length (N) v/s Comparisons (C)");
settings->xLabel = toVector(L"N");
settings->yLabel = toVector(L"C");
settings->scatterPlotSeries->push_back(series);

success = DrawScatterPlotFromSettings(imageReference, settings,
                                                        errorMessage);
if (success)
{
   vector<double> *pngdata = ConvertToPNG(imageReference->image);
   WriteToFile(pngdata, file_name);
   DeleteImage(imageReference->image);
}
else
{
   cerr << "Error: ";
   for (wchar_t c : *errorMessage->string)
   {
      wcerr << c;
   }
   cerr << endl;
}
return success;
```

```
}

void random_array_sim(const uint &begin, const uint &stop,
                      const uint &n, vector<double> &length,
                      vector<double> *comparison)
{
  if (begin >= stop || n <= 0)
    return;

  uint len, arr[stop];

  for (uint i = 0; i < n; i++)
  {
    srand(stop + begin * i);
    len = begin + rand() % (stop - begin + 1);
    length.push_back(len);

    // Random Array Generation
    for (uint j = 0; j < len; j++)
      arr[j] = i + j + rand() % (len + n * j);

    comparison[0].push_back(insertionSort(arr, len));
    // For randomness
    for (uint j = 0; j < len; j += rand() % len / 8)
      swap(&arr[j], &arr[(i + j) % len]);

    comparison[1].push_back(mergeSort(arr, 0, len - 1));

    for (uint j = 0; j < len; j += rand() % len / 8)
```

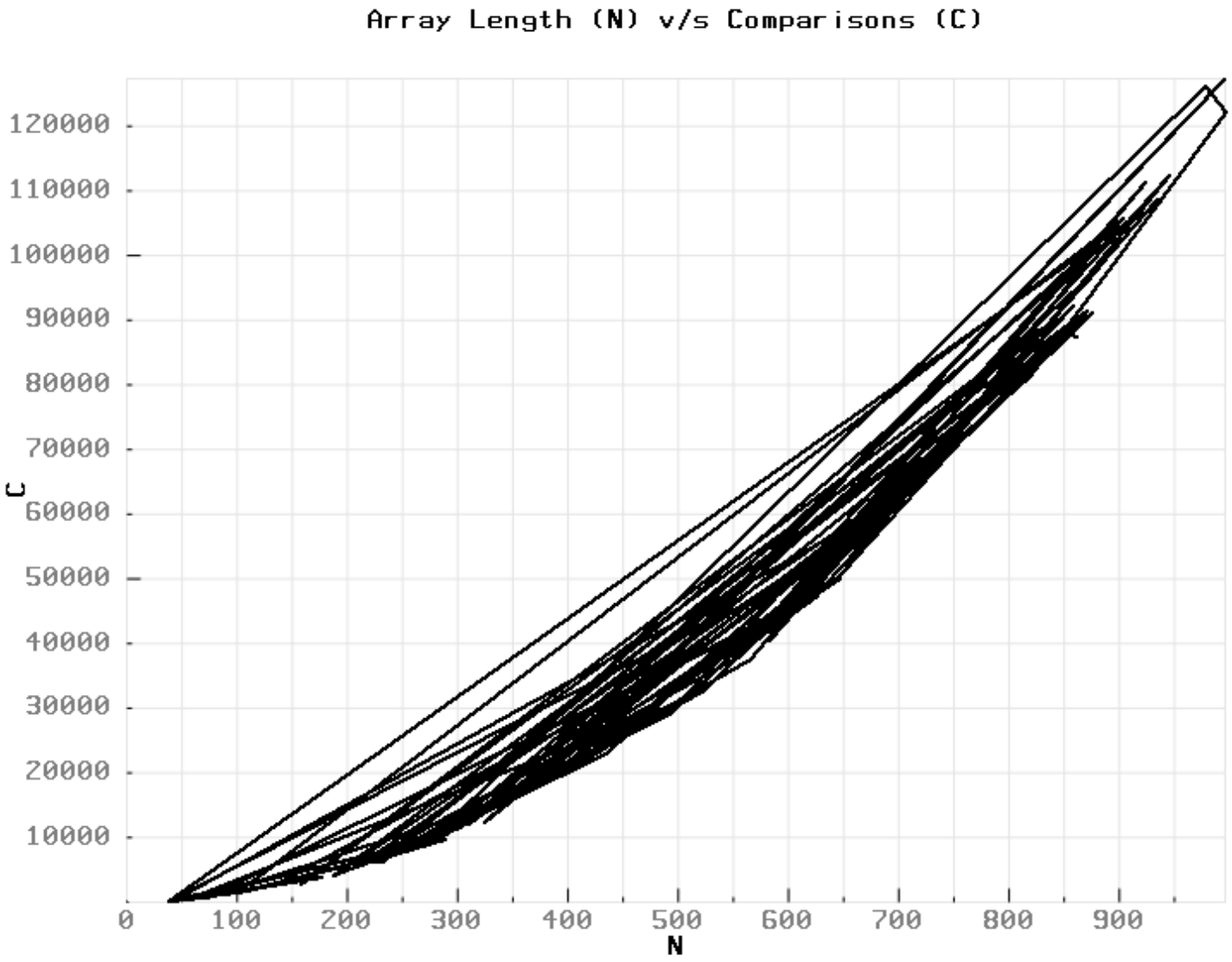
```
            swap(&arr[j], &arr[(i + j) % len]);

        comparison[2].push_back(heapSort(arr, len, greater<int>()));

        for (uint j = 0; j < len; j += rand() % len / 8)
            swap(&arr[j], &arr[(i + j) % len]);

        comparison[3].push_back(random_quickSort(arr, 0, len - 1));
    }
}

int main()
{
    vector<double> lens, cmps[4];

    string names[] = {
                        "InsertionSort.png",
                        "MergeSort.png",
                        "HeapSort.png",
                        "Random_QuickSort.png"
                      };

    random_array_sim(30, 1000, 100, lens, cmps);

    for (int i = 0; i < 4; i++)
        cout << boolalpha << "\n Creation of " << left << setw(25)
             << names[i] << " : " << drawGraph(lens, cmps[i], names[i]);
}
```

Output

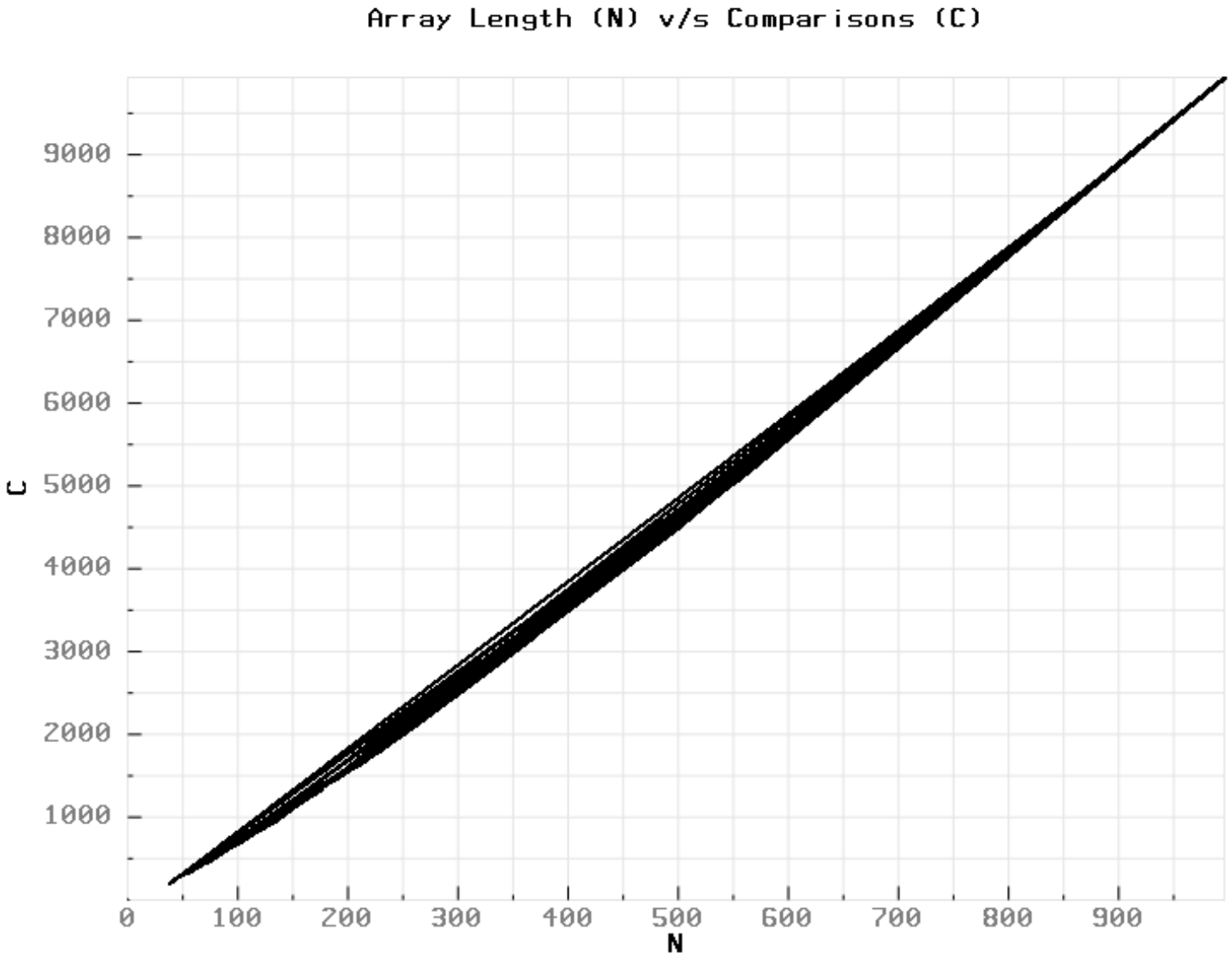
```
┌──(avi㉿kali)-[~/…/Design & Analysis _ Algorithm/Practical/Practicals/src]
└─$ g++ main.cpp pbPlots.cpp supportLib.cpp -o main

┌──(avi㉿kali)-[~/…/Design & Analysis _ Algorithm/Practical/Practicals/src]
└─$ ./main

Creation of InsertionSort.png          : true
Creation of MergeSort.png              : true
Creation of HeapSort.png               : true
Creation of Random_QuickSort.png       : true

┌──(avi㉿kali)-[~/…/Design & Analysis _ Algorithm/Practical/Practicals/src]
```



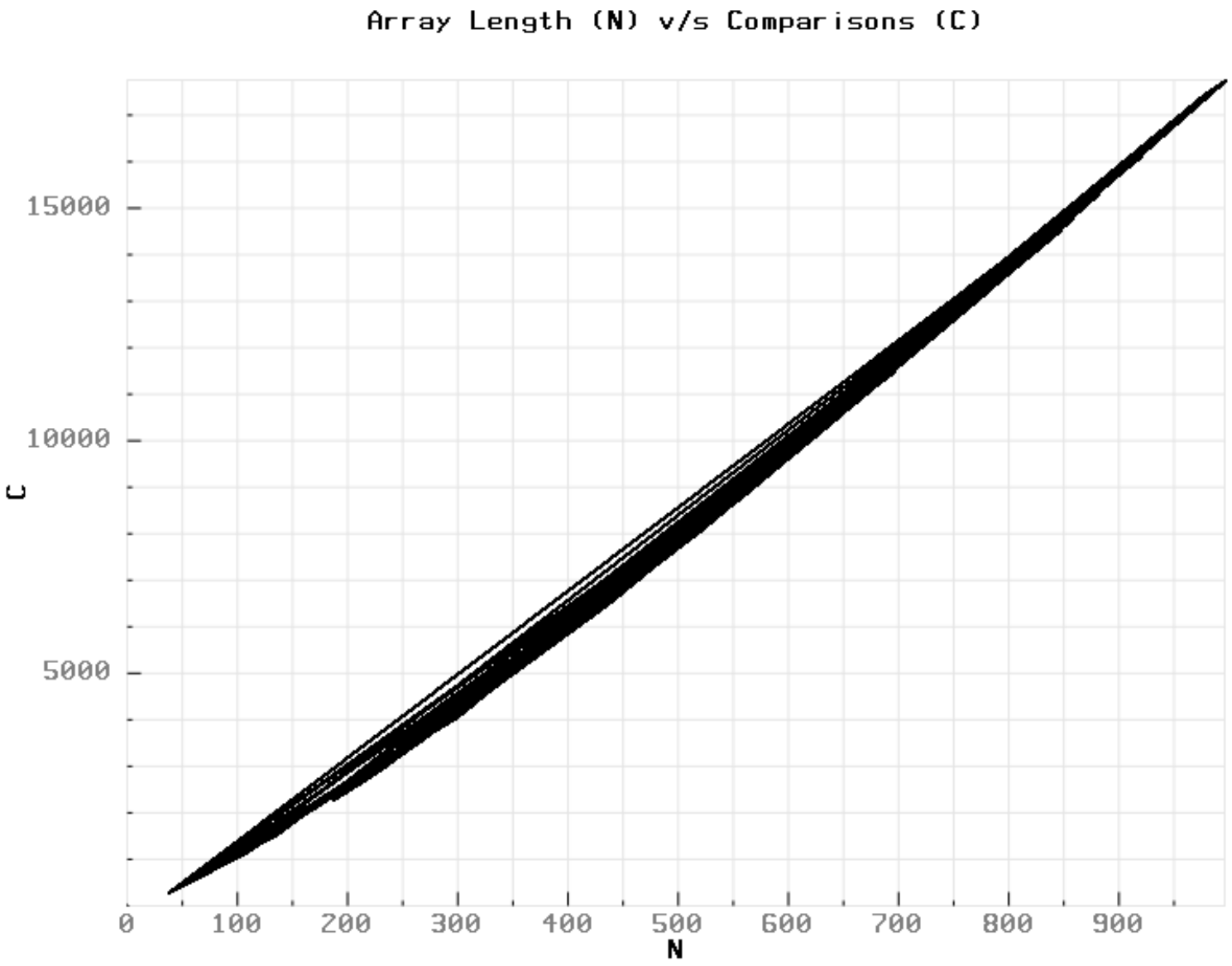
Insertion Sort



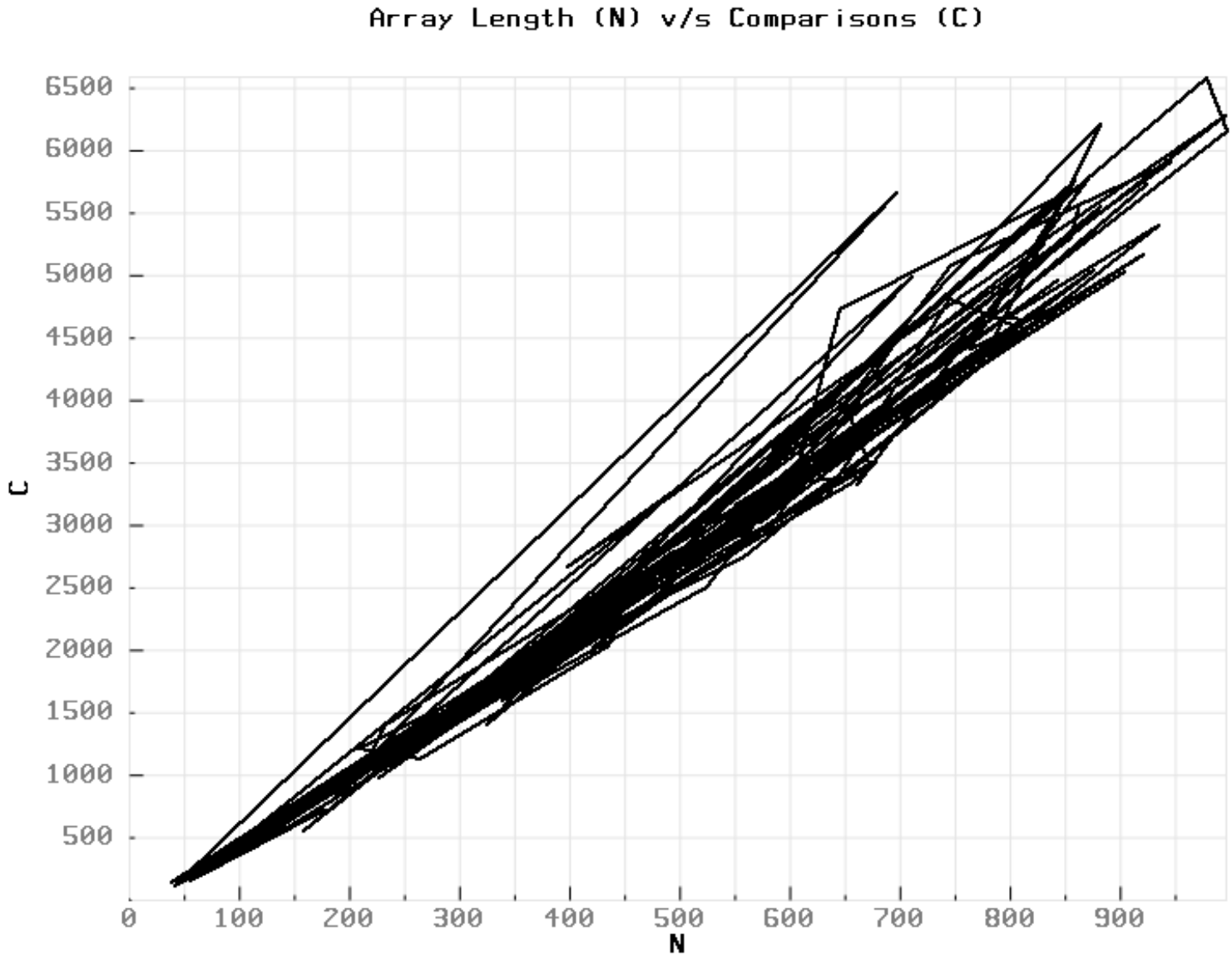
Merge Sort



Heap Sort



Randomized Quick Sort


Array Length (N) v/s Comparisons (C)
C
N
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
100
200
300
400
500
600
700
800
900