Experimental Verification and Analysis of Sorting Algorithms

Mostafa Gaafar, Jana Elmorsy, Kareem Elnaghy

Department of Computer Science and Engineering, The American University in Cairo

CSCE 1101
Dr. Howeida Ismail
May 13, 2022

Abstract

Computer science and artificial intelligence have been made to make people's lives easier and more efficient. However, computer science experts have raised multiple concerns regarding which algorithms, out of hundreds, should be used to make their application more successful and efficient. The main concerns are about sorting algorithms, which have been such a huge deal in the industry especially out of lately as it has been in need of development. Since databases in the modern world are limitless, there has become a larger demand for sorting algorithms that are more effective and efficient to help its users. Additionally, there has been a struggle by professionals on what sorting algorithms –out of the hundreds developed nowadays– is fit best for their applications. The choice of sorting algorithms depends on many aspects such as memory space needed, the size of the input data, the type of data, and many other factors. Therefore, it is important for a comparative study between all sorting algorithms. Because of this, our team has chosen to compare eleven sorting algorithms through experimentation to determine which is optimal for each application and to discover their advantages and disadvantages.

I. Introduction

Several sorting algorithms have been created over time to deal with various input data structures, sizes, and performance requirements, including time and space complexity. Each sorting algorithm has its own benefits and limitations from other sorting algorithms. Every sorting algorithm and its variations have different approaches to sorting data structure. For example, Counting and Radix sort, which are known to be greatly related, are known for how stable they are for large datasets since their time complexity is great because they are not a comparison-based algorithm. Another example is Insertion and Shell sort as they share a lot of similarities in their characteristics. While Shell sort uses a gap sequence that gets divided with each iteration, Insertion sort sorts the input data by comparing and then exchanging elements if needed. Both Tree and Heap sorting algorithms also share many similar characteristics in their approaches. They both rely on using tree structures to sort the input array. While Heap sort uses a binary heap data structure, Tree sort uses a binary search tree to obtain the sorted array from going through the tree. We chose these sorting algorithms to research on, along with others, as they share great similarities but with different approaches and that the sorting algorithms all vary in the characteristics that make them unique compared to others. We wanted to compare how the differences between sorting algorithms can impact the results and findings concluded.

II. Problem Definition

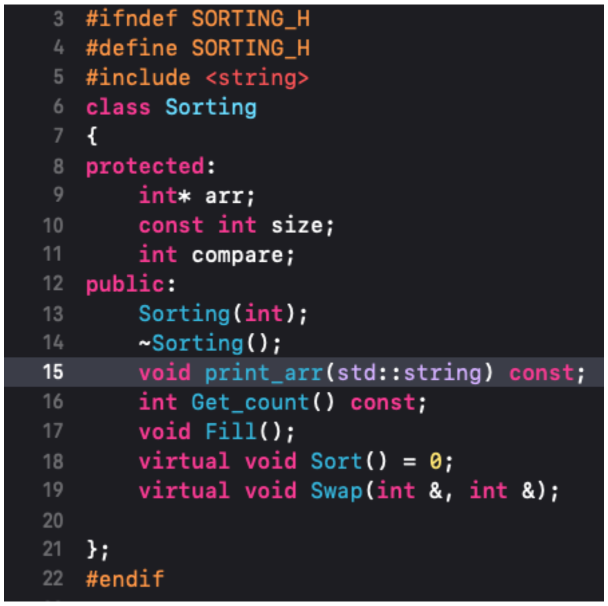
The need to develop sorting algorithms has grown in significance currently in the industry of computer science. Choosing the one algorithm that is the greatest fit for the application might be difficult for professionals. Therefore, our team took the decision to compare a total of eleven well-known sorting algorithms in order to address this problem. In order to maintain accurate findings from our results, we decided to implement our program that tests several algorithms. The number of comparisons, the number of swaps, and the time taken were the metrics we based our study on to evaluate each algorithm's performance. Based on these metrics, we concluded out findings of the algorithms' characteristics: stability, which shows how can each algorithm ensure all equal elements to be available in the output sorted array, adaptability, which means how can an algorithm adapt to different data structures efficiently, memory space, whether it is an in-place algorithm or if it requires additional memory, time complexity, which is the time taken for each algorithm to sort through the input data, and ease of implementation, whether it is easy to be implemented or taught. We used graphs in order to illustrate the results from testing each sorting algorithm to compare the model and average complexity. Different array sizes were used to show how efficient certain sorting algorithms are while working with large or small sets of data.

III. Methodology

The study aims to compare various sorting algorithms by generating a range of test cases varying in input sizes, N, with a randomized list of numbers. The input sizes used will be 1000, 2000, 3000, 5000, 7000, 10000. The main metric we are going to be focusing on measuring is the number of comparisons which will be done using a comparison counter. The number of comparisons is a metric that measures the number of times the algorithm compares two elements during the sorting process. This reflects the number of operations carried out during the process, and the more operations performed, the less efficient the algorithm is. We will first implement the sorting algorithms into a program using classes and relationships (See ***Appendix A***). Then after creating the structure, modifications will be made to each algorithm to track and store the required metrics during executions.

The program is designed in a way that the base class contains methods and attributes that are common to all sorting algorithms. The base class has three protected integer data members; a pointer to an array, constant size and a comparison counter. The public methods include a constructor that initializes the array and fills the array, a destructor to free up the memory used by the array, an operation to print the array, fill the array, swap two elements, retrieve the comparison count and a pure virtual operation called "Sort" that will be used in all derived classes to be the main sorting operation (See ***Figure 1***).

Figure 1 Code snippet of the base class for the program

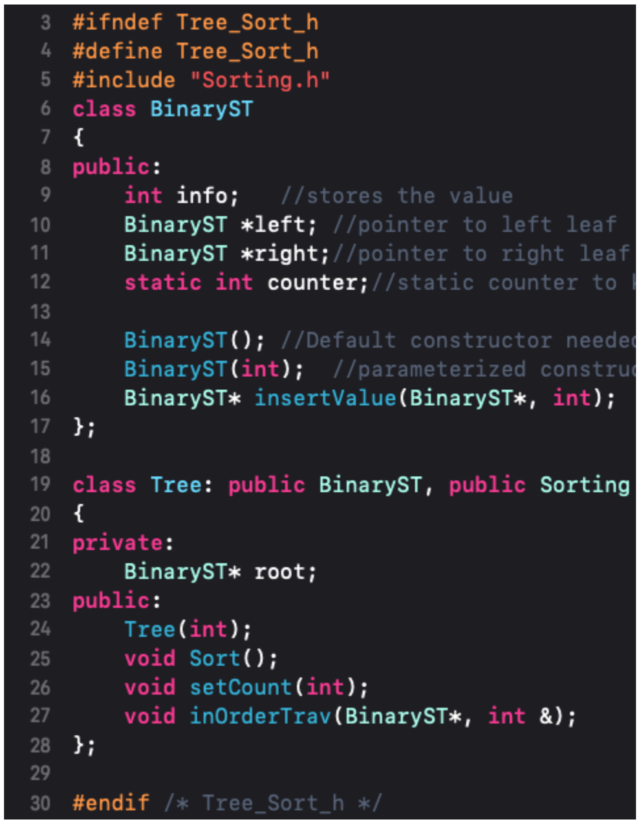


```
#ifndef SORTING_H
#define SORTING_H
#include <string>
class Sorting
{
protected:
    int* arr;
    const int size;
    int compare;
public:
    Sorting(int);
    ~Sorting();
    void print_arr(std::string) const;
    int Get_count() const;
    void Fill();
    virtual void Sort() = 0;
    virtual void Swap(int &, int &);

};
#endif
```

All the derived class will call upon this class's constructor and will override the pure virtual function with their unique sorting algorithms. The derived classes are all almost identical with them all including a constructor that calls on the base class constructor and then an overridden "Sort" function. Moreover, each class will have additional functions unique to the sorting algorithms such as the partition function for quick sort and the merge function for merge sort.

With tree sort, there is an additional class to form the binary search tree that will be used to sort the array (See ***Figure 2***). The program mainly consists of generalization relationships with an additional aggregation relationship with the Tree sort class.

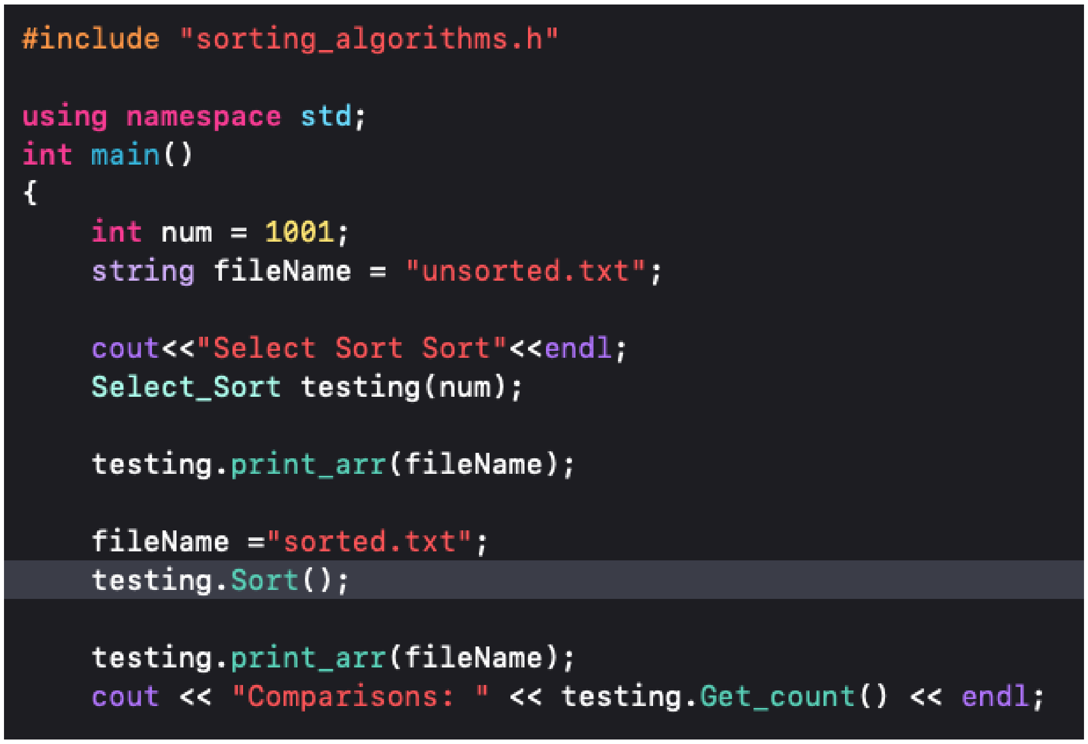
Figure 2 Code snippet of a sample derived class (Tree sort)



```
#ifndef Tree_Sort_h
#define Tree_Sort_h
#include "Sorting.h"
class BinaryST
{
public:
    int info;   //stores the value
    BinaryST *left; //pointer to left leaf
    BinaryST *right;//pointer to right leaf
    static int counter;//static counter to
    
    BinaryST(); //Default constructor neede
    BinaryST(int);  //parameterized constru
    BinaryST* insertValue(BinaryST*, int);
};

class Tree: public BinaryST, public Sorting
{
private:
    BinaryST* root;
public:
    Tree(int);
    void Sort();
    void setCount(int);
    void inOrderTrav(BinaryST*, int &);
};

#endif /* Tree_Sort_h */
```

The main function starts by declaring the size of the array to be passed into the parameterized constructor to create the dynamic array to be sorted. We added an fstream implementation to print the unsorted and sorted array into two files to avoid clutter in the compiler. So with this in mind, after declaring the size, a string variable is declared to store the file name to be passed into the print array function which creates a file and writes the elements of the array into the file. The "Sort" function is then called to sort the array using the method based on the object created. The print function is then called again with an updated file name to print the sorted array into a new file. The comparisons are then outputed using the counter getter function. ***Figure 3*** is a part of the main function which uses selection sort to sort the array. The rest of the header file code screenshots cant be seen in ***Appendix B***.

Figure 3 Code snippet of a sample of the main function (Selection Sort)



```
#include "sorting_algorithms.h"

using namespace std;
int main()
{
    int num = 1001;
    string fileName = "unsorted.txt";

    cout<<"Select Sort Sort"<<endl;
    Select_Sort testing(num);

    testing.print_arr(fileName);

    fileName ="sorted.txt";
    testing.Sort();

    testing.print_arr(fileName);
    cout << "Comparisons: " << testing.Get_count() << endl;
```

IV. Specification of Algorithms Used:

Sorting is one of the most essential processes in data organization and processing, especially when it is too much for a human to handle. Due to how urgently the world's millions of datasets need to be sorted, sorting algorithms must be developed regularly in the industry. In our project, we will discuss 11 sorting algorithms, along with their efficiency and limitations. Additionally, there will be comparisons made, both theoretical and practical, between them in order to determine which algorithm is preferable for which industry and aspect.

1. Bubble Sort:

Bubble sort is known to be the simplest sorting algorithm –this will be a factor to as why it is the slowest and inefficient one– that works by comparing each two adjacent elements and swapping them if needed until the list is sorted.

Time complexity:

# it solely depends on the number of digits (d) in the values.

- **Worst case:** O(n^2)
- **Average case:** O(n^2)
- **Best case:** O(n) –> this happens when the given array is sorted.

Advantages:

- A simple algorithm that can be easily implemented and learned.
- Stable sorting algorithm as it maintains keeping the equal values in the output array.
- It is an in-place algorithm that means it doesn’t require any additional memory space.

Disadvantages:

- Poor time complexity.

- Not suitable for larger input values.

Bubble Sort Implementation:

```
bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n - 1; i++)
      for (j = 0; j < n - i - 1; j++)
         if (arr[j] > arr[j + 1])
            swap(arr[j], arr[j + 1]);
}
```

2. **Comb Sort:**

Comb Sort is a non-comparison algorithm, which is the advanced version of Bubble Sort. The process starts by comparing the elements separated at this distance until the list eventually becomes sorted by starting with a wider gap and gradually narrowing it. The gap is divided by the shrink factor of 1.3.

Time complexity:

# it solely depends on the number of digits (d) in the values.

- **Worst case:** $O(n^2)$
- **Average case:** $O(n^2/2^{\mathbf{I}})$ → "I" stands for the increments.
- **Best case:** $O(n \log (n))$

Advantages:

- A simple algorithm that can be easily implemented and learned.
- More efficient than Bubble Sort.
- It is an in-place algorithm that means it doesn't require any additional memory space.

Disadvantages:

- Solely depends on the specified shrink factor

- Poor time complexity just as Bubble Sort
- Not suitable for larger input values.

Comb Sort Implementation:

```
int nextGap(int x)
{
   int gap = x/1.3;
   if(gap < 1)
      return 1;
      return gap;
}
void CombSort(int arr[], int size)
{
   int gap = size;
   bool swapped = true;
   while(gap != 1 || swapped == true)
   {
      gap = nextGap(gap);
      swapped = false;
      for(int i = 0; i<size - gap; i++)
      {
         if(*(arr+i) > *(arr+i+gap))
      {
            swap(*(arr+i), *(arr+i+gap));
            swapped = true;
         }
      }
   }
}
```

3. Selection Sort

Selection sort is a comparison-based algorithm that initially works by dividing the list into two arrays: the unsorted part, which contains all of the elements, and the sorted section, which initially starts with no elements. Depending on whether the array needs to be sorted in ascending or descending, it chooses the element with the greatest position in the list, which may be the smallest or biggest element. It then compares it with the element in the first index and swaps it. The process keeps on going until the arrays are sorted.

Time complexity:

- Worst, average and best time complexities all have the same, which is O(n^2)

Advantages:

- A simple algorithm to be implemented and taught
- A stable algorithm
- In-place algorithm as it doesn't need any more additional memory space.
- Efficient for small data input.

Disadvantages:

- Inefficient for large data input.
- Even if the list is partially or completely sorted, it will still go through the same process.

Selection Sort Implementation:

```
selectionSort (int arr[], int n)
{
   for (int i = 0; i < n-1; i++)
   {
      int min = i;
      for (int j = i+1; j < n; j++)
      {
        if (arr[j] < arr[min])
            min = j;
      }
      if (min!=i)
         swap(&arr[min], &arr[i]);
   }
}
```

4. **Insertion code:**

Insertion sort is a simple sorting algorithm that has evolved throughout time with research from multiple computer developers.The sorting starts by dividing the given array into two subarrays: one that is sorted (the first element) and another unsorted array. Furthermore, the sorted subarray is repeatedly filled with each element from an unsorted subarray. It compares each element one by one, starting with the second element on the list and placing it in the right position in the sorted list. This process is repeated for each consecutive value in the list until all values are sorted.

Time complexity:

- **Worst case:** O(N^2)
- **Average case:** O(N^2)
- **Best case:** O(N)

Advantages:

- Efficient for small data values.
- Efficient when the input array is almost sorted.
- A stable sorting algorithm.

Disadvantages:

- Its efficiency degrades at a larger pace as the size of input data is increased, as compared to other algorithms.
- If the array is sorted in reverse, the worst case scenario occurs.

Insertion Sort Implementation:

```
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;
  }
}
```

5. **Shell Sort:**

Donald Shell founded Shell Sort in 1959 after observing Insertion Sort and its benefits and limitations(). He observed that insertion sorting does not work on larger inputs and introduced shell sort. The process allows for comparisons on a wide range of input sizes. He introduced the notion of comparing elements with a gap (distance) between them. The algorithm starts by comparing items spaced with the specified gap size and swaps them if required. The gap then gradually narrows until all arrays are sorted, ending with the gap sequence being one.

Time complexity:

# The complexity depends on the gap sequence specified.

The following time complexity is for the <u>Shell Sequence:</u>

- **Worst case:** O(n^2)
- **Average case:** O(n log n)
- **Best case:** O(n log n)

Advantages:

- More efficient than Insertion Sort.
- Works great on larger input sizes.
- in-place sorting algorithm which means it does not need any extra memory space.

Disadvantages:

- The specified gap sequence used determines the algorithm's time complexity, which is difficult to calculate for any given array.

Shell Sort Implementation:

```
void shellSort(int arr[], int size) {
int i, j;
  for (int gap = size / 2; gap > 0; gap /= 2)
  { for (i = gap; i < size; i++)
    {
      int temp = *(arr+i);
      for (j = i; j >= gap && *(arr+(j - gap)) > temp; j -= gap)
      {
          *(arr+j) = *(arr+(j - gap));
      }
        *(arr+j) = temp;
    }
  }
}
```

6. Counting Sort

Harold Seward founded the developed version of Counting Sort in 1954. It is known to be a non-comparison algorithm as it relies on the array elements provided to generate an array with the same range (max element-min element). Its idea is to arrange array elements according to the number of occurrences of each element. It includes three arrays; the given array, the one that will be used to locate the occurrences of each element in its index number and an output sorted array. The process starts with determining the number of occurrences of each element in the array, which is then put in a temporary second array where it stores the number of occurrences of each element at its corresponding index. Then, it uses the number of occurrences for each element to determine the element's location in the output sorted array.

Time complexity:

- **Worst case:** O(n+k)
- **Average case:** O(n+k)
- **Best case:** O(n+k)

Advantages:

- It is a non-comparison algorithm; therefore, it will take less time as it wont need to compare each element to another in the array.
- It is known to be stable, as it makes sure all the equal array elements are still available in the output sorted array

Disadvantages:

- It is an in-place algorithm, which means it will need additional memory space as it includes three arrays.
- Not efficient if the range of the given array elements is large as the size of the counting array depends on it, which can be relatively large and need more memory space than needed.

Counting Sort Implementation:

```
int getMax(int arr[], int size)
  {
    int max = *(arr);
    for (int i = 1; i < size; i++)
    {
      if (*(arr + i) > max)
```

```
            max = *(arr + i);
        }

        return max;
    }
  int getMin(int arr[], int size)
    {
        int min = *(arr);
        for (int i = 1; i < size; i++)
        {
            if (*(arr + i) < min)
                min = *(arr + i);
        }
        return min;
    }
    void CountSort(int arr[], int size)
    {
        int* output = new int[size]();
        int range = getMax() - getMin();
        int* count = new int[range + 1]();
        int i;
        for (i = 0; i <size; i++)
            ++(count+(arr+i));
        for (i = 1; i <= (range); ++i)
            *(count+i) += *(count+(i - 1));

        for (i = 0; i < size; i++)
        {
            int j = (count+(arr+i));
            *(output + (--j)) = *(arr+i);
        }

        for (i = 0; i < size; i++)
            *(arr+i) = *(output+i);

        delete[]output;
        delete[]count;
    }
```

7. Radix Sort:

Radix Sort is a non-comparative sorting algorithm that acts by sorting the elements starting with the least significant digit (LSD) and continuing to the most significant digit (MSD). We start comparing each digit of an element to its corresponding digit in the other element. This process keeps on going until it has checked every digit and the sorting has ended. Since Radix Sort uses Counting Sort, it includes two additional arrays other than the unsorted given array. Radix sort can be used on many data structures other than numbers.

Time complexity:

# it solely depends on the number of digits (d) in the values.

- **Worst case:** O(n^2)
- **Average case:**O(d * n)
- **Best case:** O(n)

Advantages:

- Efficient for sorting large numbers.
- A linear time complexity, which makes it a faster sorting algorithm.
- Stable sorting algorithm as it keeps all the equal values of the array in the output array.

Disadvantages:

- It requires additional memory since it is not an in-place algorithm.

Radix Sort Implementation:

```
void RadixSort(int arr[], int size)
{   int max = getMax();
   for (int i = 1; (max / i) > 0; i *= 10)
      countSort(i);
}
void countSort(int arr[], int size, int x)
{
   int output[size];
   int i;
   int count[10] = {0};
   for (i = 0; i < size; i++)
      ((count + ((arr + i) / x) % 10))++;
   for (i = 1; i < 10; i++)
      *(count + i) += *(count + (i - 1));
      for (i = size - 1; i >= 0; i--) {
      (output + *(count + ((arr + i) / x) % 10) - 1) = *(arr + i);
      ((count + ((arr + i) / x) % 10))--;
   }   for (i = 0; i < size; i++)
 *(arr+i) = *(output+i);
}
```

8. **Merge Sort:**

Merge sort was founded by John von Neumann in 1945 by developing it into a divide-and-conquer algorithm. It is a reliable and efficient sorting algorithm that is well-known in the computer science industry as it is known to recursively sort by dividing large arrays into smaller subarrays, sorts each subarray individually, and then merges the sorted halves back together.

Time complexity:

# Since the array is divided by half, all the cases have the same time complexity

- **Worst case, Average case, Best case:** O(n * log n)

Advantages:

- A stable sorting algorithm, which means the order of equal elements will be kept in the output sorted array.
- Works greatly for large data input.
- Adaptable to any data input type.

Disadvantages:

- It requires additional memory space due to the divided sub-lists.

- Not efficient on smaller data input.
- It can handle any type of unsorted input, whether completely or partially sorted.

Merge Sort Implementation:

```
void shellSort(int arr[], int size) {
int i, j;
  for (int gap = size / 2; gap > 0; gap /= 2)
  { for (i = gap; i < size; i++)
    {
      int temp = *(arr+i);
      for (j = i; j >= gap && *(arr+(j - gap)) > temp; j -= gap)
      {
         *(arr+j) = *(arr+(j - gap));
      }
        *(arr+j) = temp;
    }
  }
}
```

9. **Tree Sort**

Tree sort is a sorting algorithm that is based on Binary Search Tree data structure The process starts on with creating a binary search tree from the input data type given. In our paper, it will be from an array data structure. A binary search tree is when every node has two children nodes: one on the left is smaller than the parent node, whereas the one on the right is larger. Furthermore, the tree sort visits the nodes in ascending order of their values. Therefore, we get the sorted output array.

Time complexity:

- **Worst case:** O(n^2)
- **Average case:** O(n log n)
- **Best case:** O(n log n)

Advantages:

- A stable algorithm as it keeps the order of the equal elements in the data output array.

Disadvantages:

- The performance of the sorting algorithms solely depends on the distribution of the input elements which might affect the balancing of the tree
- Requires additional memory space, as it creates a binary tree structure in order for it to get sorted.

Tree Sort Implementation:

```
BinaryST* BinaryST::insertValue(BinaryST* node, int x)
{
   if(!node)
   {
      return new BinaryST(x);
   }

   counter++;
   if(x > node->info)
   {
      node->right = insertValue(node->right, x);
   }

  else if(x <= node->info)
   {
      node->left = insertValue(node ->left, x);
   }

      return node;
}

Tree::Tree(int y): Sorting(y)
{
   root = nullptr;
}
void Tree::Sort()
{
   int i =0;
   root = insertValue(root, *arr);
```

```
     for(int i = 1; i<size; i++)
   {
      root = insertValue(root, *(arr+i));
   }
   compare = root->counter;
     inOrderTrav(root, i);
}
void inOrderTrav(BinaryST* node , int &i)
{
   if(node != nullptr)
   {
      inOrderTrav(node->left, i);
      *(arr +(i++)) = node->info;
      inOrderTrav(node->right, i);
   }
}
```

10. Heap Sort

Heapsort was invented by J. W. J. Williams in 1964. It is the comparison-based algorithm that uses Binary Heap data structure. It starts to convert the binary tree, which is represented as an array, into max heap by a function called 'Heapify. Max heap is a binary tree where each node's value is greater than or equal to its children's values. The process begins with swapping the maximum element with another on the top of the array. The algorithm then swaps the maximum element with the last element in the heap and then removes the maximum element from the heap, which is then put in a sorted array. The same process happens all over again to each of the remaining elements until the array is sorted.

Time complexity:

# Worst, average, and best time complexities all have the same time taken.

- O (n * log n)

Advantages:

- Consistent as it has the same time complexity for all scenarios, which can be efficient

Disadvantages:

- Unstable algorithm, as it might not keep the order of equal elements in the sorted output array.
- Requires additional memory space, as it creates a binary tree structure in order for it to get sorted.

Heap Sort Implementation:

```
heapify(int *arr,int i,int s)
{
int max = i;
int l = (2 * i) + 1;
int r = (2 * i) + 2;
if ((s>l)&&(*(arr+l) > *(arr+max)))
{
compare++;
max = l;
}
   if ((s > r) && (*(arr+r) > *(arr+max)))
   {
       compare++;
       max = r;
   }
   if (i!=max)
   {
       Swap(*(arr+i), *(arr+max));
       heapify(arr, max, s);
    }
}

heapSort(int arr, int size)
{
     for (int i = size - 1; i >= 0; i--)
     {
       Swap(*arr, *(arr+i));
       heapify(arr, 0, i);
}
}
```

11. Quick Sort

Quick sort is a divide and conquer algorithm that was founded by Tony Hoare in 1959 in hopes to improve the efficiency of sorting algorithms. The algorithm starts with assigning an element from the input array with pivot. Furthermore, the other elements are divided into two sublists according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. Tony introduced the notion of partition, in which its main aim is to put the pivot in its correct position. Moreover, in our paper we chose to use two quick sort algorithms that are almost identical but slightly differ in one aspect. Quick sort 1 uses the pivot as the **first** element in the array while Quicksort 2 uses the pivot as a **random** element.

Time complexity:

***It mainly depends on the choice of pivot.***

- Worst case: $O(n^2)$ → this means the pivot was chosen very poorly
- Average case: O(n * logn)
- Best case: O(n * logn)

Advantages:

- Efficient on larger data input.
- In-place algorithm, which means it does not require any additional memory space.
- Easy to implement and learned

Disadvantages:

- Inefficient in smaller data input.
- Unstable, as it might lose some of the equal elements when the array is sorted.

Quick Sort 1 Implementation:

```
void QuickSort(int arr[], int low, int high)
{
   if (low < high)
   {
      int pivotIndex = partition(array, low, high);
      QuickSort(array, low, pivotIndex - 1);
      QuickSort(array, pivotIndex + 1, high);
   }
}
int partition(int arr[], int low, int high) {

     int pivot = *(arr+low);
     int temp = high;
     for (int i = high; i > low; i--) {
        if (*(arr+i) > pivot)
        {
           Swap(*(arr+i), *(arr+(temp--)));
        }
           compare++;
        }
     Swap(*(arr+low), *(arr+temp));
     return temp;
}
```

Quick Sort 2 Implementation:

```
void QuickSort(int arr[], int low, int high) {
   if (low < high)
   {
      int pivotIndex = randomPartition(array, low, high);
      QuickSort(array, low, pivotIndex-1);
      QuickSort(array, pivotIndex + 1, high);
   }
}
int randomPartition(int *q, int low, int high)
{
   int randomIndex = (rand() % (high - low)) + low;
   Swap((q+(randomIndex)),(q+(low)));
   return partition(q, low, high); }
```

V. Data Specifications:

In this study, we aim to experimentally verify and analyze the performance of 11 different sorting algorithms, including Selection, Insertion, Quick, Merge, Tree, Heap, Radix, Counting, Bubble, Comb, and Shell Sort. Our objective is to determine which algorithm performs best under various input sizes and to analyze the number of comparisons. The input data will consist of randomly generated integer arrays with input sizes varying from 1000 to 10,000. Furthermore, we implemented a function that generates a Random Permutation Array which will be used across all tests with the various sorting algorithms. This ensures that each element is unique and occurs only once in the entire array, avoiding duplicates. The Random Permutation Array works by filling the array with numbers from 1 to N leaving the first element unused (0) so that we can verify that the sorting algorithm is correct. After filling the area with the values, using the “cstdlib” and "ctime” libraries, a random seed generator is created to randomly shuffle the elements in the array to ensure that the array is randomized for each test run. The program is designed to ensure that the sorting algorithms can be tested independently and accurately utilizing classes. The experiment is conducted on the same computer system to ensure reliable and consistent results. The results will be reported in a tabular and graphical format displaying the experimental and model results. The graphical representation of the results helped in visualizing the comparative analysis of the algorithms.

VI. Experimental Results:

1) Selection Sort

a) Tabular Results

Table 1 Selection Sort Algorithm

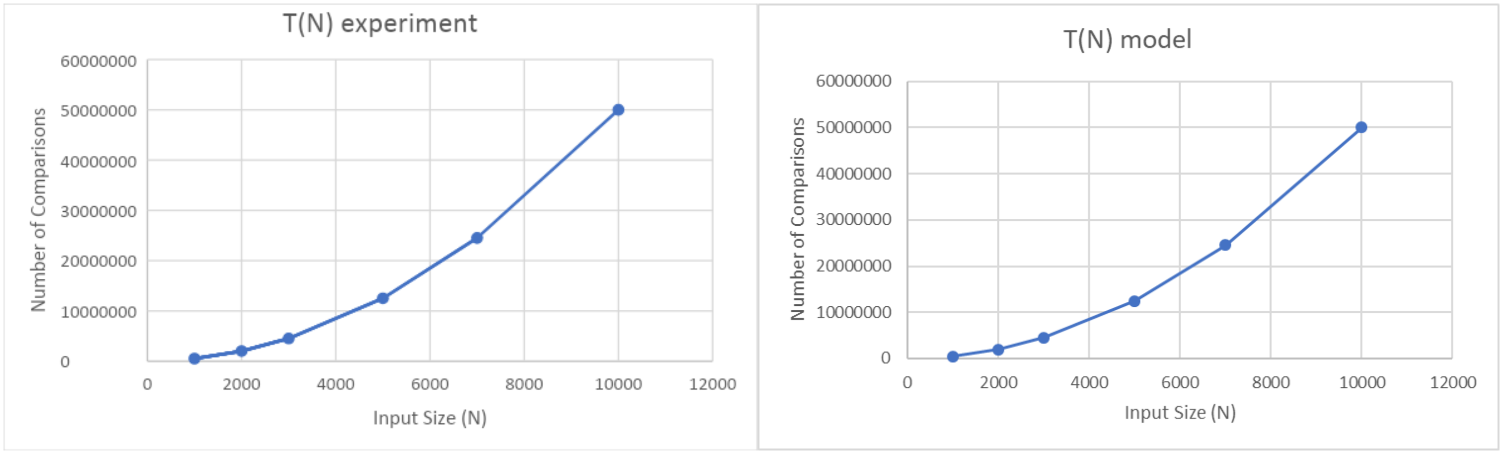
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	499500	1999000	4498500	12497500	24496500	49995000
T(N) Model	500500	2001000	4501500	12502500	24503500	50005000

b) Model Equation

$$T_{model}(N) = N(N-1)/2$$

c) Graphical Representation

Figure 1 T(N) model and experiment graphs for Selection sort



2) Insertion Sort

a) Tabular Results

Table 2 Insertion Sort Algorithm

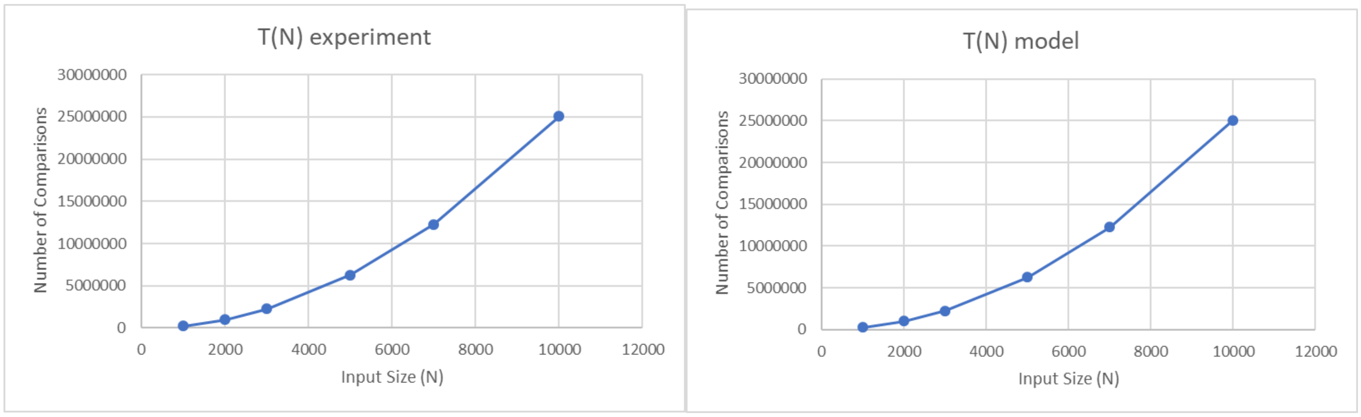
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	249373	1006305	2268399	6285015	12253786	25057274
T(N) Model	256242	1008651	2261814	6270401	12282003	25055060

b) Model Equation

$$T_{model}(N) = 4586.397 + 0.7775294*N + 0.2503769*N^2$$

c) Graphical Representation

Figure 2 T(N) model and experiment graphs for Insertion sort



3) Merge Sort

a) Tabular Results

Table 3 Merge Sort Algorithm

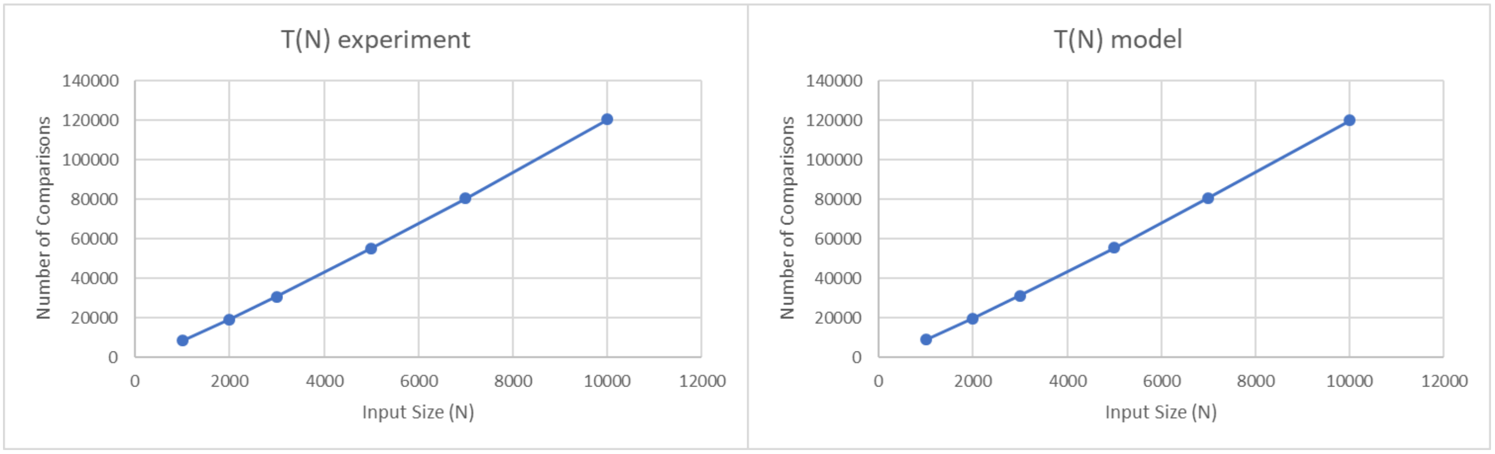
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	8678	19457	30841	55252	80659	120443
T(N) Model	9012	19822	31313	55510	80779	120042

b) Model Equation

$$T_{model}(N) = 0.9034148*N*log_2(N)$$

c) Graphical Representation

Figure 3 T(N) model and experiment graphs for Merge sort



4) Quick Sort (First Element Pivot)

a) Tabular Results

Table 4 Quick Sort Algorithm with first element pivot

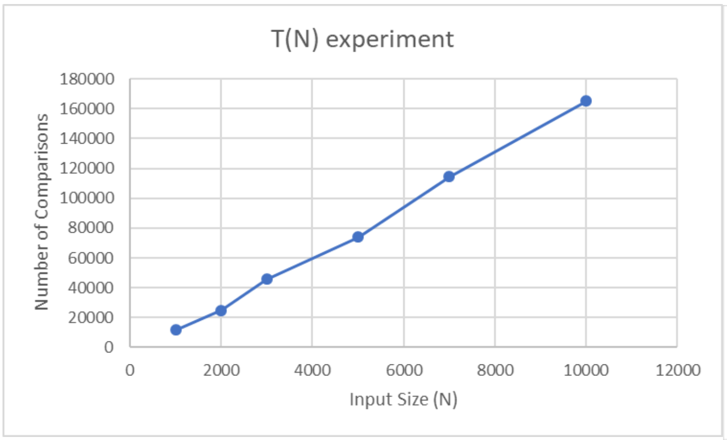
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	11524	24900	45680	73974	114533	165231
T(N) Model	12463	27413	43305	76769	111715	166014

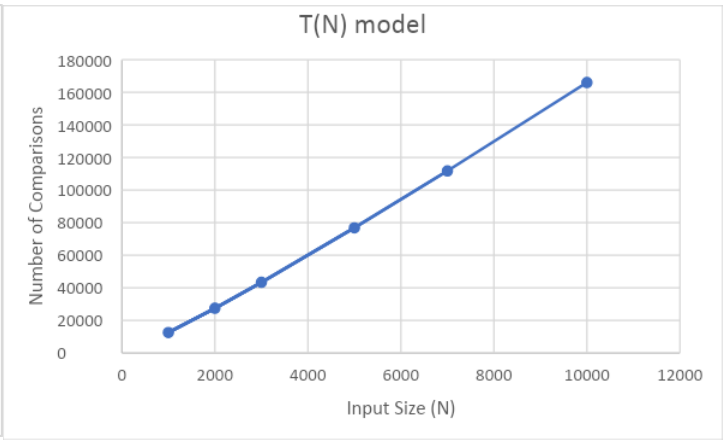
b) Model Equation

$$T_{model}(N) = 1.249246*N*log_2(N)$$

c) Graphical Representation

Figure 4 T(N) model and experiment graphs for Quick sort (first element pivot)





5) Quick Sort (Random Element Pivot)

a) Tabular Results

Table 5 Quick Sort Algorithm with Random Element Pivot

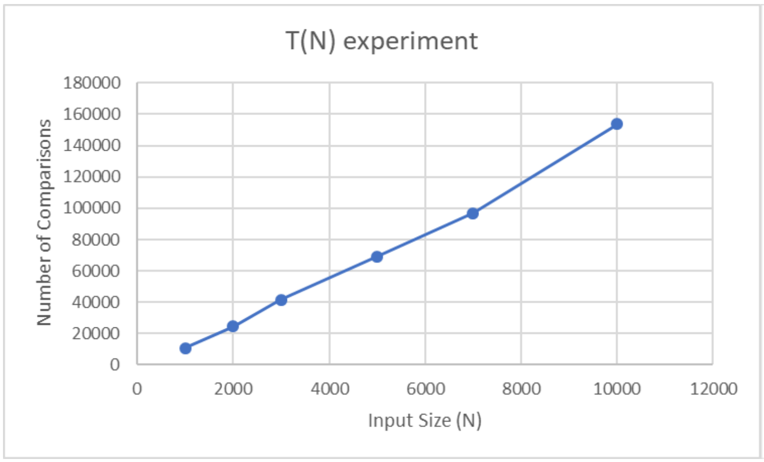
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	10822	24507	41438	69189	96730	153794
T(N) Model	11323	24904	39341	69742	101490	150820

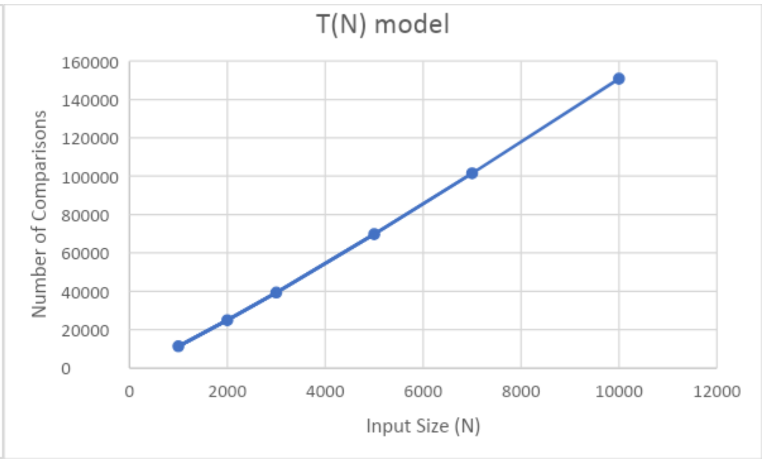
b) Model Equation

$$T_{model}(N) = 1.134908*N*log_2(N)$$

c) Graphical Representation

Figure 5 T(N) model and experiment graphs for Quick sort (Random element pivot)





6) Counting and Radix Sort

a) Tabular Results

Table 6 Counting Sort and Radix Sort Algorithm

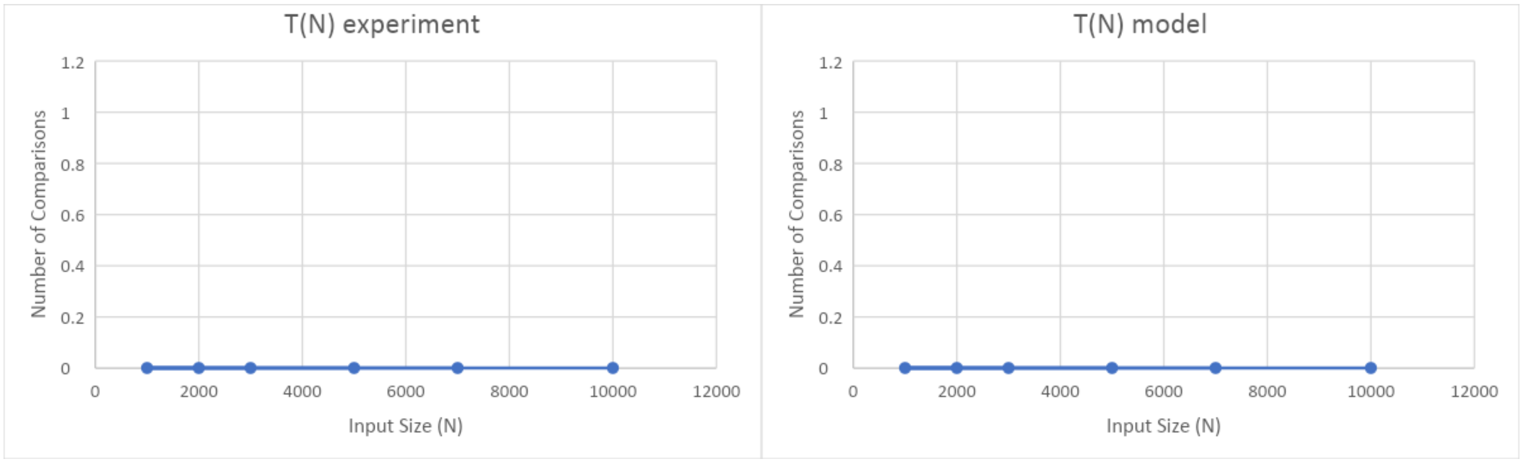
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	0	0	0	0	0	0
T(N) Model	0	0	0	0	0	0

b) Model Equation

$$T_{model}(N) = N/A$$

c) Graphical Representation

Figure 6 T(N) model and experiment graphs for Counting and Radix sort



7) Heap Sort

a) Tabular Results

Table 7 Heap Sort Algorithm

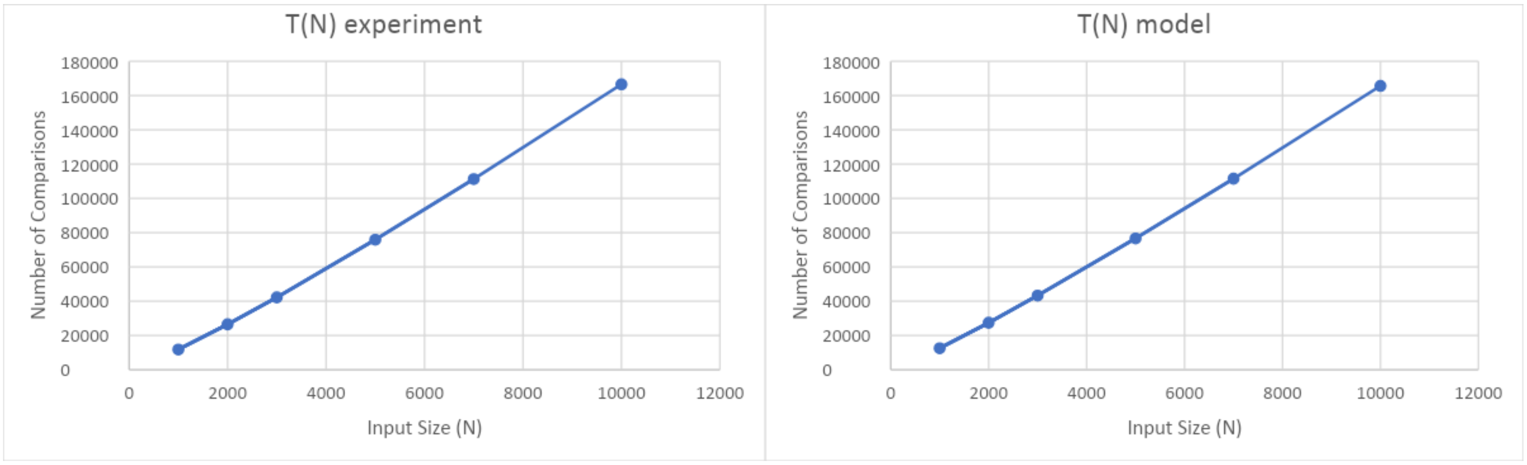
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	11723	26428	42175	75841	111231	166564
T(N) Model	12435	27350	43206	76593	111459	165634

b) Model Equation

$$T_{model}(N) = 1.246383*N*log_2(N)$$

c) Graphical Representation

Figure 7 T(N) model and experiment graphs for Heap sort



8) Shell Sort

a) Tabular Results

Table 8 Shell Sort Algorithm

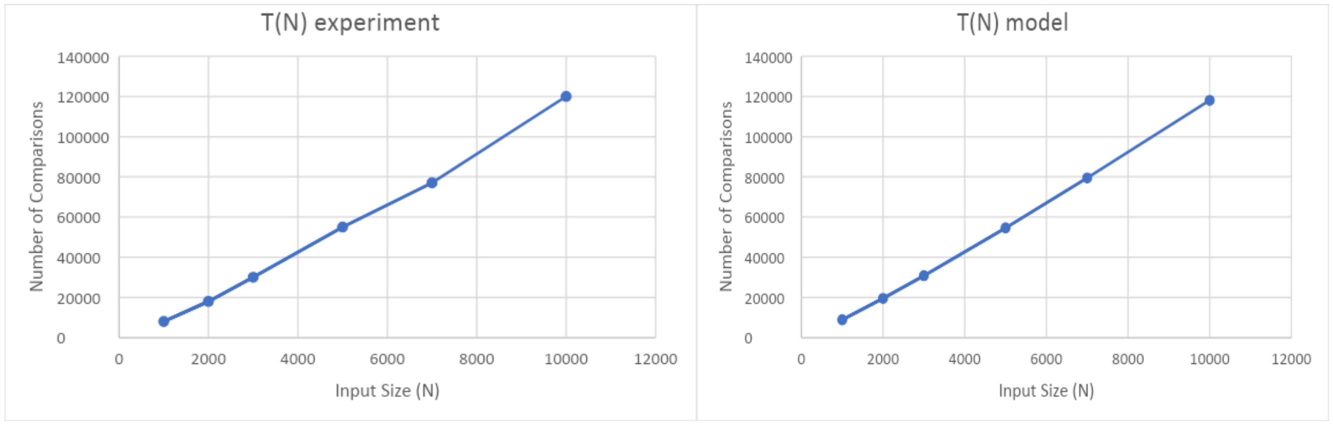
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	8015	18016	30018	55017	77019	120018
T(N) Model	8865	19498	30802	54604	79461	118084

b) Model Equation

$$T_{model}(N) = 0.888575*N*log_2(N)$$

c) Graph Representation

Figure 8 T(N) model and experiment graphs for Shell sort



9) Tree Sort

a) Tabular Results

Table 9 Tree Sort Algorithm

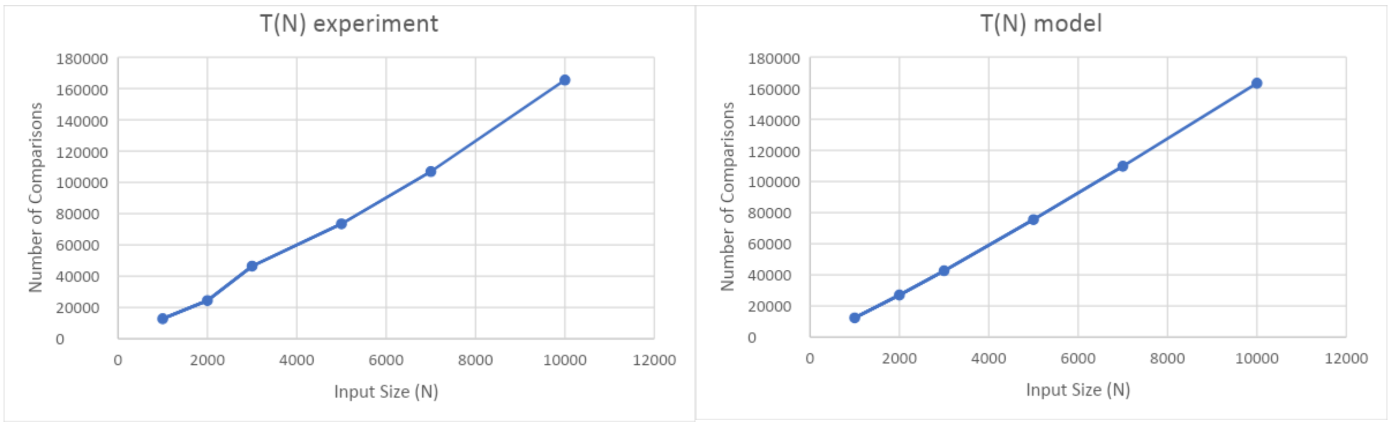
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	12688	24297	46367	73408	106896	165449
T(N) Model	12250	26944	42565	75456	109805	163176

b) Model Equation

$$T_{model}(N) = 1.22789*N*log_2(N)$$

c) Graphical Representation

Figure 9 T(N) model and experiment graphs for Tree sort



10) Comb Sort

a) Tabular Results

Table 10 Comb Sort Algorithm

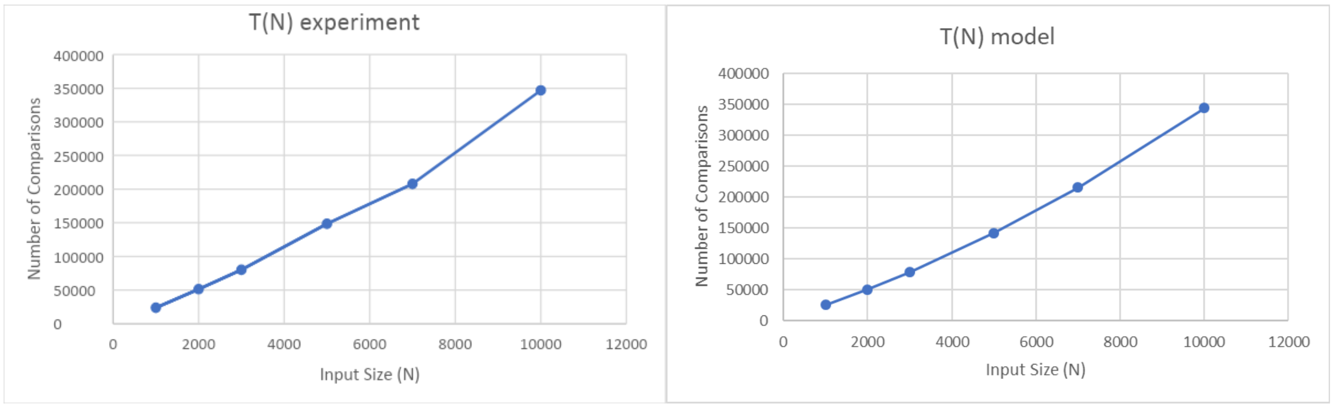
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	23730	51407	80080	148414	207752	346767
T(N) Model	25730	51096	78978	142289	215663	344591

b) Model Equation

$$T_{model}(N) = 2858.308 + 21.59016*N + 0.001257849*N^2$$

c) Graphical Representation

Figure 10 T(N) model and experiment graphs for Comb sort



11) Bubble Sort

a) Tabular Results

Table 11 Bubble Sort Algorithm

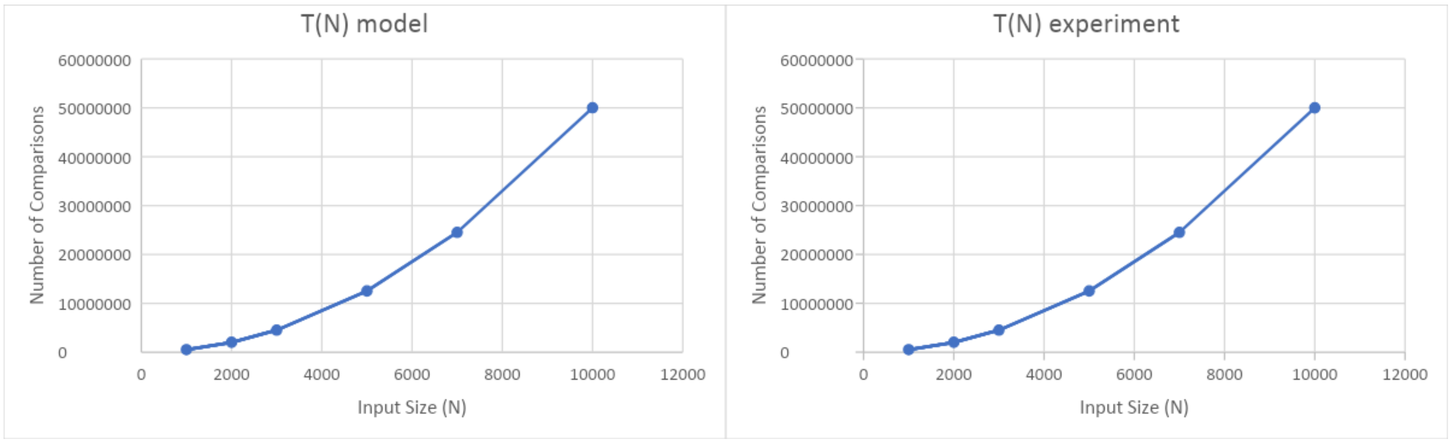
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	500500	2001000	4501500	12502500	24503500	50005000
T(N) Model	501501	2003001	4504501	12507501	24510501	50015001

b) Model Equation

$$T_{model}(N) = 1.862645e\text{-}9 + 0.5*N + 0.5*N^2$$

c) Graphical Representation

Figure 11 T(N) model and experiment graphs for Bubble sort



VII. Analysis and Critique:

Based on the experimental results, different sorting algorithms displayed varying levels of efficiency in sorting data sets of different sizes. It was anticipated that Counting and Radix sort would have the least number of comparisons as they are non-comparison based sorting algorithms. Consequently, it would be inappropriate to compare these two algorithms to the rest of the algorithms. However, it is crucial to acknowledge that these two sorting algorithms are efficient in minimizing the number of comparisons. Ignoring Counting and Radix sorting, Shell sort consistently displayed the least number of comparisons with all input sizes, starting from 8015 and increasing to 120018 comparisons. The second algorithm with the least number of comparisons is merge sort which starts with 8678 and increases to 120443 comparisons with the largest input size. In contrast, Bubble and Selection sort displayed the least efficiency, with the number of comparisons reaching 50,050,000 and 49,995,000, respectively, for the largest input size. Notably, the number of comparisons for Bubble and Selection sorts was considerably high when compared to similar algorithms such as Insertion, Comb, and Shell, which employ the methods of comparing and swapping elements using different techniques. For instance, Comb and Shell compare elements with a bigger gap than 1, while Insertion compares previous elements. The experimental results were similar with the model results, indicating a high level of accuracy. One limitation of the experiment is that it only considered the number of comparisons and did not take into account other factors that should be considered such as the time taken and memory usage. This makes the difference when comparing Quick and Merge sort for instance since although Merge sort performs better in terms of minimizing comparisons, Quick sort is more memory efficient. Another factor that would require further research is comparing the performance of sorting algorithms in different scenarios such as with already partially sorted arrays or arrays containing duplicates. Additionally, the experiment does not fully test Counting and Radix sorts capabilities since there are no duplicates in the array. In the end, divide-and-conquer algorithms were shown to be superior in minimizing the number of comparisons, along with Shell and the non-comparison based sorting algorithms.

VIII. Conclusion:

From the results of our comparative study, it has been concluded that each sorting algorithm varies based on the input data sizes and what type it is. However, we used it with array data types and concluded some findings. The findings shed light on how some algorithms are efficient for smaller input sizes, while others are more efficient for larger input sizes. Firstly, counting and radix sort work efficiently on large datasets because of taking little to no time in sorting as they are a non-comparison algorithm. They are highly stable algorithms , as all the equal elements in the array were still available in their right position of the sorted output array. On the other hand, Bubble and Selection sort showed otherwise as they are less efficient form larger datasets, as they have the most amount of comparisons and time taken to sort the array given. However, that does not mean they can not work efficiently on smaller datasets.

Our research has shown the potential limitations that can make a sorting algorithm unfit for a user's application. A sorting algorithm might work on a data type or data structure better than the other. Therefore, it is important to note that just because we used an array data type in our research paper doesn't indicate that the sorting algorithm can't be applied effectively with other data types. Additionally, there are many characteristics that distinguish each sorting algorithm from the other such as the memory space needed, time complexity, stability, and adaptability to any input data type. However, due to our team’s time and hardware constraints, it was best to measure the comparisons and just consider other important characteristics when analyzing our data, from which we concluded some results about each sorting algorithm. It is important to consider the specific needs of the application before opting for a sorting algorithm since some might prioritize the space complexity over the time complexity. Therefore, the industry needs future research that considers all of the factors that can play in distinguishing all sorting algorithms –even the non-famous ones– from one another. Additionally, further research in the field is required to continue improving the efficiency and effectiveness of sorting algorithms for various data structures, applications and sizes.

References

https://www.geeksforgeeks.org/sorting-algorithms/

https://www.geeksforgeeks.org/insertion-sort/

https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/quick-sort/

https://www.geeksforgeeks.org/heap-sort/

https://www.geeksforgeeks.org/counting-sort/

https://www.geeksforgeeks.org/radix-sort/

https://www.geeksforgeeks.org/selection-sort/

https://www.geeksforgeeks.org/bubble-sort/

https://www.geeksforgeeks.org/shellsort/

https://www.geeksforgeeks.org/tree-sort/

https://www.geeksforgeeks.org/comb-sort/

https://www.programiz.com/dsa/sorting-algorithm

https://www.youtube.com/watch?v=o4bAoo_gFBU&list=PLuZ_bd9XlByzTIP5j1aWXo7smCIx vzd2D

Dr. Howeida's Slides

Gill, Sandeep Kaur, et al. "A Comparative Study of Various Sorting Algorithms." Papers.ssrn.com, 2018, papers.ssrn.com/sol3/papers.cfm?abstract_id=3329410.

Appendix A

Here are the UML class diagrams for our program. The classes are separated based on a sorting algorithm and the screenshots found below are split to show segments of the full diagram before revealing the entire diagram. The UML diagrams were designed using starUML and are the most recently updated versions.

Figure A1 UML diagrams for insertion and bubble sort

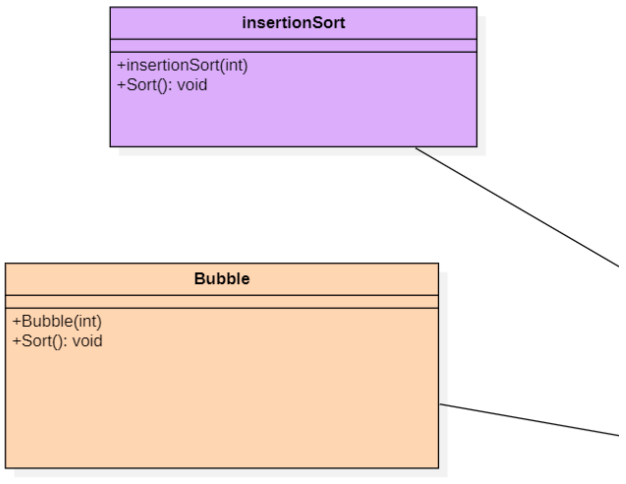


Figure A2 UML diagrams for merge and counting sort

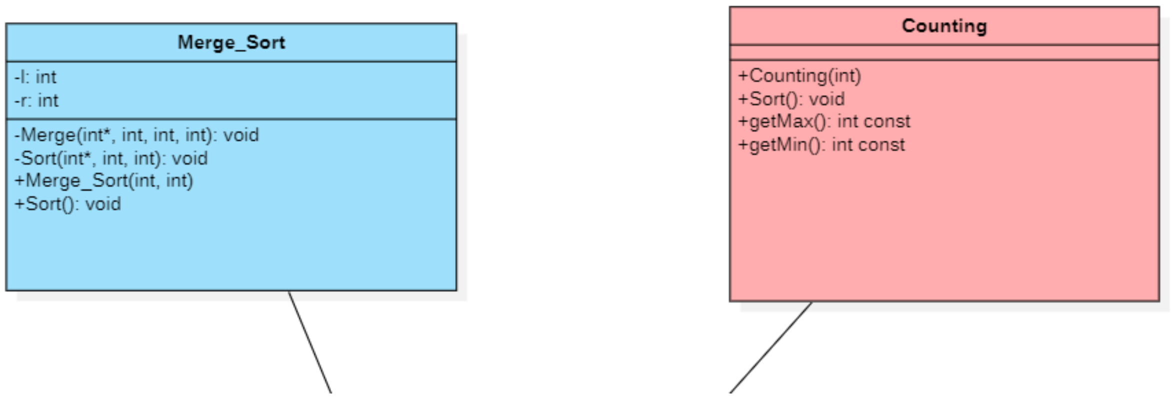


Figure A3 UML diagrams for heap and tree sort and the BST class

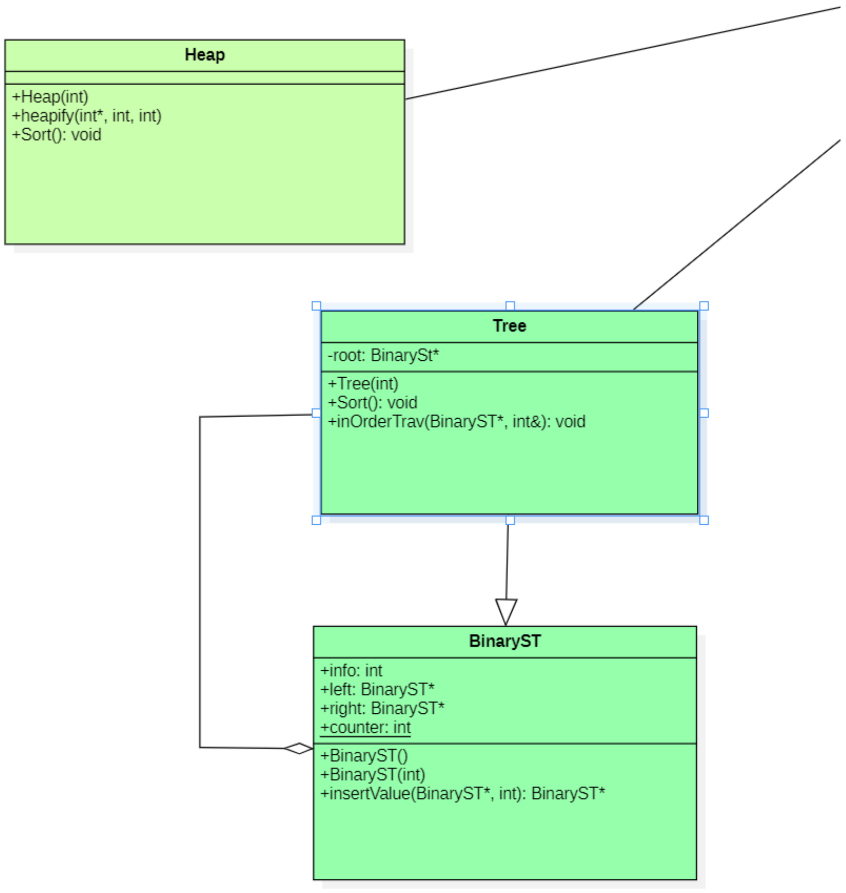


Figure A4 UML diagrams for Shell and both pivot instances of Quick sort

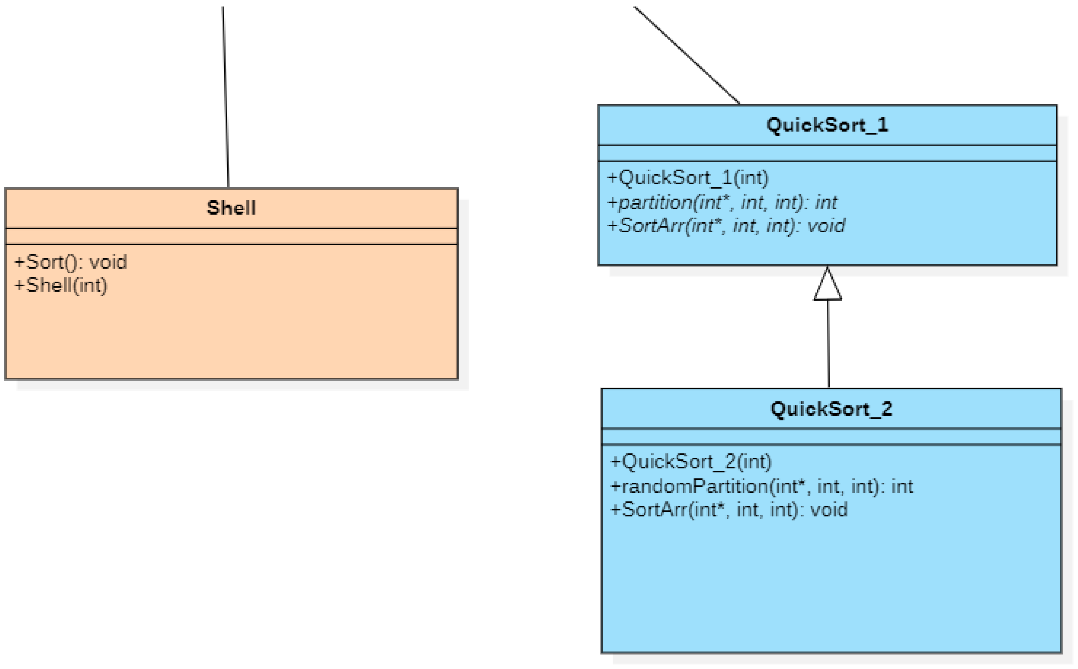

Shell
+Sort(): void
+Shell(int)
QuickSort_1
+QuickSort_1(int)
+partition(int*, int, int): int
+SortArr(int*, int, int): void
QuickSort_2
+QuickSort_2(int)
+randomPartition(int*, int, int): int
+SortArr(int*, int, int): void

Figure A5 UML diagrams for comb, selection and radix sort

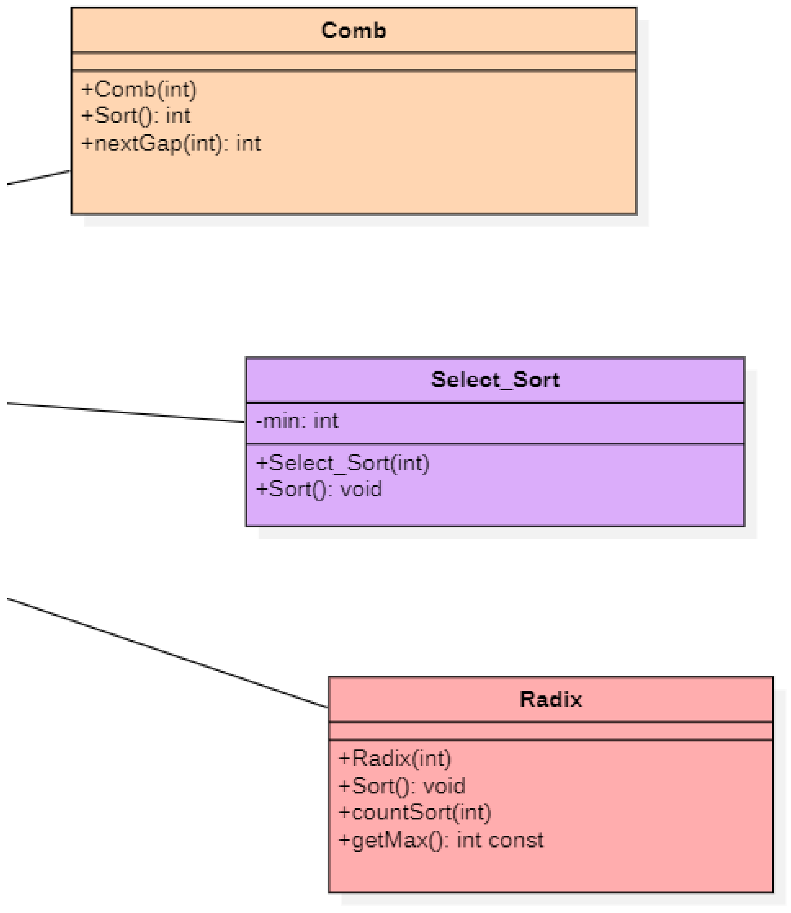


Figure A6 UML diagrams for the base class called "Sorting"

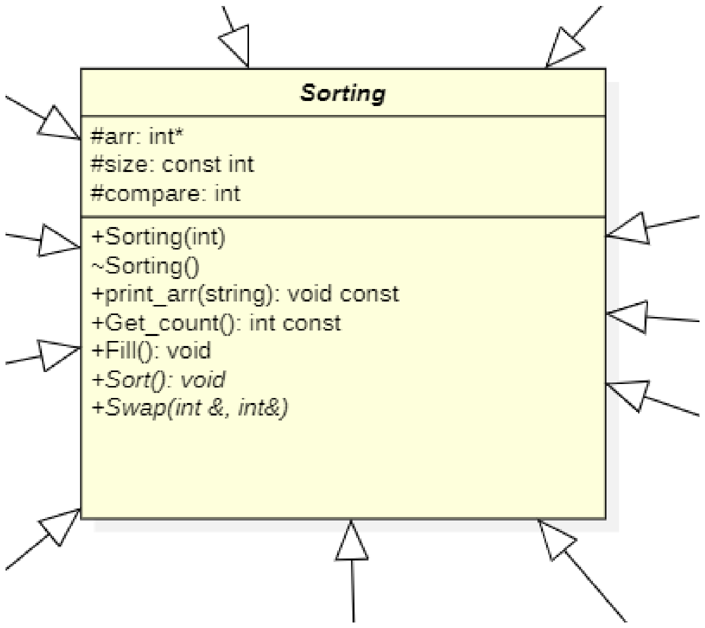
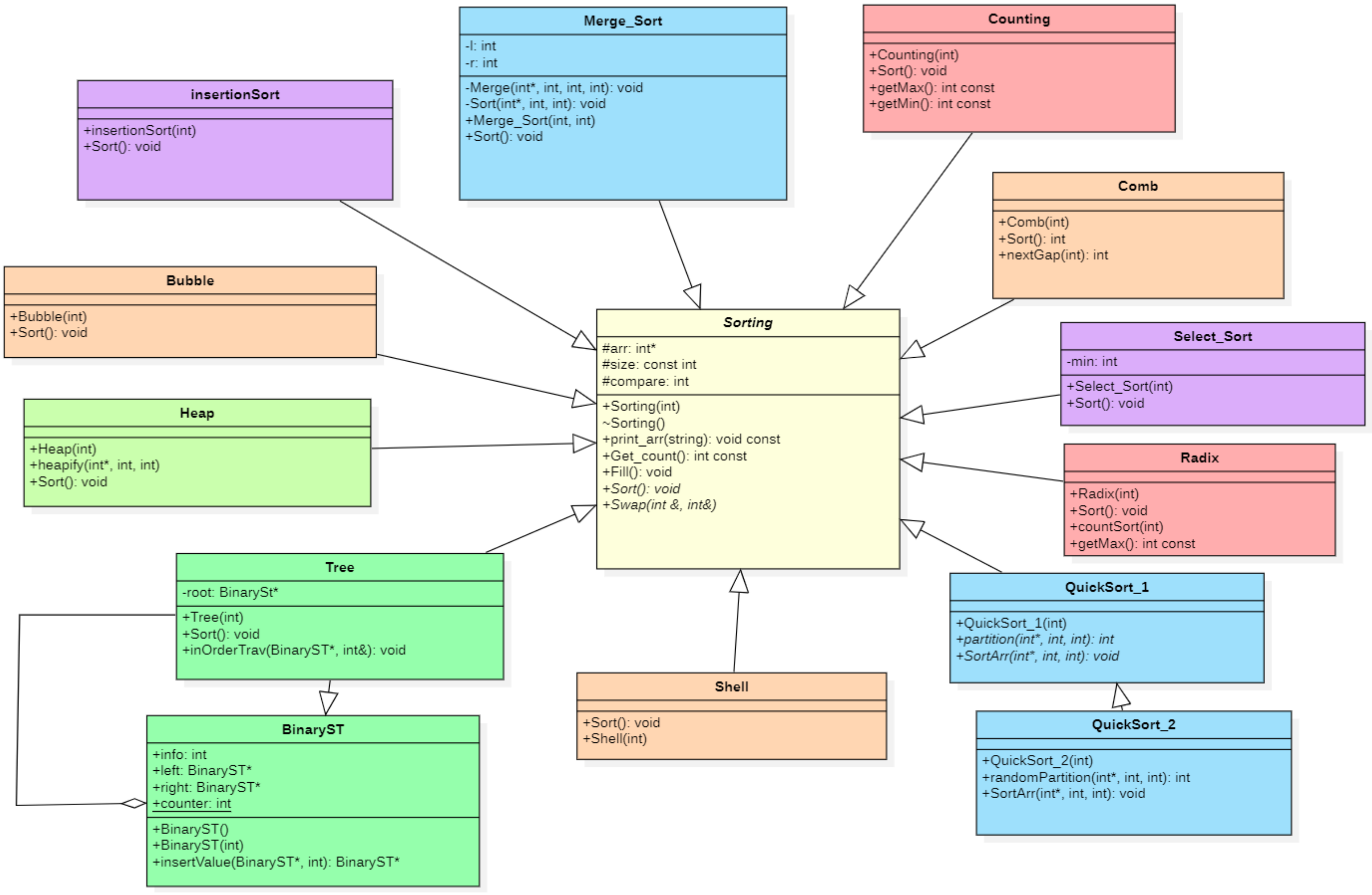


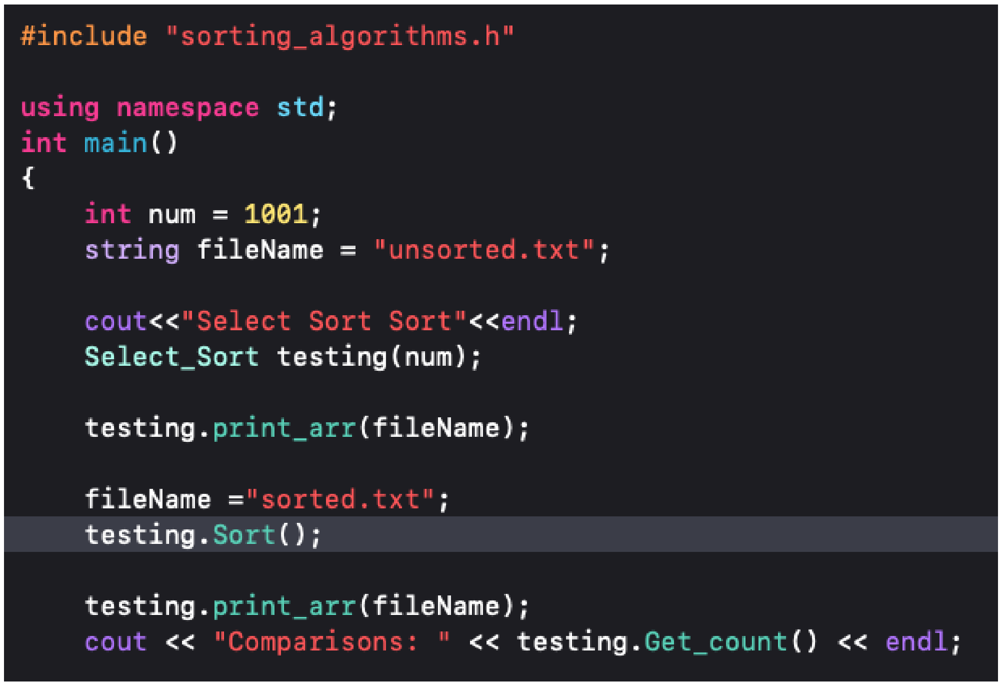
Figure A7 Entire UML diagram for program



Appendix B

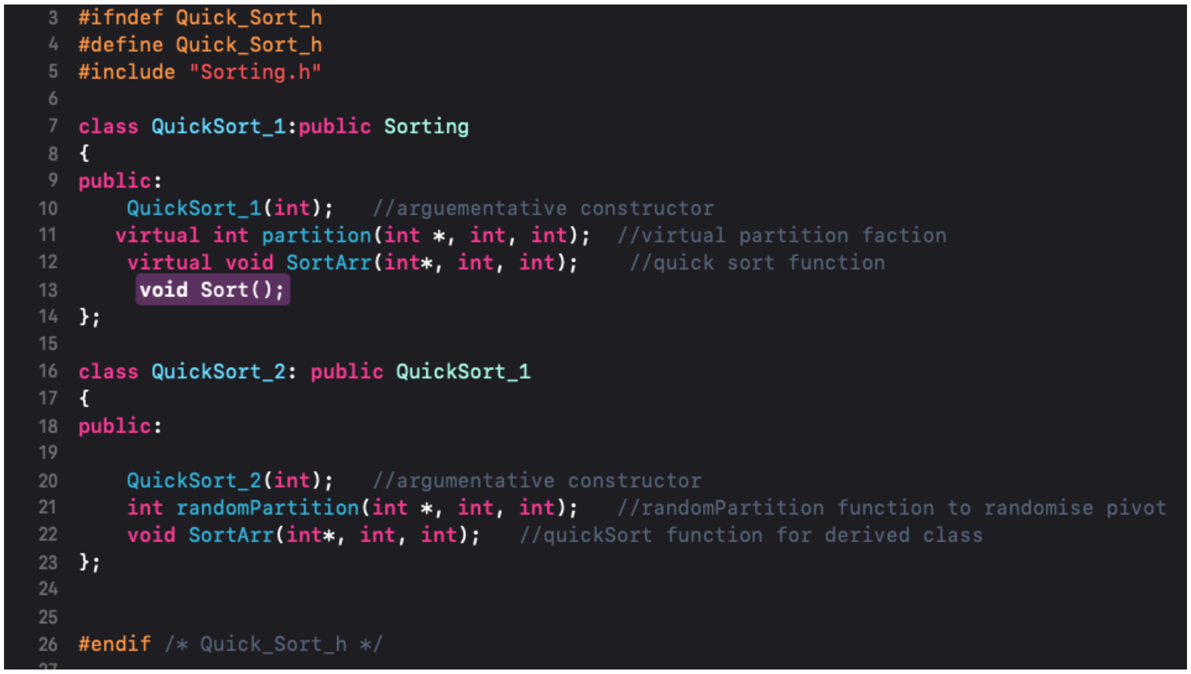
Here are code snippets of the main file and header files that were used in the program. They consist of header files for each class seen in ***Appendix A*** along with an additional header file that contains all of the #includes needed for the main to avoid clutter in the main function.

Figure B1 Code snippet from the main showing the general code for testing the algorithms



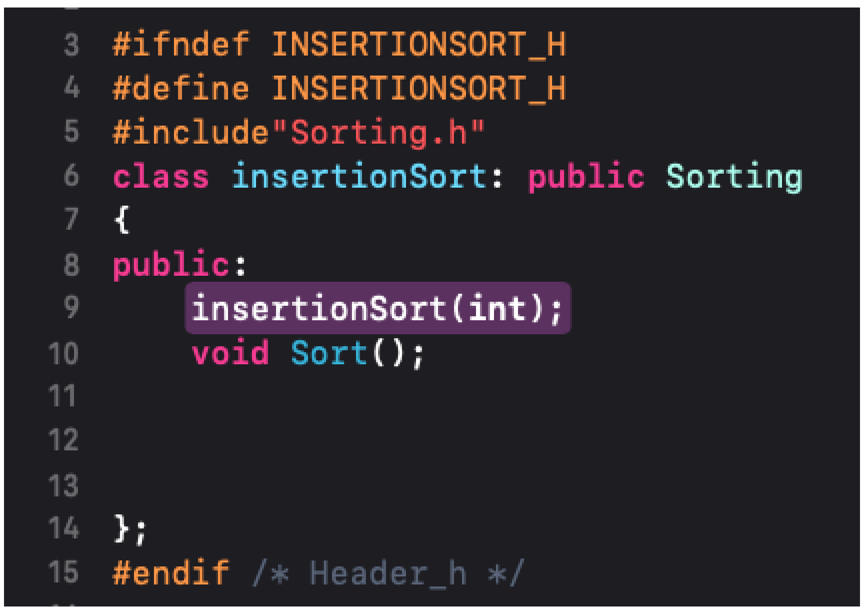
```
#include "sorting_algorithms.h"

using namespace std;
int main()
{
    int num = 1001;
    string fileName = "unsorted.txt";

    cout<<"Select Sort Sort"<<endl;
    Select_Sort testing(num);

    testing.print_arr(fileName);

    fileName ="sorted.txt";
    testing.Sort();

    testing.print_arr(fileName);
    cout << "Comparisons: " << testing.Get_count() << endl;
```

Figure B2 Code snippet showing the Quick sort header file



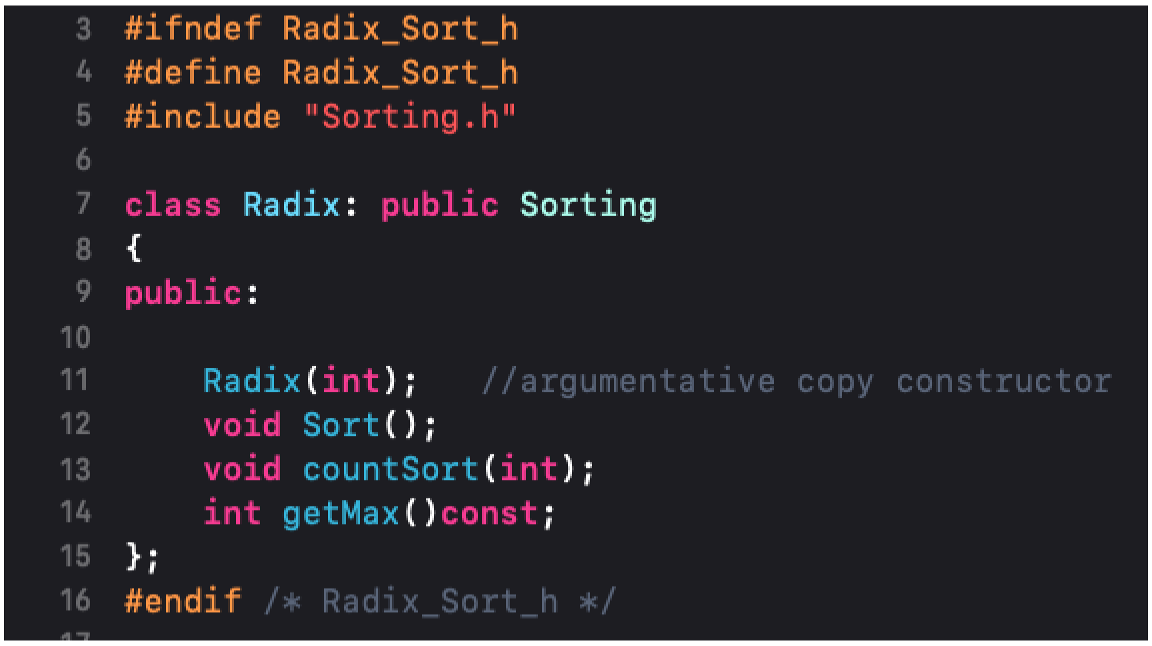
```
#ifndef Quick_Sort_h
#define Quick_Sort_h
#include "Sorting.h"

class QuickSort_1:public Sorting
{
public:
    QuickSort_1(int);   //arguementative constructor
   virtual int partition(int *, int, int);  //virtual partition faction
    virtual void SortArr(int*, int, int);    //quick sort function
     void Sort();
};

class QuickSort_2: public QuickSort_1
{
public:

    QuickSort_2(int);   //argumentative constructor
    int randomPartition(int *, int, int);   //randomPartition function to randomise pivot
    void SortArr(int*, int, int);   //quickSort function for derived class
};


#endif /* Quick_Sort_h */
```

Figure B3 Code snippet showing the Insertion sort header file



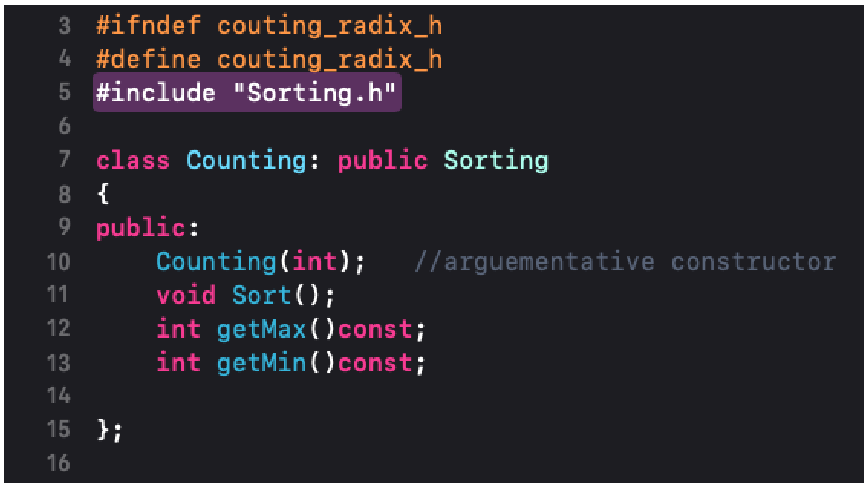
```cpp
#ifndef INSERTIONSORT_H
#define INSERTIONSORT_H
#include"Sorting.h"
class insertionSort: public Sorting
{
public:
    insertionSort(int);
    void Sort();



};
#endif /* Header_h */
```

Figure B4 Code snippet showing the Radix sort header file



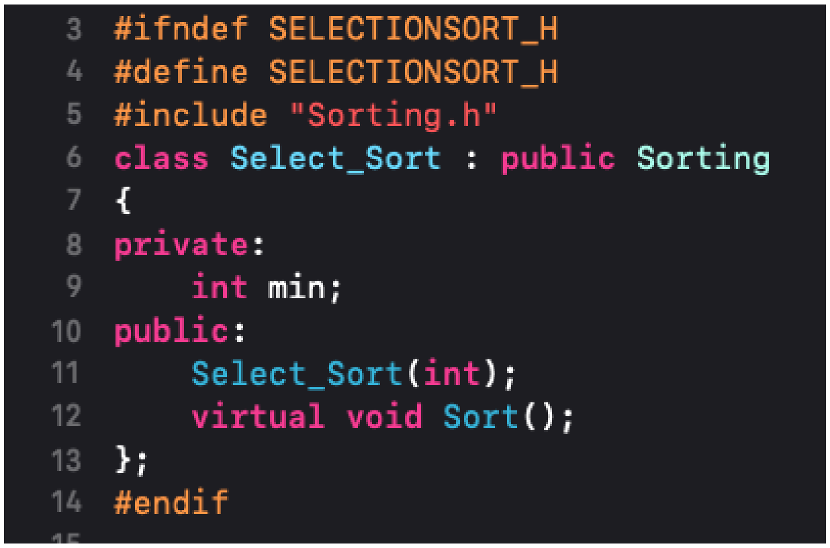
```cpp
#ifndef Radix_Sort_h
#define Radix_Sort_h
#include "Sorting.h"

class Radix: public Sorting
{
public:

    Radix(int);    //argumentative copy constructor
    void Sort();
    void countSort(int);
    int getMax()const;
};
#endif /* Radix_Sort_h */
```

Figure B5 Code snippet showing the Counting sort header file



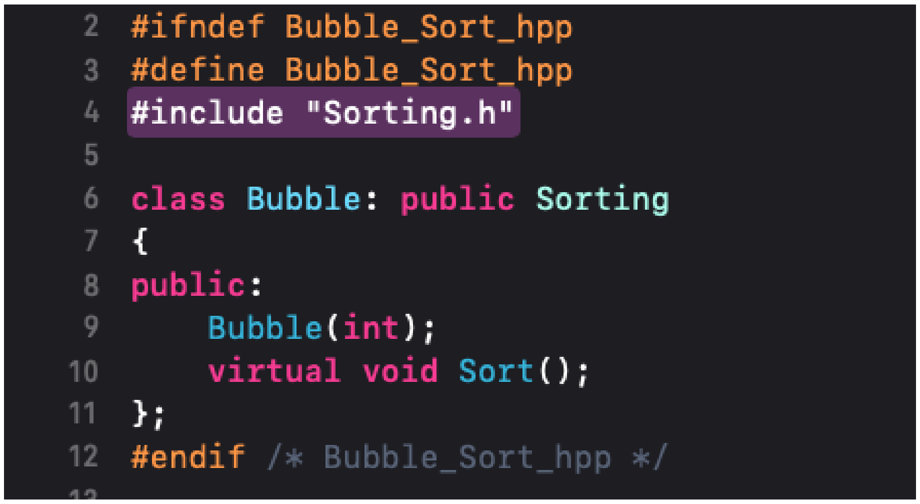
```cpp
#ifndef couting_radix_h
#define couting_radix_h
#include "Sorting.h"

class Counting: public Sorting
{
public:
    Counting(int);    //arguementative constructor
    void Sort();
    int getMax()const;
    int getMin()const;

};

```

Figure B6 Code snippet showing the Selection sort header file



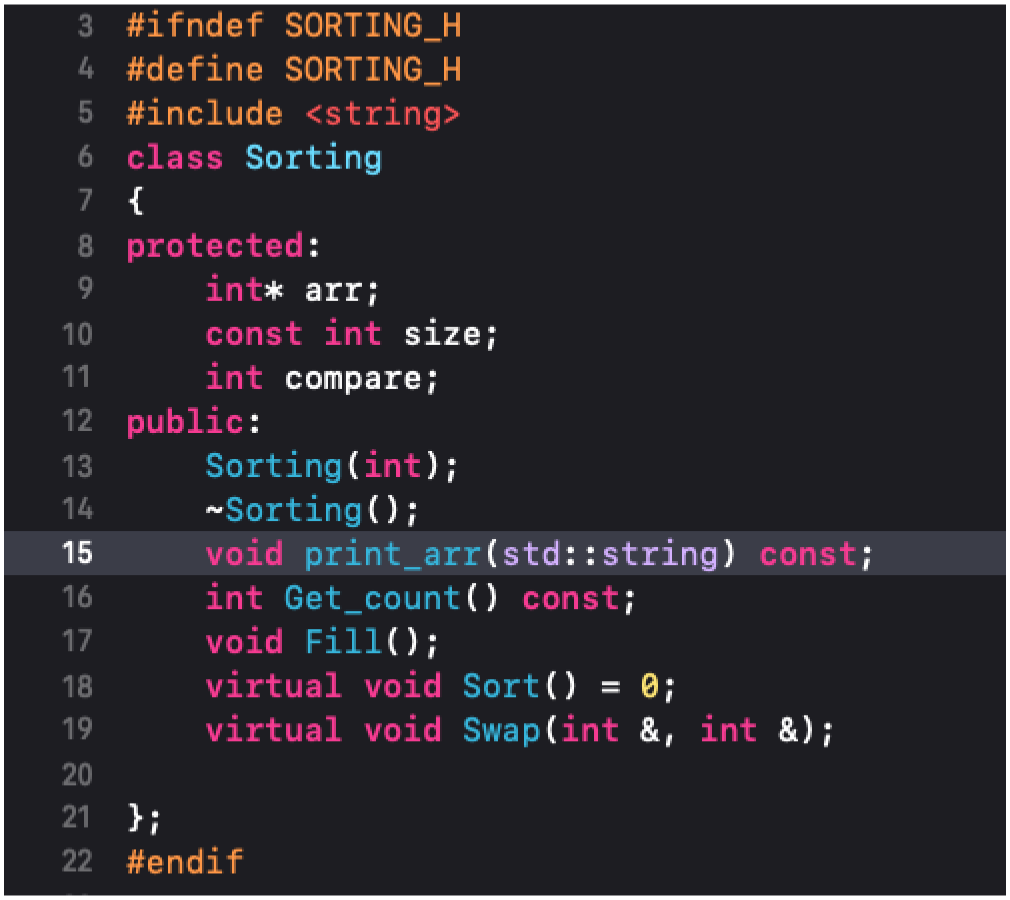
```cpp
#ifndef SELECTIONSORT_H
#define SELECTIONSORT_H
#include "Sorting.h"
class Select_Sort : public Sorting
{
private:
    int min;
public:
    Select_Sort(int);
    virtual void Sort();
};
#endif
```

Figure B7 Code snippet showing the Bubble sort header file



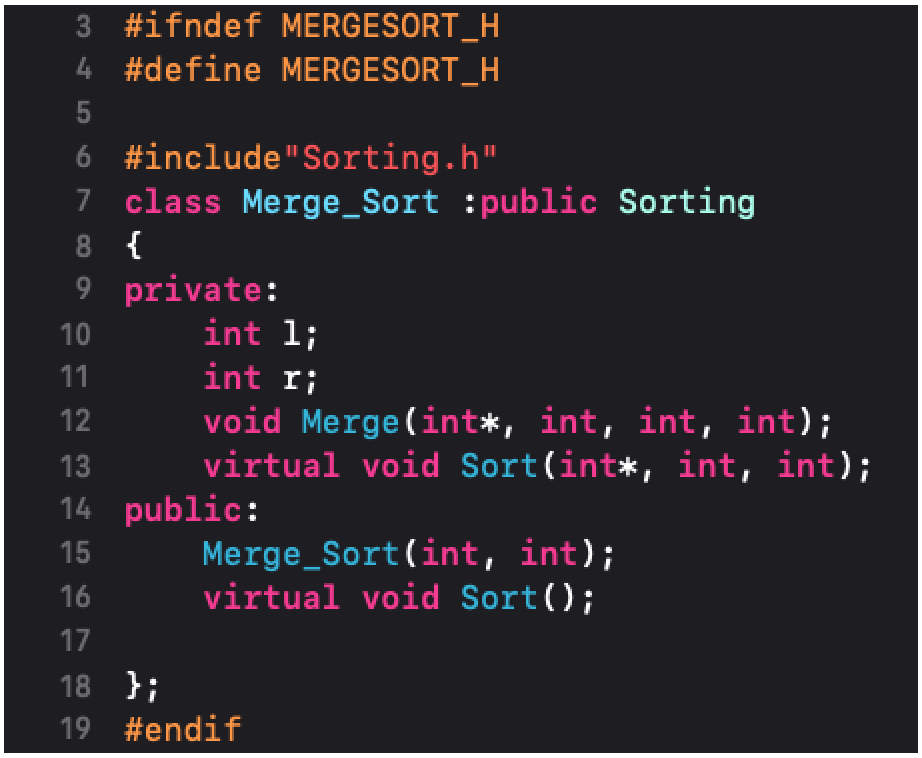
```cpp
#ifndef Bubble_Sort_hpp
#define Bubble_Sort_hpp
#include "Sorting.h"

class Bubble: public Sorting
{
public:
    Bubble(int);
    virtual void Sort();
};
#endif /* Bubble_Sort_hpp */
```

Figure B9 Code snippet showing the base class “Sorting” header file



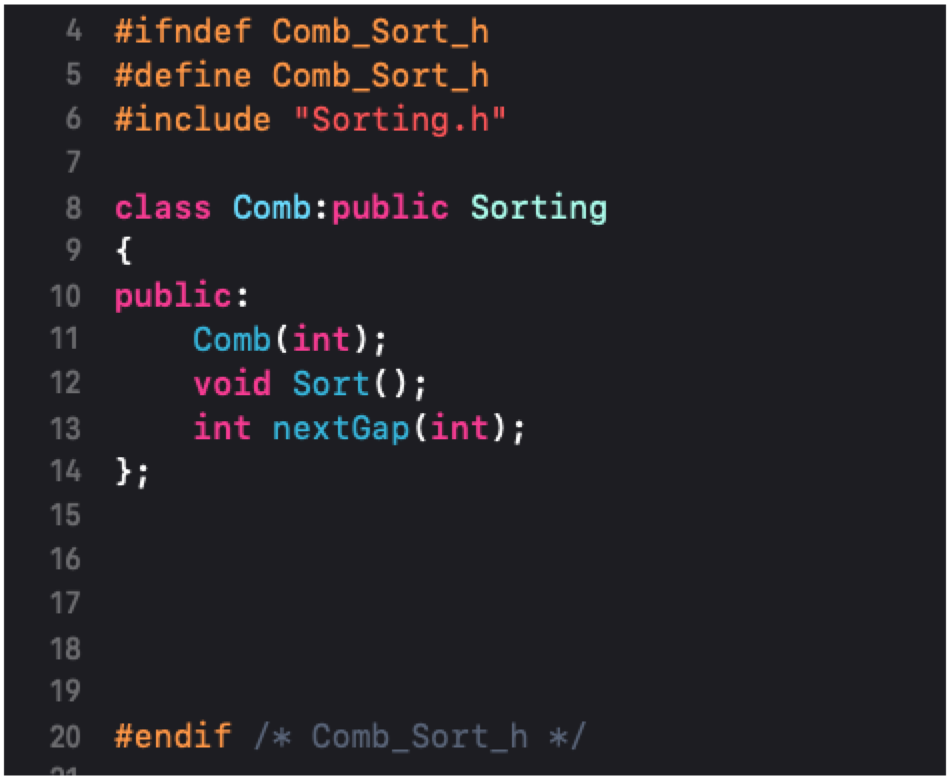
```
#ifndef SORTING_H
#define SORTING_H
#include <string>
class Sorting
{
protected:
    int* arr;
    const int size;
    int compare;
public:
    Sorting(int);
    ~Sorting();
    void print_arr(std::string) const;
    int Get_count() const;
    void Fill();
    virtual void Sort() = 0;
    virtual void Swap(int &, int &);

};
#endif
```

Figure B10 Code snippet showing the Merge sort header file



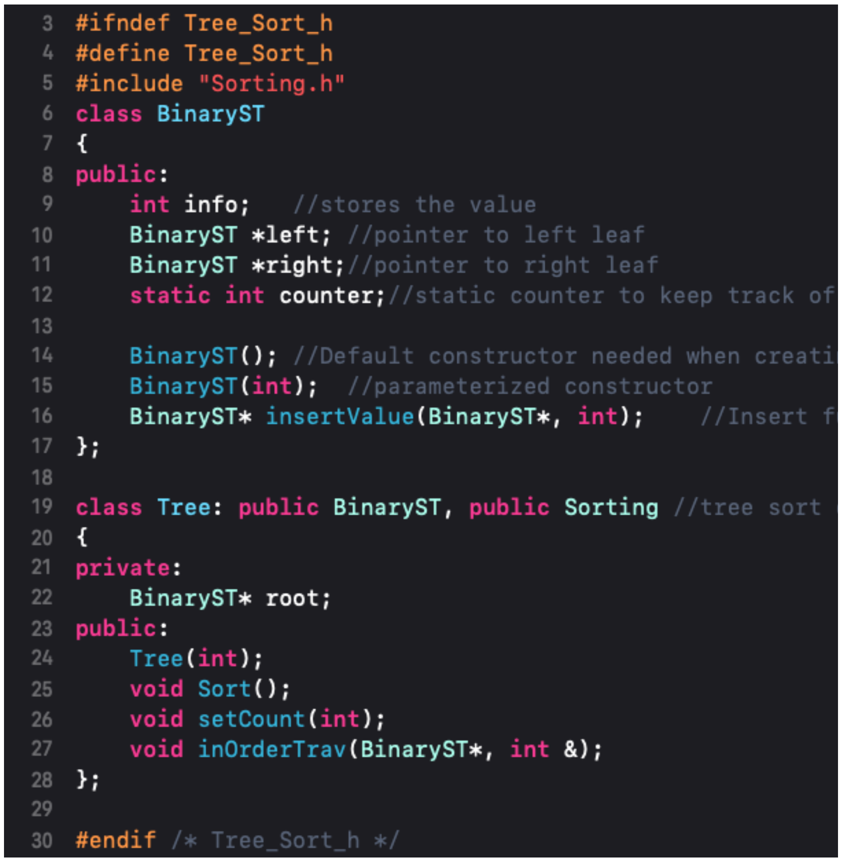
```
#ifndef MERGESORT_H
#define MERGESORT_H

#include"Sorting.h"
class Merge_Sort :public Sorting
{
private:
    int l;
    int r;
    void Merge(int*, int, int, int);
    virtual void Sort(int*, int, int);
public:
    Merge_Sort(int, int);
    virtual void Sort();

};
#endif
```

Figure B11 Code snippet showing the Comb sort header file



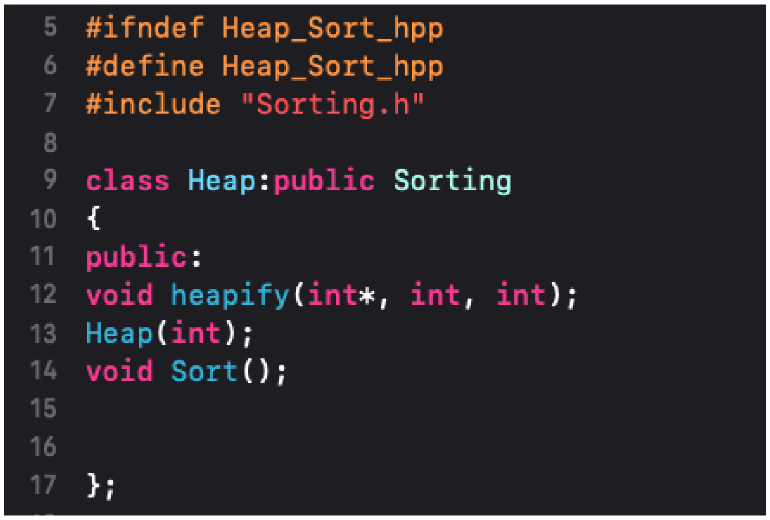
```
#ifndef Comb_Sort_h
#define Comb_Sort_h
#include "Sorting.h"

class Comb:public Sorting
{
public:
    Comb(int);
    void Sort();
    int nextGap(int);
};






#endif /* Comb_Sort_h */
```

Figure B12 Code snippet showing the Tree sort and BST header file



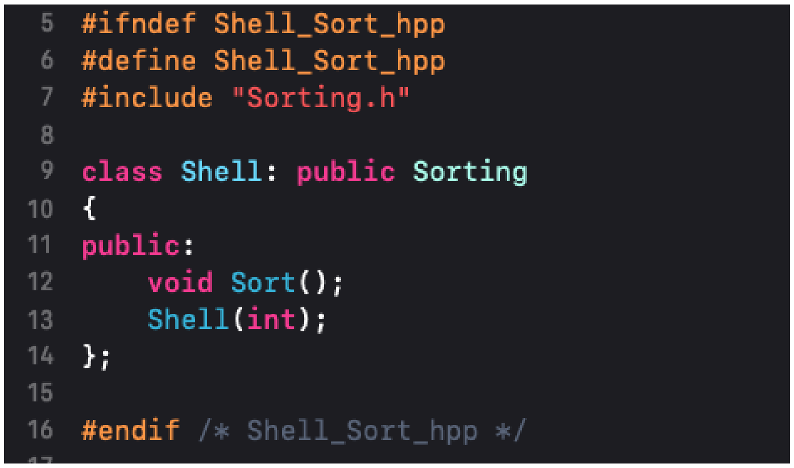
```
#ifndef Tree_Sort_h
#define Tree_Sort_h
#include "Sorting.h"
class BinaryST
{
public:
    int info;   //stores the value
    BinaryST *left; //pointer to left leaf
    BinaryST *right;//pointer to right leaf
    static int counter;//static counter to keep track of

    BinaryST(); //Default constructor needed when creati
    BinaryST(int);  //parameterized constructor
    BinaryST* insertValue(BinaryST*, int);    //Insert f
};

class Tree: public BinaryST, public Sorting //tree sort
{
private:
    BinaryST* root;
public:
    Tree(int);
    void Sort();
    void setCount(int);
    void inOrderTrav(BinaryST*, int &);
};

#endif /* Tree_Sort_h */
```

Figure B13 Code snippet showing the Heap sort header file



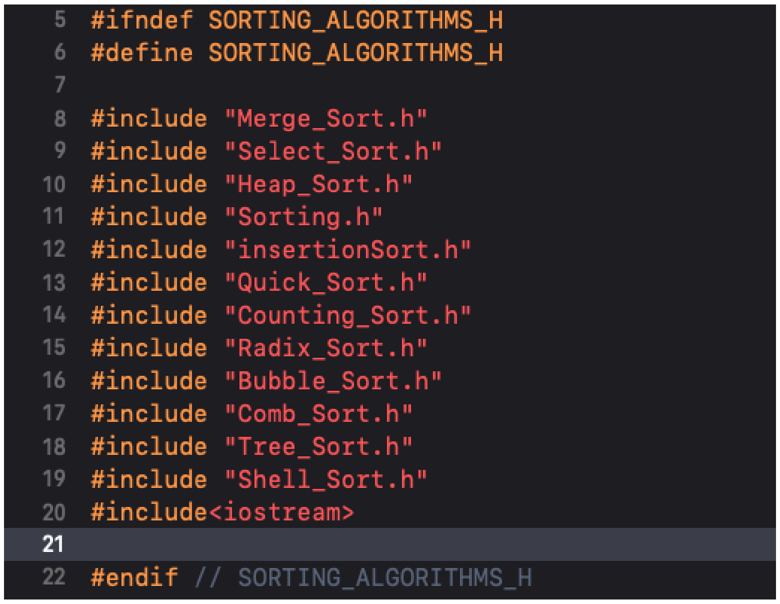
```cpp
#ifndef Heap_Sort_hpp
#define Heap_Sort_hpp
#include "Sorting.h"

class Heap:public Sorting
{
public:
void heapify(int*, int, int);
Heap(int);
void Sort();


};
```

Figure B14 Code snippet showing the Shell sort header file



```cpp
#ifndef Shell_Sort_hpp
#define Shell_Sort_hpp
#include "Sorting.h"

class Shell: public Sorting
{
public:
    void Sort();
    Shell(int);
};

#endif /* Shell_Sort_hpp */
```

Figure B15 Code snippet showing the header file with all the #includes



```cpp
#ifndef SORTING_ALGORITHMS_H
#define SORTING_ALGORITHMS_H

#include "Merge_Sort.h"
#include "Select_Sort.h"
#include "Heap_Sort.h"
#include "Sorting.h"
#include "insertionSort.h"
#include "Quick_Sort.h"
#include "Counting_Sort.h"
#include "Radix_Sort.h"
#include "Bubble_Sort.h"
#include "Comb_Sort.h"
#include "Tree_Sort.h"
#include "Shell_Sort.h"
#include<iostream>

#endif // SORTING_ALGORITHMS_H
```