A Survey, Discussion and Comparison of Sorting Algorithms

by
Ashok Kumar Karunanithi

Department of Computing Science
Umeå University

Master's Thesis, 30hp
Supervisor: Frank Drewes
Examiner: Jerry Eriksson

June 2014

Abstract

Sorting is a basic task in many types of computer applications. Especially when large amounts of data are to be sorted, efficiency becomes a major issue. There are many different sorting algorithms and even more ways in which they can be implemented. The efficiency of real implementations is often at least as important as the theoretical efficiency of the abstract algorithm. For example, Quicksort is well-known to perform very well in most practical situations, regardless of the fact that many other sorting algorithms have a better worst-case behaviour. The goal of this master thesis is to make a survey of sorting algorithms and discuss and compare the differences in both theory and practice. There are several features that interests in this thesis such as finding possible implementations of each algorithm and discuss their properties and administer considerable experiments in order to obtain empirical data about their practical efficiency in different situations. Finally we present the comparison of different sorting algorithms with their practical efficiency and conclude the theoretical findings and the knowledge gained from this study.

Acknowledgement

I would like to express my sincere gratitude to supervisor Frank Drewes for his valuable support, comments and help throughout this work.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In computer science, sorting is an essential work for many applications towards searching and locating a prominent number of data. General description of sorting believed to be the process of rearranging the data into a particular order. The orders used are either in numerical order or lexicographical order. Sorting arranges the integer data into increasing or decreasing order and an array of strings into alphabetical order. It may also be called as ordering the data. Sorting is considered as one of the most fundamental tasks in many computer applications for the reason that searching a sorted array or list takes less time when compared to an unordered or unsorted list [8].

There have been many attempts made to analyze the complexity of sorting algorithms and many interesting and good sorting algorithms have been proposed. There are more advantages in the study of sorting algorithms in addition to understanding the sorting methods. These studies have gained a significant amount of power to solve many other problems. Even though sorting is one of the extremely studied problems in computer science, it remains the most general integrative algorithm problem in practice [18].

Moreover, each algorithm has its own advantages and disadvantages. For instance, bubble sort would be efficient to sort a small number of items, . On the other hand, for a large number of items quick sort would perform very well. Therefore, it is not perpetually thinkable that one sorting method is better than another sorting method. Moreover the performance of each sorting algorithm relies upon the data being sorted and the machine used for sorting [18].

In general, simple sorting algorithms perform two operations such as compare two elements and assign one element. These operations proceed over and over until the data is sorted [20]. Moreover, selecting a good sorting algorithm depending upon several factors such as the size of the input data, available main memory, disk size, the extent to which the list is already sorted and the distribution of values [18]. To measure the performance of different sorting algorithm we need to consider the following facts such as the number of operations performed, the execution time and the space required for the algorithm [12].

Since sorting algorithms are common in computer science, some of its context contributes to a variety of core algorithm concepts such as divide-and-conquer algorithms, data structures, randomized algorithms, etc. The majority of an algorithm in use have an algorithmic efficiency of either $O(n^2)$ or $O(n \log n)$.

1.1 Aim of the thesis

The main objective of this thesis is to make a survey of sorting algorithms based on a literature study and their possible implementations. The algorithms will be defined and explained and their efficiency and other properties will be discussed and compared in detail. The literature will be searched for different flavors and improvements of the basic algorithms, and their reported advantages and disadvantages will be summarized and discussed.

This study also aims to collect the runtime data of different sorting algorithms in order to compare their implementation variants from a practical perspective. Quantities of interest include the number of operations of different types that are executed, the resulting absolute running time, and the memory space consumption. Such data can be collected by using a tailor-made package. The goal is to gather and evaluate data that makes a detailed analysis of the practical efficiency of the algorithms and their concrete implementations possible. The experiment will be repeated on different Test Cases for instance, different types of input data, such as arrays of numbers and strings of different sizes.

1.2 Outline of the thesis

The section 2 describes the fundamentals of analysis of algorithms with worst case, best case and average case running time analysis proceeds with a basic introduction to growth of the function and comparison of growth rate functions. In addition, a general introduction to running time calculation and classification of sorting algorithms are discussed similarly.

The section 3 describes the survey of sorting algorithms that considered in this thesis. Besides each algorithm are defined and explained clearly and their efficiency and other properties are discussed. Finally their advantages and disadvantages are summarized.

The section 4 describes the design specification and practical implementation of various sorting algorithms. Besides quantities of interest and different test case scenarios are explained.

In section 5, we analyzed the results of different test cases mentioned in section 4 and compared various sorting algorithms from the practical perspective with the help of observed results.

The section 6 describes the discussion and summary of various sorting algorithms based on the insight gained from the previous sections and concluded this section with some problem examples and suggested suitable algorithms for that example.

1.3 Limitations

Since there are many sorting algorithms are advanced, it is not possible to consider all of them in this study. Therefore only basic and the most popular algorithms are considered and analyzed.

Chapter 2

Background

2.1 Analysis of Algorithm

The analysis of algorithm defines that the estimation of resources required for an algorithm to solve a given problem. Sometimes the resources include memory, time and communication bandwidth. In which running time and memory required are of primary concern for the reason that algorithms that needed a month or year to solve a problem is not useful. Besides that, it also requires gigabytes of main memory to solve a problem and is not efficient. In general, to find a suitable algorithm for a specific problem we have to analyze several possible algorithms. By doing so, we might locate more than one useful algorithm. One way to recognize the best suitable algorithm for a given problem is to implement both algorithms and find out their efficiency, most importantly the running time of a program. If the observed running time matches that predicted running time of the analysis and also outperform the other algorithm that would be the best suitable algorithm for the given problem. Generally various factors affect the running time of a program in which the size of input is the primary concern. Moreover, most of the basic algorithms perform very well in small arrays and takes longer time for bigger size. Typically, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input [9]. Furthermore, To analyze an algorithm we should use mathematical techniques that investigate algorithms independently of specific implementations, hardware and data. This methodology is known as asymptotic algorithm analysis.

2.2 Running Time Analysis

The running time analysis is a theoretical process to categorize the algorithm into a relative order among function by predicting and calculating approximately the increase in running time of an algorithm as its input size increases. For instance a program can take seconds, hours or even years to complete the execution, usually this depends upon the particular algorithm used to construct the program. Besides the run time of a program describes the number of primary operations executed during implementation.

To ensure the execution time of an algorithm we should anticipate the worst case, average case and best case performance of an algorithm. These analyze, assist the understanding of algorithm complexity.

2.2.1 Worst-Case Analysis

The worst case analysis anticipates the greatest amount of running time that an algorithm needed to solve a problem for any input of size n. The worst case running time of an algorithm gives us an upper bound on the computational complexity. Significantly it also guarantees that the performance of an algorithm will not get worse. In general, we consider worst case performance of an algorithm very often.

2.2.2 Best-Case Analysis

The best case analysis anticipates the least amount of running time that an algorithm needs to solve a problem for any input of size n. In this the running time of an algorithm gives us a lower bound on the computational complexity. Most of the analysts do not consider the best case performance of an algorithm for the reason that it is not useful.

2.2.3 Average-Case Analysis

The average case analysis anticipates the average amount of running time that an algorithm needed to solve a problem for any input of size n. Generally the average case running time is considered approximately as bad as the worst case time. However from a practical point of view, it is frequently useful to review the performance of an algorithm if we average its behavior over all possible sets of input data. One of the obstacles in the average case analysis is that it is much more difficult to carry out the process and typically requires considerable mathematical refinement, for that reason worst case analysis became prevalent.

2.3 Growth of Function

2.3.1 Introduction

As mentioned before, the main reason to analyze an algorithm is to anticipate the running time of an algorithm. Typically, we should investigate the computational complexity of an algorithm independent of the programming language, programming styles and computers used. However the data being solved must be concerned without the specific instances of the problem being solved [30]. According to the definition, the growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows [22]. This allows us to compare the relative performance of different algorithms. For instance, when the input size n increased to a greater extent, merge sort with its $n \log n$ worst case running time overcomes insertion sort whose worst case running time is n^2. It defines that the merge sort is asymptotically more efficient than insertion sort. We will discuss more about asymptotic in the following sections.

2.3.2 Asymptotic Notation

In computer science, generally we use some notation to illustrate the asymptotic running time of algorithms which are defined in terms of functions whose domains are the set of natural numbers and sometimes real numbers [9]. There are five standard asymptotic notations advanced in which we examine three basic asymptotic notations such as Big-O, Big-Ω and Big-Θ.

Big-O Notation

The Big-O notation is used to describe the tight upper bound on a function, which states that the maximum amount of resources needed by an algorithm to complete the execution.

According to the definition in [9], Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ (or $f(n) \in O(g(n))$) if there exists a real constant $c > 0$ and there exists an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every integer $n \geq n_0$.

Big-Ω Notation

The Big-Ω notation is used to describe the tight lower bound on a function, which states that the minimum amount of resources needed by an algorithm to complete the execution.

According to the definition in [9], Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $\Omega(g(n))$ (or $f(n) \in \Omega(g(n))$) if there exists a real constant $c > 0$ and there exists an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every integer $n \geq n_0$.

Big-Θ Notation

The Big-Θ notation is used to describe a function that has both tight upper bound and tight lower bound.

According to the definition in [9], Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $\Theta(g(n))$ (or $f(n) \in \Theta(g(n))$) if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

2.3.3 Standard Growth Rates

There is various growth rates advanced to describe the running time of a given algorithm. The following table defines each function with their name.

Function	Growth Rate Name
c	Constant
$logn$	Logarithmic
log^2n	Log-Squad
n	Linear
$nlogn$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Table 2.1: Different Growth Rate Functions

As mentioned before the growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows. The growth rate c describes a constant running time. The algorithm with a logarithmic growth rate considers being best suitable for many problems for the reason that it requires the least amount of resources. Likewise the algorithm with quadratic growth rate considers the worst case performance with greater running time equation, however better than cubic and exponential. The linear growth rate defines that as the size of the input grows, the running time of the algorithm grows in the same proportion. Besides doubling the size of input approximately doubles the running time. The exponential growth rate function describes that the algorithm with a running time grows exponentially with the increase of input size n. A graph will describe the comparison between these various growth rate functions in the following [29].

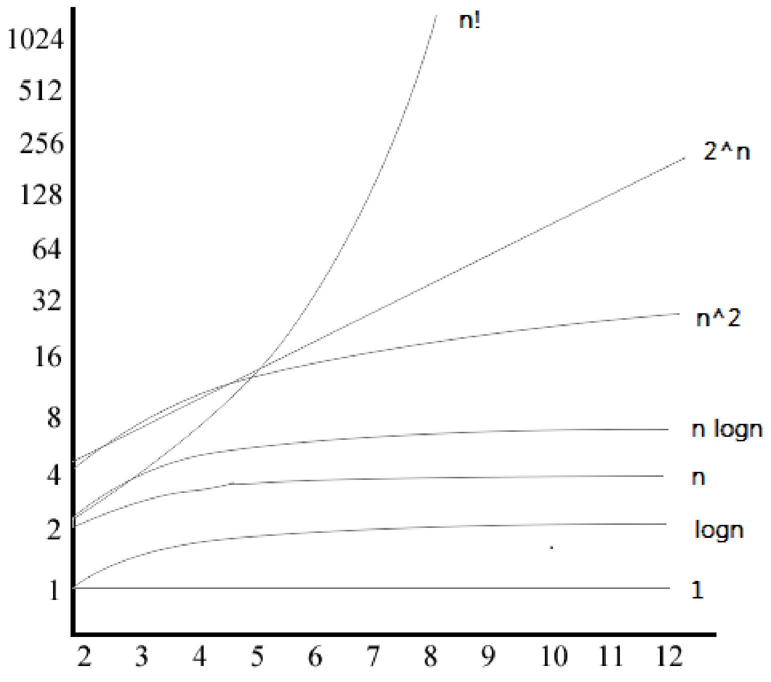


Figure 2.1: Graph for Different Growth rate functions

2.3.4 Order of an Algorithm

According to the definition, “Algorithm A is order $f(n)$ denoted as $O(f(n))$ if constants k and n_0 exist such that A requires no more than $K \cdot f(n)$ time units to solve a problem of size $n \geq n_0$“

The requirement of $n \geq n0$ in the definition of $O(f(n))$ formalizes the notion of sufficiently large problems. In general, many values of k and n can satisfy this definition.

2.3.5 Running Time Calculations

One can evaluate the running time of a program in different ways. In general, there is more than one algorithm anticipated to take similar time to complete, for the reason that we have to program both algorithms and decide which is faster in practical [26]. In the following, we described how to calculate the running time of a simple program.

```
sum-of-list(A,n)            Cost            No.of times
{
    int i, total=0;         1(C_1)              1           /*1*/
    for(i=0;i<n;i++)        2(C_2)              n+1         /*2*/
        total=total+A[i];    2(C_2)              n           /*3*/
    return total;           1(C_1)              1           /*4*/
}
```

As by the rule, the cost for each statement is described in the above program. Lines 1 and 4 computes only one time, so it cost one each. Likewise in line 4 takes 1 unit of cost for the reason that it executes once. In line 2, for loop has 1 unit of cost for initialize and n+1 unit of cost for testing the condition and n unit of cost for increment in total 2n+2.

$$\begin{aligned} T_{sum-of-list} &= 1 + 2(n+1) + 2n + 1 \\ &= 4n + 4 \\ T(n) &= Cn + C' \text{ , where Cn and C' are constants} \end{aligned}$$

Likewise for different types of problems if we calculate the running time for instance,

$$\begin{aligned} T_{sum} = K \quad &\Rightarrow \quad T(n) = O(1) \\ T_{sum-of-list} = C.n + C' \quad &\Rightarrow \quad T(n) = O(n) \\ T_{sum-of-matrix} = a.n^2 + b.n + c \quad &\Rightarrow \quad T(n) = O(n^2) \end{aligned}$$

2.4 Classification of Sorting Algorithms

In general, sorting algorithms can be classified with various parameters. Some of the them are detailed below.

2.4.1 Computational Complexity

The main factor that classifies the sorting algorithm is time or computational complexity. In general, it predicts the performance of each algorithm in which some has worst case performance with large input size of n over other has best case performance under optimal condition. Typically serial sorting algorithms good behavior is $O(n \log n)$ and bad behavior is $O(n^2)$ [26].

2.4.2 Space Complexity

Space complexity of an algorithm is another factor that considers seriously when selecting an algorithm. There are two kinds of memory usage patterns such as "in-place" sorting in which the algorithm needs constant memory size (i.e) $O(1)$ beyond the items being sorted. While the other sorting methods use additional memory space according to their relative input size may called as "out-place" sorting. Due to this the in-place sorting algorithms are slower than the algorithms that uses additional memory. Sometimes escalation can be achieved by considering $O(\log n)$ additional memory in in-place sorting algorithms.

2.4.3 Stability

A stable sorting algorithm preserves the relative order of elements with equal values. For instance a sorting algorithm is stable means whenever there are two elements a[0] and a[1] with the same value and with a[0] show up before a[1] in the unsorted list, a[0] will also show up before a[1] in the sorted list.

2.4.4 Recursive and Non-Recursive

A recursive algorithm means it calls itself with smaller input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller input. Usually the problem can be solved utilizing solutions to smaller variants of the same problem, and the smaller variants reduce to easily solvable instance, then one can use a recursive algorithm to solve that problem [24]. Quick sort and merge sort are examples for recursive algorithms while insertion sort and selection are non-recursive since it does not follow these steps.

2.4.5 Internal Sort Vs External Sort

The sorting algorithms are mainly classified into two categories specifically internal sorting and external sorting. Internal sorting defines the sorting of arrays stored in the random access memory and the external sorting stores the files on disks and tapes for sorting [28].

Chapter 3

Survey of Sorting Algorithms

In this section, we describe different sorting algorithms that have considered in this study. Each algorithm is defined and explained with an example and their efficiencies and other properties are discussed. Finally their advantages and disadvantages are summarized. In this survey, we distinguish two types of algorithms. The former one sorts the elements by comparing with one another, so they named as comparison based algorithm and the latter one does not sort by compare, alternatively each uses their own approach, so they named as non-comparison based algorithm. Moreover, based on [9] most of the algorithms explained in this section. The algorithms considered in this study are as follows.

Comparison Based Algorithms

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

Non-Comparison Based Algorithms

7. Radix Sort
8. Bucket Sort
9. Counting Sort

3.1 Bubble Sort

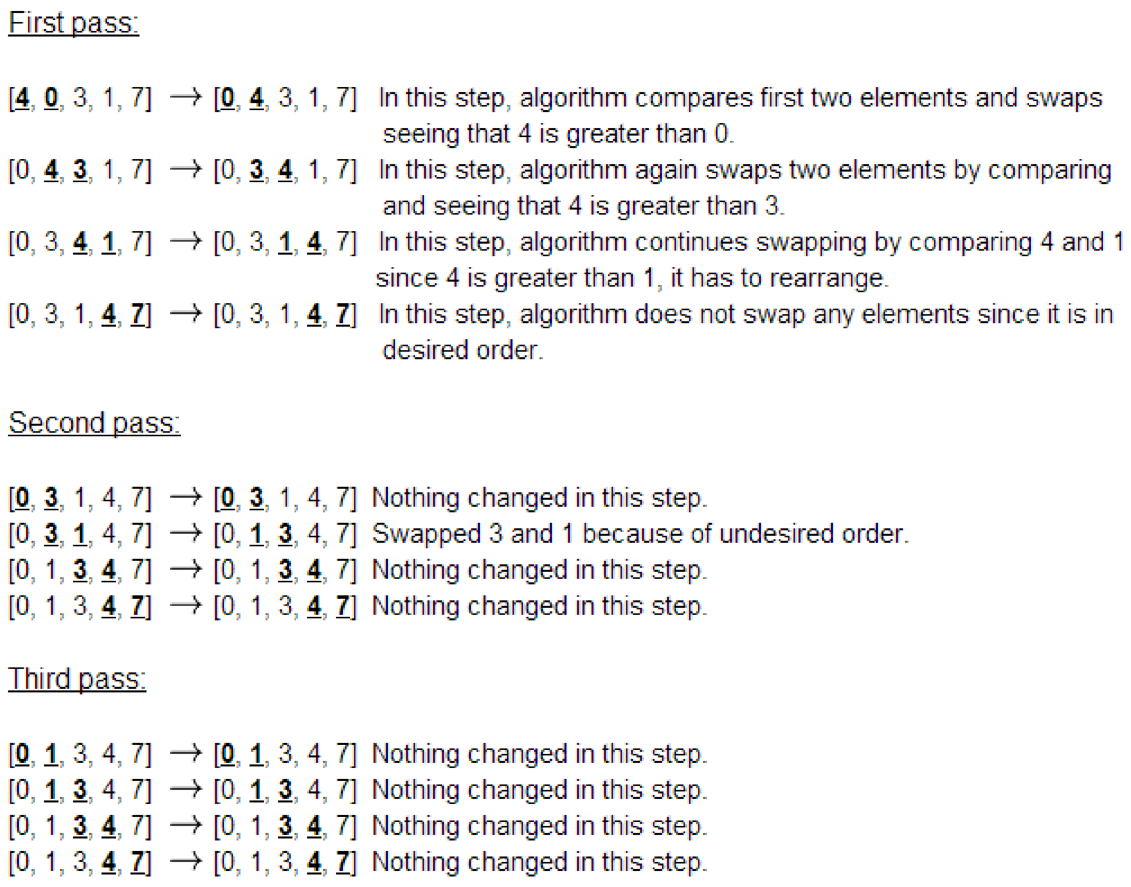
Bubble sort is a simple and the slowest sorting algorithm which works by comparing each element in the list with its neighboring elements and swapping them if they are in undesirable order. The algorithm continues this operation until it makes a pass right through the list without swapping any elements, which shows that the list is sorted. This process makes the algorithm works slower when the size of the input n increased. Because of this reason it considered to be the most inefficient sorting algorithm with large amount of data. The algorithm [12] for bubble sort is as follows.

Algorithm 1 Bubble Sort

```
1: procedure BUBBLE SORT. Records $R_1, .., R_N$ are rearranged in place; after sorting is not known to be in its final position; thus we are indicating that nothing is known at this point.
2:     [Initialize BOUND.] Set BOUND ← N. (BOUND is the highest index for which the record is not known to be in its final position; thus we are indicating that nothing is known at this point.)
3:     [Loop on j.] Set $t \leftarrow 0$. Perform step 4 for $j = 1, 2$...,BOUND-1, and then go to step 5. (If BOUND =1, this means go directly to step 5.)
4:     [Compare/exchange $R_j : R_j + 1$.] If $K_j > K_j + 1$, interchange $R_j \leftrightarrow R_j + 1$ and set $t \leftarrow j$.
5:     [Any exchanges?] If $t = 0$, terminate the algorithm. Otherwise set BOUND ← t and return to step 3.
6: end procedure
```

3.1.1 Explanation

Consider an array of elements A [4, 0, 3, 1, 7] and sorting the array in ascending order using bubble sort. In each step highlighted elements are being compared.



First pass:

[**4**, **0**, 3, 1, 7] → [**0**, **4**, 3, 1, 7] In this step, algorithm compares first two elements and swaps seeing that 4 is greater than 0.
[0, **4**, **3**, 1, 7] → [0, **3**, **4**, 1, 7] In this step, algorithm again swaps two elements by comparing and seeing that 4 is greater than 3.
[0, 3, **4**, **1**, 7] → [0, 3, **1**, **4**, 7] In this step, algorithm continues swapping by comparing 4 and 1 since 4 is greater than 1, it has to rearrange.
[0, 3, 1, **4**, **7**] → [0, 3, 1, **4**, **7**] In this step, algorithm does not swap any elements since it is in desired order.

Second pass:

[**0**, **3**, 1, 4, 7] → [**0**, **3**, 1, 4, 7] Nothing changed in this step.
[0, **3**, **1**, 4, 7] → [0, **1**, **3**, 4, 7] Swapped 3 and 1 because of undesired order.
[0, 1, **3**, **4**, 7] → [0, 1, **3**, **4**, 7] Nothing changed in this step.
[0, 1, 3, **4**, **7**] → [0, 1, 3, **4**, **7**] Nothing changed in this step.

Third pass:

[**0**, **1**, 3, 4, 7] → [**0**, **1**, 3, 4, 7] Nothing changed in this step.
[0, **1**, **3**, 4, 7] → [0, **1**, **3**, 4, 7] Nothing changed in this step.
[0, 1, **3**, **4**, 7] → [0, 1, **3**, **4**, 7] Nothing changed in this step.
[0, 1, 3, **4**, **7**] → [0, 1, 3, **4**, **7**] Nothing changed in this step.

Figure 3.1: Bubble Sort

In the above figure, the algorithm pass right through the list without swapping any elements in the third pass and it stopped at the end of this pass by concluding that the list is sorted.

3.1.2 Performance Analysis

Bubble sort is considered to be the most inefficient algorithm for the reason that it has a worst case and average case complexity of $O(n^2)$, where n is the number of elements to be sorted. Likewise some other simple sorting methods such as insertion sort and selection sort has the same worst case complexity of $O(n^2)$ however the efficiency of bubble sort is comparatively lesser than these algorithms. Hence these computational complexity shows that bubble sort should not be considered over a large amount of data items.

Moreover, there is a better way of implementing the bubble sort described in the modified bubble sort. It suggests a few changes to the standard bubble sort which includes a flag that is set if an exchange is made after an entire pass over the array. If no exchange is made then it certainly show that the array is already in order. Which gives the best case complexity of $O(n)$ if the array is already sorted. Besides another variant of bubble sort is Bidirectional Bubble Sort or Cocktail Sort [12] which sorts the list in both directions each pass through the list. This process slightly reduced the number of comparisons. Moreover Butcher [5] proposed a method whose running time is better than both straight bubble sort and Bidirectional Bubble sort.

3.1.3 Advantages and Disadvantages

Even though Bubble sort considered to be the most inefficient algorithm, it has some advantage over other algorithms such as simplicity and ease of implementation and the ability to identify the list is already sorted if it is efficiently implemented. Moreover bubble sort uses $O(1)$ auxiliary space for sorting.

On the other hand the drawbacks of bubble sort include code inefficient, inappropriate for large volumes of data elements and repetitive problems as well.

3.2 Insertion Sort

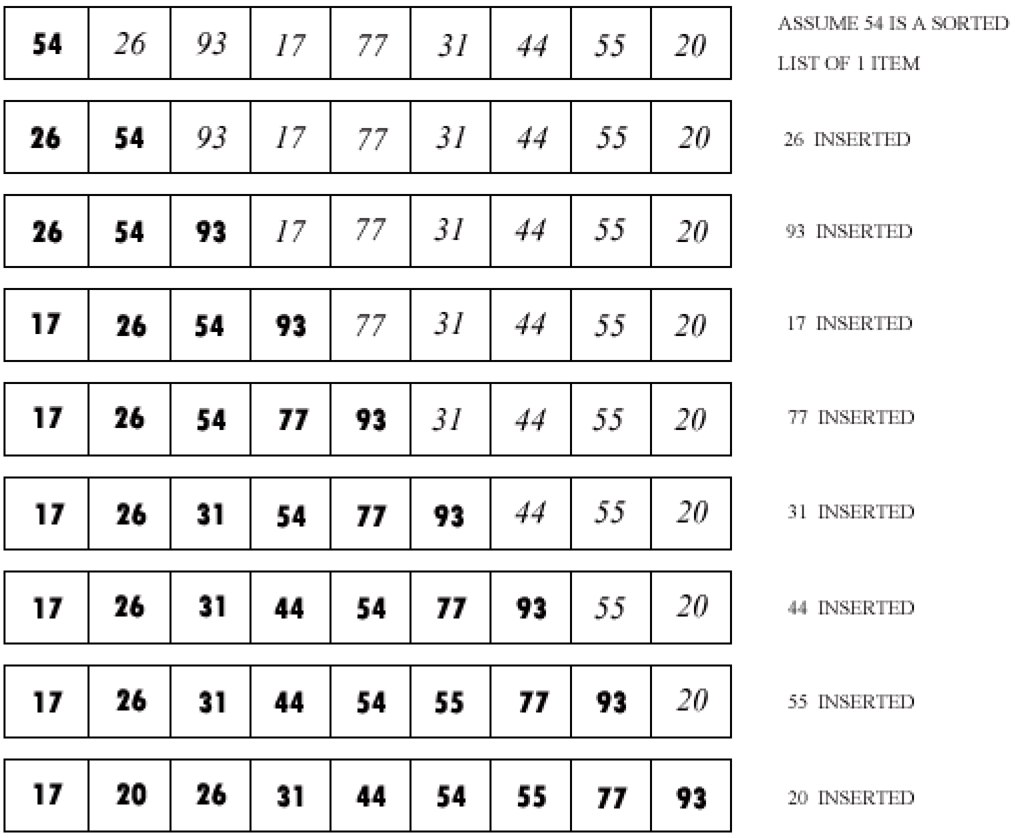
Insertion sort is a simple and efficient sorting algorithm useful for small lists and mostly sorted list. It works by inserting each element into its appropriate position in the final sorted list. For each insertion it takes one element and finds the appropriate position in the sorted list by comparing with neighboring elements and inserts it in that position. This operation is repeated until the list becomes sorted in the desired order. Insertion sort is an in- place algorithm and needed only a constant amount of additional memory space. It becomes more inefficient for the greater size of input data when compared to other algorithms. However in general insertion sort is frequently used as a part of more sophisticated algorithms. The algorithm that explains insertion sort is as follows.

Algorithm 2 Insertion Sort

```
1: procedure INSERTION SORT($list$)
2:     for $i \leftarrow 2, list.length$ do
3:         $key = list[j]$
4:         $i = j - 1$
5:         while $i \leq n$ do
6:             $list[i+1] = list[i]$
7:             $i = i - 1$
8:         end while
9:         $list[i+1] = key$
10:    end for
11: end procedure
```

3.2.1 Explanation

Let us consider an array of elements A [54, 26, 93, 17, 77, 31, 44, 55, 20] and sorting the array into ascending order using Insertion sort. The following figure [3] will demonstrate the process.



54	*26*	*93*	*17*	*77*	*31*	*44*	*55*	*20*	ASSUME 54 IS A SORTED LIST OF 1 ITEM
26	**54**	*93*	*17*	*77*	*31*	*44*	*55*	*20*	26 INSERTED
26	**54**	**93**	*17*	*77*	*31*	*44*	*55*	*20*	93 INSERTED
17	**26**	**54**	**93**	*77*	*31*	*44*	*55*	*20*	17 INSERTED
17	**26**	**54**	**77**	**93**	*31*	*44*	*55*	*20*	77 INSERTED
17	**26**	**31**	**54**	**77**	**93**	*44*	*55*	*20*	31 INSERTED
17	**26**	**31**	**44**	**54**	**77**	**93**	*55*	*20*	44 INSERTED
17	**26**	**31**	**44**	**54**	**55**	**77**	**93**	*20*	55 INSERTED
17	**20**	**26**	**31**	**44**	**54**	**55**	**77**	**93**	20 INSERTED

Figure 3.2: Insertion Sort

In the above figure, the dark numbers represent the sorted elements and the light numbers represent the unsorted elements. First the algorithm takes 26 as a key and compares with its preceding element 54 and swaps since it is greater than key. In second iteration it takes 93 as key and compares with its preceding element since it is less than key there is no swapping and inserts 93 in the same position. In third iteration it takes 17 as a key and compares with its preceding element 93 and swaps since it is greater than key and again compares with its preceding element 54 and swaps since it is greater than key. Likewise it compares all the preceding elements and placed in an appropriate position. These operations continue till the list being sorted. In the last row we can see all the elements are placed in its appropriate positions and the list is sorted.

3.2.2 Performance Analysis

Insertion sort is an iterative algorithm, in which it requires $n - 1$ iterations for n elements. For instance in the above figure it required 8 iterations to complete the sorting of 9 elements. Hence in the worst case scenario, if the elements are in reverse order the computational complexity could be quadratic i.e $O(n^2)$. However, if the array is already sorted, the computational complexity predicted to be linear i.e $O(n)$. This would be the best case running time of insertion sort. The average case running time is also $O(n^2)$ which conceives insertion sort inefficient for large arrays. One of the popular variants of Insertion sort is Shell sort [23]. It is an in-place and unstable comparison sorting method with the best case performance of $O(n \log n)$

3.2.3 Advantages and Disadvantages

Like bubble sort, insertion sort is also inefficient for large arrays, however, it works twice as efficiently as the bubble sort. Moreover it requires numerous elements to scan for identifying the $k + 1th$ element except selection sort it needed to scan all the rest of the element to find the desired one. The feasibility of best case running time of $O(n)$ takes place only when the list is already sorted.

For the reason that it is simple and very efficient for smaller arrays, insertion sort is used in some sophisticated algorithm such as quick sort, merge sort for greater efficiency. Moreover insertion sort uses $O(1)$ auxiliary space for sorting.

3.3 Selection Sort

Selection sort is another simple sorting method that works better than bubble sort and worse than insertion sort. It works by finding the smallest or highest element (most probably desired element) from the unsorted list and swaps with the first element in the sorted list and then finds the next smallest element from the unsorted list and swap with the second element in the sorted list. Consequently sorted elements are increasing at the top of an array and the rest will remain unsorted. The algorithm continues this operation until the list being sorted. Selection sort is also an in-place algorithm since it requires a constant amount of memory space. Like some other simple sorting methods, selection sort also inefficient for large arrays. The algorithm [19] for selection sort is as follows.

Algorithm 3 Selection Sort

Input: A list in the functin of an array of size n with L (left) and R(right) being the selection range, where $1 \leq L, R \leq n$.
Output: The index of array A where the minimum is located.
Remark: If the list contains more than one minimum then it returns the index of the first occurrence.

```
1: procedure SELECTION SORT($list$)
2:     $min = A[L]$
3:     minLoc = L
4:     for $i = L + 1$ to $R$ do
5:         if $min \geq A[i]$ then
6:             $min = A[i]$
7:             $minLoc = i$
8:         end if
9:     end for
10:    $return(minLoc)$
11: end procedure
```

Input: X and Y are two variables.
Output: The value of array X goes to Y and vice versa.
Remark: X and Y should be passed as pointers.

```
1: procedure SWAP
2:     temp = X
3:     X = Y
4:     Y = temp
5: end procedure
```

3.3.1 Explanation

Let us consider an array of elements A [154, 126, 193, 117, 177, 131, 144, 155, 120] and sorting the array in ascending order using selection sort. In this example the algorithm is finding the highest element in each iteration. The following figure [3] will demonstrate the process.

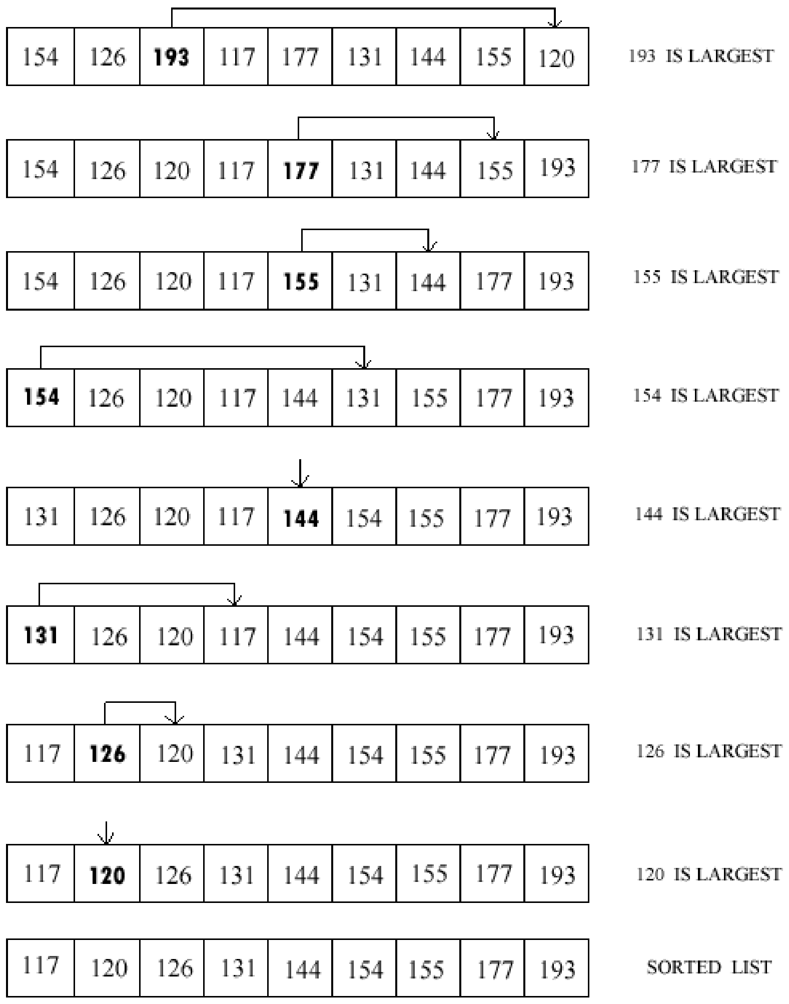


Figure 3.3: Selection Sort

In the above figure the algorithm finds the largest element in the list and swaps it with the appropriate position. The selected highest element is highlighted in dark numbers. In this first iteration it selects 193 and swaps it with the rightmost position in the list containing 120. In the second iteration it finds the next higher element 177 and swaps it with the second position from the right. It will increase the right position by 1 on each iteration the algorithms find and swaps an element. Likewise the algorithm continues this operation until the list being sorted.

3.3.2 Performance Analysis

Since selection sort is an iterative algorithm like other simple sorting methods, it requires $n - 1$ iterations for n elements. In general selecting the highest element needed to compare all n elements in the list at first iteration and swapping them if required. Likewise to select the next highest element it needs to compare $n-1$ elements in the list and so on. Hence it requires $O(n^2)$ comparisons and $n-1$ swaps to sort the list of n elements. Since it has the worst case running time of $O(n^2)$ it is not efficient for large arrays. However, when compared to another quadratic complexity algorithm such as bubble sort, selection sort is much better in efficiency. There are two variants of selection sort are quite popular such as Quadratic Sort[11] and Tree Sort [19][11]. In Quadratic sort for instance, a list A of 16 items are grouped in four sub groups of four items each and then finds the largest item in each subgroup and storing them in a sub list called B. Latter the largest item in the sub list is found by sequential search and swapped with the last position in the list A. This operation continues till the list being sorted. Besides, Tree sort uses the concept of the knockout tournament.

3.3.3 Advantages and Disadvantages

Like other iterative sorting algorithms selection sort is not efficient for large arrays and has better efficiency than bubble sort. Because of its simplicity and ease to implement selection sort would recommend for certain conditions. However, it is advisable to consider insertion sort instead of selection sort of problems with the smaller input size.

3.4 Merge Sort

Merge sort [7] uses the divide and conquer approach to solve a given problem. It works by splitting the unsorted array into n sub array recursively until each sub array has 1 element. In general an array with one element is considered to be sorted. Consequently it merges each sub array to generate a final sorted array. The divide and conquer approach works by dividing the array into two halves such as sub array and follows the same step for each sub array recursively until each sub array has 1 element. Later it combines each sub array into a sorted array until there is only 1 sub array with desired order. This can be also be done non-recursively however, most consider only recursive approaches for the reason that non recursive is not efficient. Merge sort is a stable sort meaning that it preserves the relative order of elements with equal key. The algorithm for merge sort is as follows.

Algorithm 4 Merge Sort

```
1: procedure MERGE SORT($list, start, end$)
2:     if $start < end$ then
3:         $mid = (start + end)/2$
4:         $MERGESORT(list, start, mid)$
5:         $MERGESORT(list, mid + 1, end)$
6:         $MERGE(list, start, mid, end)$
7:     end if
8: end procedure
```

```
procedure MERGE($list, start, mid, end$)
    $n1 = mid - start + 1$
    $n2 = end - mid$
    $letL[1..n1 + 1]andR[1..n2 + 1]benewarrays$
    for $i \leftarrow 1, n1$ do
        $L[i] = A[p + i - 1]$
    end for
    for $i \leftarrow 1, n1$ do
        $R[i] = A[q + j]$
    end for
    $L[n1 + 1] = \infty$
    $R[n2 + 1] = \infty$
    $i = 1$
    $j = 1$
    for $k \leftarrow p, r$ do
        if $L[i] \geq R[j]$ then
            $A[k] = L[i]$
            $i = i + 1$
        else
            $A[k] = R[j]$
            $j = j + 1$
        end if
    end for
end procedure
```

3.4.1 Explanation

Let us consider an array of elements A [54, 26, 93, 17, 77, 31, 44, 55] and sorting the array into ascending order using merge sort. we use divide and conquer approach with recursive method. The following figure will demonstrate the process.

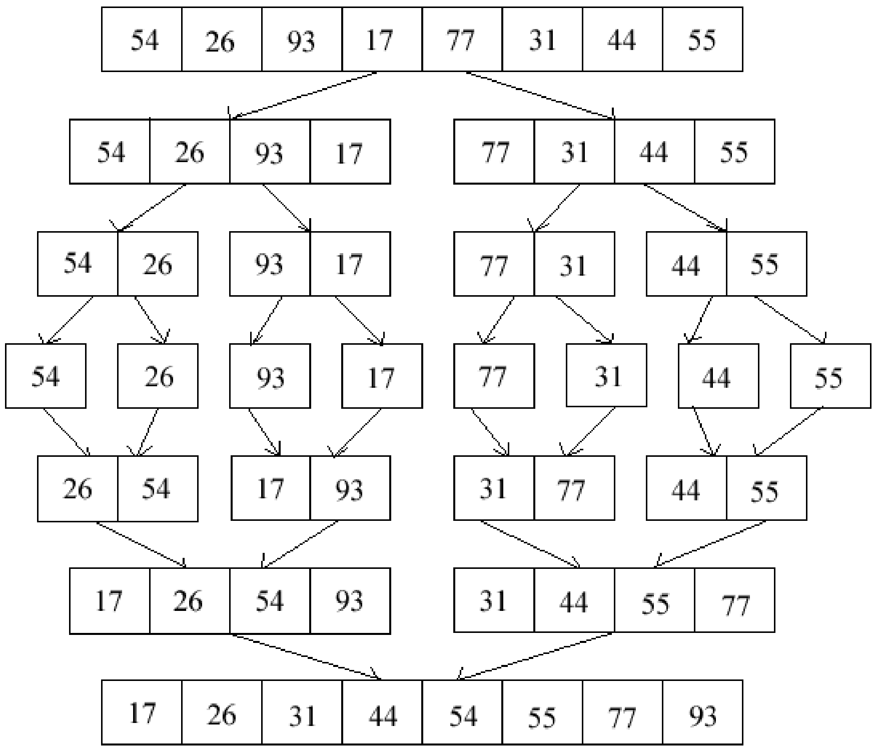


Figure 3.4: Merge Sort

In the above figure the algorithm divides the unsorted array into two equal sub arrays and repeats this operation until each sub array has 1 element. For instance the array [54, 26, 93, 17, 77, 31, 44, 55] divides into eight sub arrays with 1 element each (i.e) [54], [26], [93], [17], [77], [31], [44], [55]. Now each sub array considered to be sorted and the merging process proceeds. The merge operation scans the two sub array and finds the smallest element and thus removes the smallest element and placed in the final sorted sub array. This operation is repeated until there is only one sub array left. This sub array considered to be sorted. For instance the merge operation takes two sub arrays [54] and [26] , it scans these two arrays and finds that 26 is smallest element, so it placed 26 in the first position on the merged sub list and 54 in the second position. This step continues until only one sub array which is sorted (i.e) [17, 26, 31, 44, 54, 55, 77, 93].

3.4.2 Performance Analysis

Merge sort is an efficient sorting method when compared to simple sorting methods and also some sophisticated methods such as heap sort and quick sort. The worst case and average case running time complexity of merge sort is $O(n \log n)$. However it considered not efficient for the reason that it requires twice the number of additional memory for the second array than any other sophisticated algorithms. Merge sort use a separate array to store the entire sub array along with the main array. Generally it is an external sorting algorithm which needs $O(n)$ additional memory space for n elements. This fact makes merge sort inefficient for the application that run on machines with limited memory resource. Therefore, merge sort recommends for large data sets with external memory.

3.4.3 Advantages and Disadvantages

One disadvantage of merge sort is it requires twice as much additional memory than other sophisticated sorting algorithms and likewise it is not recommended for smaller arrays for the reason that it works recursively and it required $O(n)$ auxiliary space for sorting. In addition, it is difficult to implement the merge operation.

However merge sort is considered for the inputs with their data elements stored in a linked list, because merging does not require random access to the list elements. Even though it does not know the length of the list in the beginning in some cases, it requires a single scan through the list to figure it out.

3.5 Quick Sort

Quick sort [13] is the fastest general purpose internal sorting algorithm on the average among other sophisticated algorithms. Unlike merge sort it does not require any additional memory space for sorting an array. For the reason that it is widely used in most real time application with large data sets. Quick sort uses divide and conquer approach for solving problems. Quick sort is quite similar to merge sort. It works by selecting elements from unsorted array named as a pivot and split the array into two parts called sub arrays and reconstruct the former part with the elements smaller than the pivot and the latter with elements larger than the pivot. This operation is called as partitioning. The algorithm repeats this operation recursively for both the sub arrays.

In general, the leftmost or the rightmost element is selected as a pivot. Selecting the left most and right most element as pivot was practiced in the early version of quick sort and this causes the worst case behavior, if the array is already sorted. Later it was solved by various practices such as selecting a random pivot and taking the median of first, middle and last elements. Quick sort is an in-place algorithm and it works very well, even in a virtual memory environment. The algorithm for quick sort is as follows.

Algorithm 5 Quick Sort

```
1: procedure QUICK SORT($list, start, end$)
2:     if $start<end$ then
3:         $index = PARTITION(list, start, end)$
4:         $QUICKSORT(list, start, index - 1)$
5:         $QUICKSORT(list, index - 1, end)$
6:     end if
7: end procedure
```

```
procedure PARTITION($list, start, end$)
    $pivot = list[end]$
    $key = start$
    for $i = start, end - 1$ do
        if $list[i] \leq pivot$ then
            $swap(list[i], list[key])$
            $key + +$
        end if
    end for
    $swap(list[key], list[end])$
    return $key$
end procedure
```

3.5.1 Explanation

Let us consider an array of elements A [3, 7, 8, 5, 2, 1, 9, 5, 4] and sorting the array into ascending order using quick sort. we use divide and conquer approach with recursive method. The following figure will demonstrate the process.

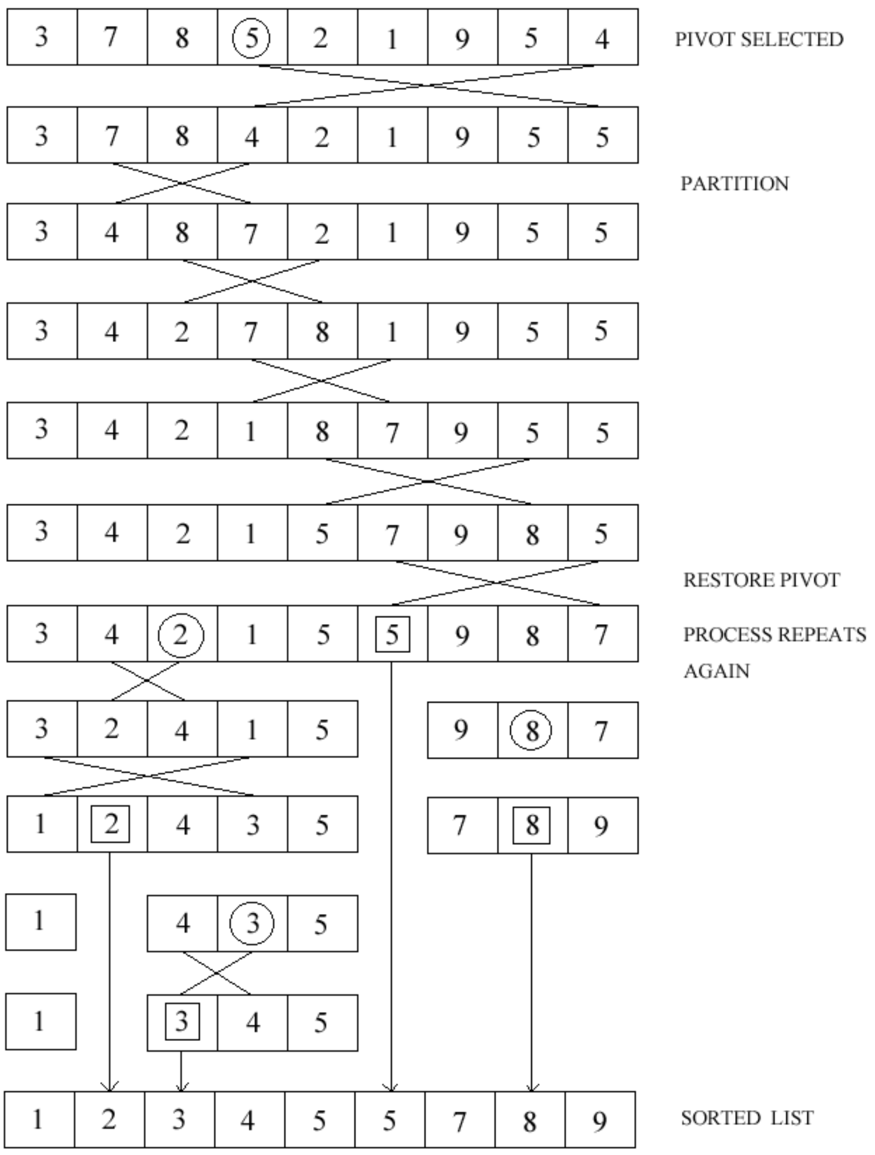


Figure 3.5: Quick Sort

In the above figure, the algorithm first selects the random element as a pivot and hides the pivot in the next meanwhile it reconstructs the array with elements smaller than pivot in the left sub array, likewise elements larger than pivot in the right sub array. The algorithm repeats this operation until the array being sorted. All the pivot elements are highlighted in circles, for instance, in the above figure the first pivot selected was 5, So it moves all the elements higher than 5 such as 9, 8, 7 to the right sub array, likewise smaller elements such as 3, 4, 2, 1, 5 to the left sub array. In general pivot's task is to facilitate with splitting the array and in the final sorted array pivot placed in the position called split point. This operation is repeated with the left sub array [3, 4, 2, 1, 5] and right sub array [9, 8, 7] recursively until the list being sorted (i.e) [1, 2, 3, 4, 5, 5, 7, 8, 9]. Moreover, it preserves the relative order of elements with equal keys in the above figure, for the reason that it is considered as a stable sort.

3.5.2 Performance Analysis

Quick sort is the fastest sort on the average running time complexity of $O(n \log n)$ when compared to other sophisticated algorithms. Usually, selecting the leftmost or rightmost element as a pivot causes the worst case running time of $O(n^2)$ when the array is already sorted. Likewise, it is not efficient if all the input elements are equal, the algorithm will take quadratic time $O(n^2)$ to sort an array of equal elements. However, these worst case scenarios are infrequent. There are more advanced version of quick sort are evolved with a solution to selecting pivot. Qsort [6] is one of the variants of quick sort, which is faster and more robust than the standard method. Quick sort is the better option if speed is greatly significant and also for large data sets. It is quite complicated to sort an array of smaller size, so we can implement a quick sort often with insertion sort to sort smaller arrays.

3.5.3 Advantages and Disadvantages

One of the great advantages of quick sort is that it is fast and efficient for large data sets. However it is not efficient if the array elements are already sorted and also each element in the array are equal. This gives the worst case time complexity of $O(n^2)$. Moreover it's not efficient to sort real objects. Quick sort might be space expensive for large data sets due to the fact that it uses $O(\log n)$ auxiliary space for recursive function calls. Moreover quick sort carry out sequential traverse through array elements which results in good locality of reference and cache behavior for arrays [15].

3.6 Heap Sort

Heap Sort [27] is based on the heap data structure and in-place sorting algorithm. It is quite slower than merge sort in real application even though it has the same theoretical complexity. Unlike merge sort and quick sort it does not work recursively. In general, heap is a specialized tree based data structure that satisfies the heap property, mostly we use binary tree. The tree structure is well balanced, space efficient and fast. Heap sort works by building a heap from the input array and then removing the maximum element from the heap and placing it at the end of the final sorted array i.e $n - 1th$ position. Every time when it removes the maximum element from the heap it restores the heap property until the heap is empty. Thus it removes the second largest element from a heap and puts it on the $n - 2th$ position and so on. The algorithm repeats this operation until the array being sorted. Heap sort does not preserve the relative order of elements with equal keys; hence it is not a stable sort. The algorithm for Heap sort is as follows.

Algorithm 6 Heap Sort

```
1: procedure HEAP SORT($list$)
2:     $BUILD - MAX - HEAP(list)$
3:     for $i = list.lengthdownto2$ do
4:         $swap(list[1], list[i])$
5:         $list.heapsize = list.heapsize - 1$
6:         $MAX - HEAPIFY(list, i)$
7:     end for
8: end procedure
```

```
procedure BUILD-MAX-HEAP($list$)
    $list.heapsize = length(list)$
    for $i = [length(list)/2]$ do
        $MAX - HEAPIFY(list, i)$
    end for
end procedure
```

```
procedure MAX-HEAPIFY($list, i$)
    $l = LEFT(i)$
    $r = RIGHT(i)$
    if $l \leq list.heapsizeandlist[l] > list[i]$ then
        largest=l
    else
        $largest = i$
    end if
    if $r \leq list.heapsizeandlist[r] > list[largest]$ then
        largest = r
    end if
    if $largest \neq i$ then
        $swap(list[i], list[largest])$
        $MAX - HEAPIFY(list, largest)$
    end if
end procedure
```

3.6.1 Explanation

Let us consider an array of elements A [16, 14, 10, 8, 7, 9, 3, 2 4, 1] and sorting the array into ascending order using heap sort. The following figure [1] will demonstrate the process.

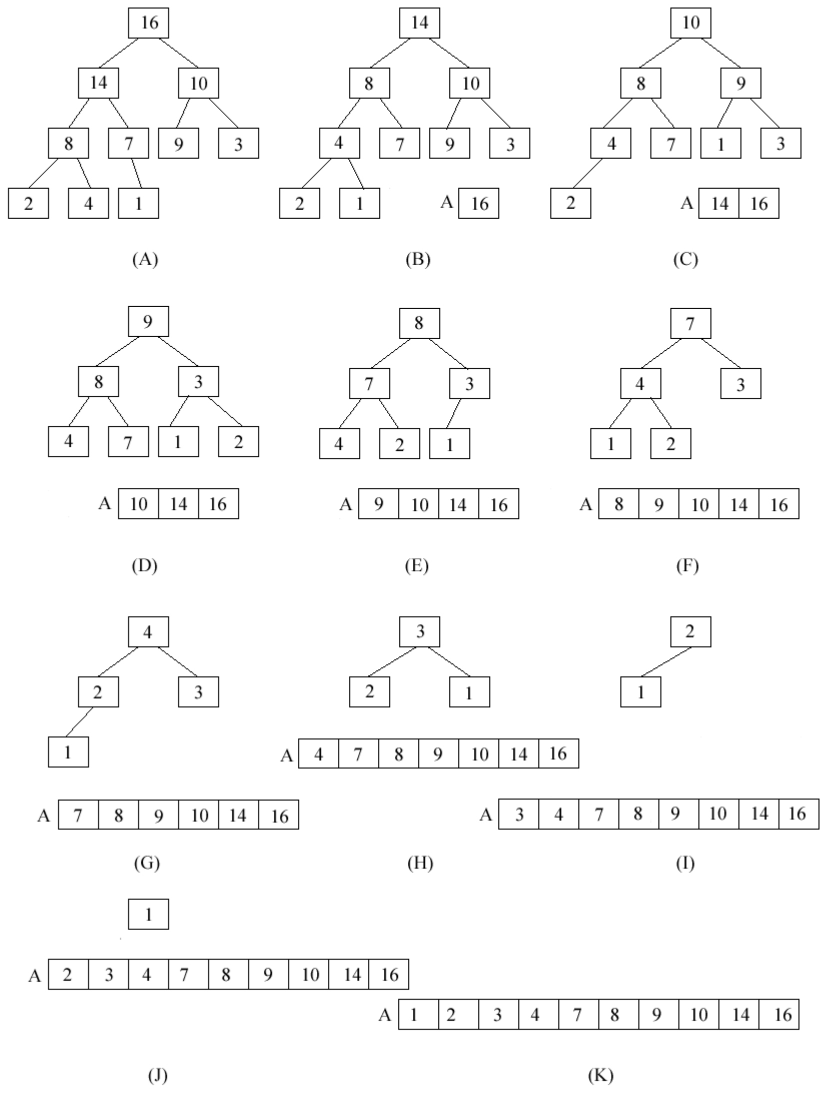


Figure 3.6: Heap Sort

In the above figure, the heap structure is built from the given array of elements [16, 14, 10, 8, 7, 9, 3, 2 4, 1]. The algorithm removes the maximum element from the tree and placed in the rightmost position in the final sorted array. We can see each time when the maximum element is removed the tree restores its heap property. The heap property states that every node should has the higher value than its child node. For instance, in the above figure the top node has 16 maximum than its child node 14 and 10. Likewise 14 is maximum than its child 8 and 10. This process repeats every time when the maximum element removes from the tree. Furthermore the second maximum element which is 14 removed and placed in the n-2th position in the final sorted array. The algorithm repeats this operation until the array being sorted. The final sorted array is [1, 2, 3, 4, 7, 8, 9, 10, 14, 16].

3.6.2 Performance Analysis

The worst case and average case running time complexity of heap sort is $O(n \log n)$. When compared to other sophisticated sorting algorithms with same computational complexity heap sort performs quite slowly in real time application. Even though it performs less than another algorithm, it is usually considered for problems with large data sets, for the reason that it does not work recursively. Moreover heap sort needed only a constant amount of additional memory space for arrays at all.

There are many variants of heap sort were advanced, most of the methods are trying to minimize the number of comparisons and the running time. The Bottom-Up-Heap Sort [25] beats the quick sort on average, if n $\geq$ 400 and the worst case number of comparisons can be bounded by $1.5nlogn + O(n)$. Besides Min-Max Heaps [17] presented the double ended priority queues and this could achieve linear time.

3.6.3 Advantages and Disadvantages

One of the disadvantages of heap sort is that it works slower than other sorting methods with same computational complexity. Moreover, it is not efficient for parallelization. However, heap sort is considered often for large data sets for the reason that it does not work recursively all the time. Sorting data stored in linked list data structure, heap sort is not recommended due to the fact that it's difficult to convert the linked list to heap structure.

3.7 Radix Sort

Radix sort [21] is a linear sorting algorithm and works without comparing any elements unlike other sorting methods such as insertion sort and quick sort. Radix sort works by sorting data elements with keys. Keys are usually represented in integers mostly binary digits and sometimes it considers an alphabet as keys for strings. Radix sort works by sorting each digit on the input element and for each of the digits in that element. In general, it might start with least significant digit and then followed by next significant digit till the most significant digit. This process somewhat considered to be unreasonable most of the time. Radix sort is a stable sort for the reason that it preserves the relative order of element with equal keys.

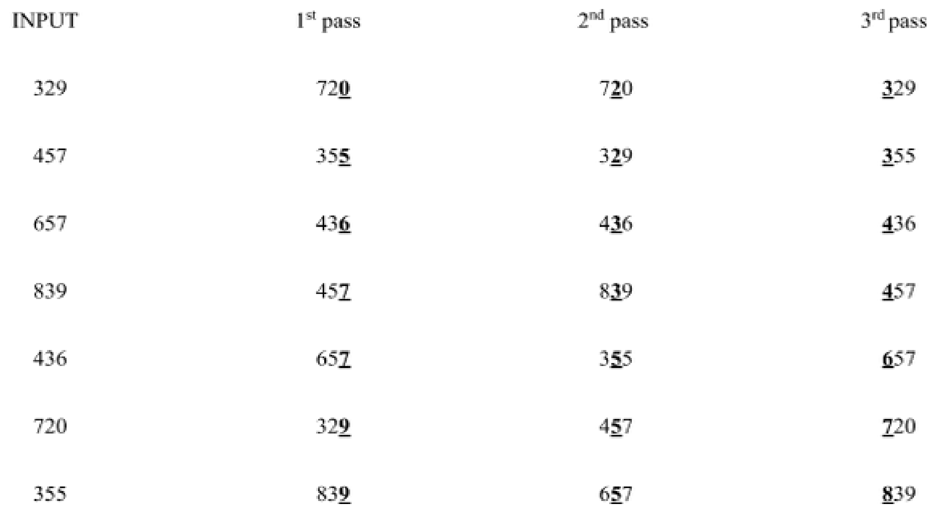
There are two classifications of radix sort such as least significant digit (LSD) and most significant digit (MSD). The Least significant digit method works by processing the integer representation starting from the least digit and shift in order to obtain the most significant digit. Likewise the Most significant digit works the opposite way. The algorithm for radix sort is as follows.

Algorithm 7 Radix Sort

```
1: procedure RADIX SORT($list$)
2:     for $i = 1$ To $d$ do
3:         Use a stable sort to sort array A on digit i
4:     end for
5: end procedure
```

3.7.1 Explanation

Let us consider an array of elements A [329, 457, 657, 839, 436, 720, 355] and sorting the array into ascending order using radix sort. The following figure [9] will demonstrate the process.



INPUT	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 3.7: Radix Sort

In the above figure, the algorithm first takes the input elements [329, 457, 657, 839, 436, 720, 355] and proceeds the first pass with least significant digit as a key. Major events are highlighted in the above figure. Likewise the algorithm repeats the process with next significant digit continuously to the most significant digit until the array of elements being sorted. For instance, in the above figure it sorts the last digit in the first pass and the middle digit in the second pass and the first digit in the third pass. As a result the array is sorted [329, 355, 436, 457, 657, 720, 839]. Moreover it preserves the relative order of an element with equal keys, for instance, in the above figure it preserves 7 at first pass, 2 and 3 in second pass and 4 in the third pass. For this reason it's called as stable sort.

3.7.2 Performance Analysis

The efficiency of radix sort is difficult to describe when compared to other sophisticated algorithms. In theory, the average run time complexity of radix sort is $O(d{\cdot}n)$. The practical efficiency depends on the value of d, where d is the number of digits in each array value. If the number of digits is constant for all the values in the array, then the performance of radix sort would be more efficient than other sophisticated algorithms. However, with different number of digits in array values, the algorithm required $O(\log n)$ computational time, which is almost identical to other sophisticated algorithms such as quick sort and merge sort. Therefore radix sort would become inefficient for applications with distinct array values.

It is known that breaking the $O(n \log n)$ computational complexity is difficult for any key comparison method, however the radix sort has this potential to sort N keys in $O(N)$ operations. Because of this, many variants are advanced such as A Fast Radix Sort [10] which is faster than any quick sort variants and Forward Radix Sort [4] which combines the advantages of the conventional left-to-right and right-to-left radix sort, thus it will work very well in practice.

3.7.3 Advantages and Disadvantages

One of the advantages of radix sort is that its efficiency does not reflect upon with the type and size of input data being sorted. Likewise, when compared to the other integer sorting algorithm, radix sort can handle larger keys more efficiently On the other hand it is less flexible and complex to program for a wide variety of functionality and requirements. Moreover radix sort takes more memory space than other sophisticated algorithm, hence a problem with the memory storage space is often considered as primary concern then radix sort won't be a good choice.

3.8 Counting Sort

Counting sort is an integer sorting algorithm with linear running time complexity. Like radix sort, counting sort also works based on the keys with range between 0 and n, where n is size of the input array. Counting sort assumes that the input consists of an integer in a small range which makes it faster. It works by counting the number of occurrences of each element in the input usually called as keys and store this information into another array called C. Finally, it determined the position of each key value in the final sorted array by using some arithmetic operations on the data in array C. It is not a comparison sort and it preserves the relative order of an element with equal keys. Counting sort is often used as a subroutine in radix sort. The algorithm for counting sort is as follows.

3.8.1 Explanation

Let us consider an array of elements A [3, 6, 4, 1, 3, 4, 1, 4] and sorting the array into ascending order using counting sort. The following figure [2] will demonstrate the process.

Algorithm 8 Counting Sort

```
1: procedure COUNTING SORT($list, C, k$)
2:     $letC[0..k]$ be a new array
3:     for $i \leftarrow 0, k$ do
4:         $C[i] \leftarrow 0$
5:     end for
6:     for $j \leftarrow 1, list.length$ do
7:         $C[list[j]] \leftarrow C[list[j]] + 1$
8:     end for
9:     for $i \leftarrow 1, k$ do
10:        $C[i] \leftarrow C[i] + C[i-1]$
11:    end for
12:    for $j \leftarrow list.length$ downto 1 do
13:        $B[C[A[j]]] \leftarrow A[j]$
14:        $C[A[j]] \leftarrow C[A[J]] - 1$
15:    end for
16: end procedure
```

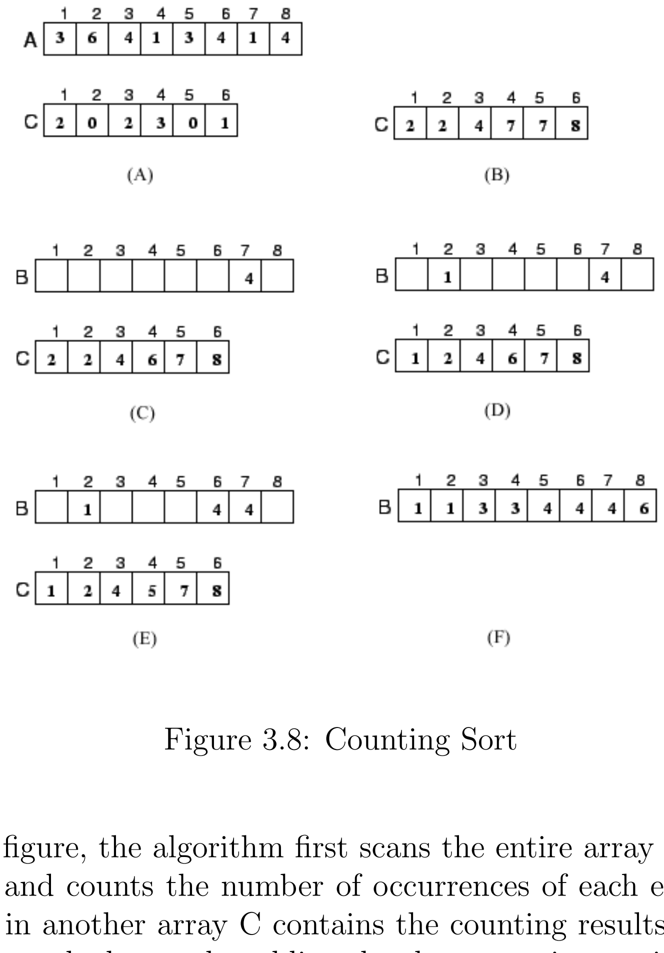


Figure 3.8: Counting Sort

In the above figure, the algorithm first scans the entire array of elements [3, 6, 4, 1, 3, 4, 1, 4] and counts the number of occurrences of each element and stores this information in another array C contains the counting results [2, 0, 2, 3, 0,1]. Later it computes each element by adding the element to its previous element in the array C. As a Result the array C now has [2, 2, 4, 7, 7, 8]. Finally it determines the position of each element in the final sorted array and place them one by one and also decrease the count of relevant values in the array C by 1. The algorithm repeats this operation until the array being sorted. The final sorted array has the elements [1, 1, 3, 3, 4, 4, 4, 6].

3.8.2 Performance Analysis

For the reason that counting sort is straightforward algorithm, it is simple to analyze its computational complexity. The worst case and average case performance of counting sort is $O(n+k)$ In order to obtain maximum efficiency k must not be higher than n. when compared to other linear sorting algorithm counting sort is easy to implement it does not require any special data structure to store its elements.

3.8.3 Advantages and Disadvantages

One of the advantages of counting sort is that it uses key values as indexes into an array, for this reason it can be used into another sorting algorithm as a subroutine, hence it should maintain the relative order of element with equal keys. On the other hand, it is not suitable for large data sets and also for strings.

3.9 Bucket Sort

Bucket sort is a distribution sorting algorithm and not a comparison sort. Like counting sort it assumes some factors such as the input is extracted from a uniform distribution which makes it work faster. The technique behind this algorithm is to split the given array into a number of sub arrays called buckets. Each bucket is then sorted independently either using various sorting algorithms or recursively implement the same technique. It is a linear sorting algorithm and also preserves the relative order of an element with equal keys. The algorithm for bucket sort is as follows.

Algorithm 9 Bucket Sort

```
procedure BUCKET SORT($list$)
    let B[0...n-1] be a new array
    $n \leftarrow list.length$
    for $i \leftarrow 0, n-1$ do
        make B[i] an empty list
    end for
    for $i \leftarrow 1, list.length$ do
        insert list[i] into B[list.length[i]]
    end for
    for $i \leftarrow 0, n-1$ do
        sort list B[i] with insertion sort
    end for
    concatenate the lists B[0], B[1],....,B[n-1] together in order
end procedure
```

3.9.1 Explanation

Let us consider an array of elements A [77, 16, 38, 25, 71, 93, 22, 15, 24, 69] and sorting the array into ascending order using bucket sort. The following figure will demonstrate the process.

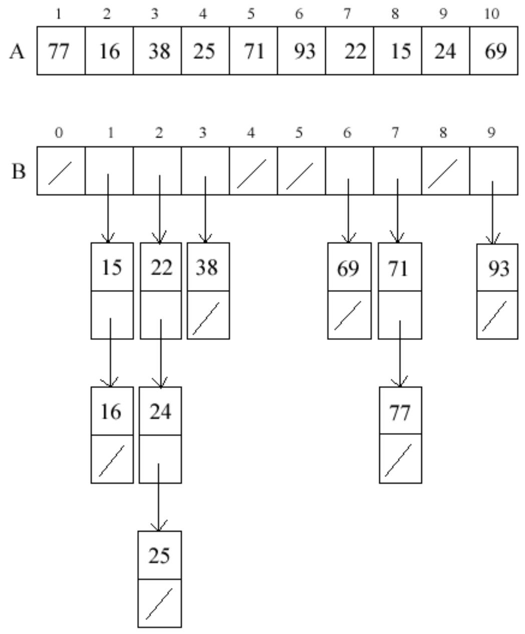


Figure 3.9: Bucket Sort

In the above figure, the algorithm first takes the input element [77, 16, 38, 25, 71, 93, 22, 15, 24, 69] and then created the buckets by splitting the array. In each bucket of i holds values in the interval [i/10, (i+1)/10]. For instance, in the above figure the bucket 1 holds 15, 16 and bucket 2 holds 22, 24, 25. Finally all buckets are concatenated in the sorted array.

3.9.2 Performance Analysis

Bucket sort has a worst case computational complexity of $O(n^2)$ and an average case complexity of $O(n + k)$. It uses an insertion sort to insert elements into a bucket which increase the running time in a worst case scenario. It needed $O(n \cdot k)$ memory space for storing arrays at worst case scenario.

3.9.3 Advantages and Disadvantages

One of the advantages of bucket sort is that it runs in linear time in the average case and it is not suitable for large data sets. On the other hand, to know how many buckets we have to know the maximum value of an element that can be found in the input array then we can set up buckets or else it would be difficult. Another disadvantage is that it needed massive memory to set up enough buckets.

Chapter 4

Measuring the Practical Efficiency of Sorting Algorithms

In general, efficiency of an algorithm primarily relies upon three factors and sometimes it also considered how effectively the algorithm implemented under what circumstances and type of machines used. To be independent, the analysis should be organize unassociated with some of the above mentioned factors such as what machines used, programming style and running environment. However the three primary factors that considered in every analysis is to ensure the efficiency of an algorithm such as overall running time of an algorithm, number of comparison and assignment operations performed and memory space needed for the program and also for the data used during running time. These are the computational resources needed for the algorithm to implement. The necessity of an analysis is to make sure that which algorithm required fewer resources to solve a given problem. In this way we can find out the optimal algorithm for a given problem.

4.1 Quantities of Interest

One of the main objectives of this thesis is to compare the sorting algorithm from a practical perspective. For the reason that there are three factors that interested the whole period of study. They are defined and explained in the following sections.

4.1.1 Overall Running Time

There are two major reasons to calculate the running time of an algorithm, the farmer is to understand how the time grows as a function with respect to its input parameters and the latter is to compare two or more algorithms for the same problem. The overall running time or the time complexity of an algorithm states that the amount of time taken by an algorithm regarding to the amount of input data to the algorithm. Usually the time taken includes the number of memory accesses performed, the number of comparisons between integers, and the number of times some inner loop is executed. Likewise there are other factors not related to algorithms can also affect the running time [14].

4.1.2 Number of Operations Performed

In general, there are two different types of operations are performed in most of the sorting algorithms, such as comparison operation and assignment operation. For instance, to sort an array of integers the algorithm needs to compare each element in that array to find out the appropriate position in the final sorted array, this method is called as comparison operation. It affects the overall running time of an algorithm, due to the fact that if the input size grows, then the number of comparisons will also increase. In assignment operation, each time when the desired position found, the algorithm need to swap the element by using a temporary variable. This will also affect the running time of an algorithm, however when compared to comparison operation the influence of an assignment operation in the running time is comparatively less.

4.1.3 Memory Consumption

The memory required for an algorithm during run time is another important concern while selecting an optimal algorithm for a given problem. When we talk about the memory consumption, it generally means main memory or RAM. Memory or space complexity states that amount of memory taken by an algorithm with regards to the amount of input of an algorithm. Certainly each program needed some amount of memory to store the program itself and also the data required during execution. Moreover, the major consideration is the additional amount of memory utilized by the program, for the reason that some algorithm might need extra memory, according to the size of input and it is not suitable for the machine with minimum resource.

4.2 Software Profiling

Software profiling is the investigation of a program's behavior using information collected during the execution of the program. The aim of this analysis is to determine which part of a program need to improve in order to extend its efficiency. Some of the performance enhancement measures are increasing the overall speed, minimizing the memory usage, determining the need of time consuming functions and calls, avoiding the unnecessary computation, avoiding re-computation by storing the results for future use. There are various techniques used by profilers based on the different quantities of interest such as event-based, statistical, instrumented and simulation methods.

There are many varieties of commercial profiler tools advanced based on the analysis requirement. some famous profilers used for C++ are AQTime, AMD CodeAnalyst, Proffy, CodeTune and IBM Rational Purify. Each profilers has its own advantages, disadvantages and efficiency problems. Unfortunately there is no one suitable profiler that supports all the requirements of the planned experiment. Therefore, we have designed and developed our own program with simplicity and ease of implementation.

4.3 Design and Implementation

In the previous sections of this chapter, we have discussed about software profiling and its usage, along with the quantities of interest that considered for this study. Based on the above analysis, we have designed a C++ program with the ability to collect desired data during execution. The design includes the various sorting algorithms along with the profiler to collect the desired data while executing the program. The program is capable of creating 300000 random numbers of integers and strings. The size of the array elements varied from 1000 to 300000 and change automatically once each algorithm completed its execution with certain array size. The $rand()$ function was used to generate random numbers and $srand()$ function was used as a seed to initialize the random number generator every time the program being executed. In general, the maximum value returned by the $rand()$ function is 32767 however this program has the ability to generate 200000 unique values. The data collected during the experiment was the running time of an algorithm and the number of comparison and assignment operations performed. All the collected data's were written in a file after each algorithm completed its execution for further evaluation. In order to measure the running time of each sorting algorithm the $clock()$ function was used and divided by CLK_TCK to get the time in seconds.

The experiment was repeated for various test case scenarios such as different types of input data for instance arrays of numbers and strings of different sizes. Each string element in the array was the alphanumeric values that are randomly generated at the beginning of the program execution. Desired data were collected for each experiment and written in a file for further evaluation. Data gathered during the execution of the program was later used for comparing various sorting algorithms and their efficiency against various experiments. In this C++ program, instead of passing arrays to each algorithm, we have declared arrays globally and then accessed from each algorithm.

All the experiments were conducted on the model machine described below as our test bed. Test bed: Intel Core i3 CPU M350 @ 2.27 GHz, 2GB RAM, 360 GB HDD, Windows 7 Home Premium Service Pack 1, 32-bit Operating System, Microsoft visual studio 8.

4.3.1 Algorithms and their Implementation

In this study, most of the sorting algorithms were implemented based on the standard algorithms explained in the literature and articles. However, some algorithms were deviated from standard methods to improve the overall efficiency during the execution. A C++ program for each sorting algorithm is explained in the Appendix to illustrate the practical implementation of each algorithm.

The Appendix A illustrates the bubble sort; the code clearly shows its simplicity and inefficiency. This was improved in the modified bubble sort explained in Appendix B. In the improved bubble sort the second loop does not run for $n-1$ values for that reason that at any given stage the array has both the sorted and unsorted values, so to skip the sorted values the loop only run for $n-k-1$ values. Moreover a flag is used to break from unnecessary loop run once the array being sorted. Initially it is set to 0 and then change to 1 in every swap. Selection sort is another inefficient and simple sort similar to bubble sort explained in the Appendix C. For each iteration, the outer for loop assigns the elements as *imin* and the inner for loop finds the actual minimum value by passing through the entire array against the *imin* value. Later it will swap the positions.

The Appendix D illustrates the insertion sort and the Appendix E illustrates the quick sort. The efficiency of the quick sort in Appendix E is comparatively low when compared to other quick sort implementations. However, this is the simplest and the standard way of implementing the quick sort in the beginning, later some figured out the efficient way of selecting the pivot element. One of the flaws in this implementation is that pivot is selected from the end of the array each time. In Appendix F, the improved version of quick sort explained, in which the recursion and partition were done in same function instead of two functions in the inefficient implementation discussed in Appendix E. Moreover, the pivot is selected from the mid of the array each time. This makes the algorithm more efficient in all cases when compared with other implementations. However, there are more ways to implement quick sort more efficient.

The Appendix G illustrates the merge sort, which use recursion and divide and conquer method. The merge sort function continuously divide the array into sub arrays till the condition low <high fails. Once it reaches that point it moves to merge function, there it merges the array elements using a temporary array. This is one of the drawbacks of merge sort that it required an extra array to sort, this demands additional memory usage. The Appendix H illustrates the code implementation of heap sort. The method used here is bottom up, implies that building a heap from the bottom up by shifting downward to organize heap property.

In Appendix I, the code illustrates the bucket sort implementation. In which the bucket is created according to the number of unique elements or range of values in the given array, which mentioned in the variable bsize. Once the bucket is created the array elements are placed into their associated bucket, the second for loop doing this operation. Later the elements in buckets have been placed in the appropriate position of the final sorted array which is done in third for loop. The Appendix J and K illustrate the code implementation of radix sort and counting sort. In the beginning, it gets the maximum value in the array and then used two pass method, the first pass determines the number of keys will fit into each bucket and the second pass places each key into the appropriate bucket.

4.3.2 Different Test Case Scenarios

In this study, various experiments were conducted in the above mentioned model machine to observe the behavior of each algorithm. These test cases are

Test Case 1: In this experiment the randomly generated array values were integers and the data collected during the experiment was the running time of each sorting algorithm. The array size varied from 1000 to 300000 for each of the sorting algorithm.

Test Case 2: In this experiment the randomly generated array values were strings of size three and the data collected during the experiment was the running time of each sorting algorithm. The array size varied from 1000 to 300000 for each of the sorting algorithm.

Test Case 3: In this experiment the randomly generated array values were strings of size five and the data collected during the experiment was the running time of each sorting algorithm. The array size varied from 1000 to 300000 for each of the sorting algorithm.

Test Case 4: In this experiment the randomly generated array values were integers and the data collected during the experiment was the number of comparison and assignment operations performed for each sorting algorithm. The array size varied from 1000 to 300000 for each of the sorting algorithm.

Test Case 5: In this experiment the randomly generated array values were integers and the data collected during the experiment was the amount of memory space used for each sorting algorithm. The array size varied from 1000 to 300000 for each of the sorting algorithm.

The experiments was repeated multiple times for each test case and the data mentioned in this study are averages of 10 samples. The Results of all the test cases are discussed in the following chapter with the help of data and projected graphs.

In addition, the source code for these experiments can be downloaded from
https://github.com/kumar2013/Sorting

Chapter 5

Comparison of Sorting Algorithms

In the previous section, we have discussed the design and implementation of various algorithms considered for this study along with various test case experiments which are conducted for the evaluation of the practical efficiency of sorting algorithms. In this chapter we have discussed the experiment results besides analysis and comparison of various sorting algorithms with the help of experiment results. According to the theoretical study, we have concluded and compared the time complexity of various sorting algorithms in three different cases such as average, best and worst along with the stability and method of working. The table described below represents the variations, where n defines the number of items to be sorted, k defines the range of numbers in the list.

Sorting Method	Best Case	Worst Case	Average Case	Stable	Method
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes	Exchange
Modified Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Exchange
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes	Selection
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Insertion
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	Merge
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	Yes	Partition
Randomized Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	Partition
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	Selection

Table 5.1: Comparison of Comparison Based Sort

Sorting Method	Best Case	Worst Case	Average Case	Stable
Radix Sort	$O(n \cdot \frac{k}{s})$	$O(n \cdot \frac{k}{s})$	$O(n \cdot \frac{k}{s})$	No
Counting Sort	$O(n + 2^k)$	$O(n + 2^k)$	$O(n + 2^k)$	Yes
Bucket Sort	$O(n \cdot k)$	$O(n^2)$	$O(n \cdot k)$	Yes

Table 5.2: Comparison of Non-Comparison Based Sort

5.1 Experiment Results and Analysis

In this section, we have described our experiment results and analyzed the data being gathered during various experiments. Results of these experiments are described and analyzed with the table of data and observed behavior.

5.1.1 Test Case 1

Results of Test case 1 are shown in Table 5.3. It shows the running times of n^2 sorting class algorithms. The size of the array elements are varied from 1000 to 300000.

No.of Data Elements	Bubble Sort	Modified Bubble Sort	Selection Sort	Insertion Sort
1000	0.0129	0.0082	0.0041	0.0022
2000	0.0306	0.0203	0.0091	0.0066
3000	0.0674	0.0455	0.0196	0.0142
5000	0.1766	0.1237	0.053	0.0394
10000	0.7029	0.5046	0.209	0.1559
20000	2.184	2.0253	0.8485	0.6237
30000	6.3735	4.6168	1.9163	1.4009
50000	17.8113	12.8607	5.2546	3.8873
100000	71.5413	51.143	21.0054	15.5116
200000	284.1705	203.669	82.058	61.3766
300000	644.3463	456.8564	182.2555	136.6884

Table 5.3: Running Time (in sec) of n^2 Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 1)

The results of $n \log n$ sorting class algorithms are shown in Table 5.4. The algorithms described in the table are quick sort, merge sort, heap sort and improved quick sort.

The results of non-comparison based sorting algorithms are shown in Table 5.5 . The algorithms described in the table are radix sort, counting sort and bucket sort.

By observing the table of results we have plotted a graph for each algorithm to analyze the practical performance. In the following, each figure describes the practical performance of sorting algorithms for randomly generated n elements. In Figure 5.1, it is observed that modified bubble sort takes less time when compared to normal inefficient bubble sort besides insertion sort performs well when compared to other $O(n^2)$ Sorting algorithms considered for this study. Moreover in Figure 5.2, the standard quick sort performs less when compared to other $nlogn$ sorting methods, however the improved version quick sort 2 outperforms and takes less time when compared with the rest and proved it is efficient. Likewise, in Figure 5.3, the graph shows that bucket sort takes less time when compared to other non-comparison based sorting algorithm. All these results observed from Test Case 1 which described in the previous sections.

No.of Data Elements	Quick Sort	Quick Sort 2	Merge Sort	Heap Sort
1000	0.0006	0.0003	0.0003	0.0003
2000	0.0007	0.0006	0.0008	0.0006
3000	0.0017	0.0009	0.0019	0.0008
5000	0.0022	0.0016	0.0018	0.0016
10000	0.0057	0.0032	0.0035	0.0036
20000	0.1120	0.0067	0.0079	0.0074
30000	0.0211	0.0110	0.0118	0.0111
50000	0.0286	0.0183	0.0198	0.0198
100000	0.0638	0.0377	0.0409	0.0437
200000	0.1392	0.0778	0.0882	0.0931
300000	0.2145	0.1204	0.1319	0.1448

Table 5.4: Running Time (in sec) of $n \log n$ Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 1)

No.of Data Elements	Radix Sort	Bucket Sort	Counting Sort
1000	0.0009	0.0026	0.0026
2000	0.0011	0.0020	0.0030
3000	0.0020	0.0022	0.0025
5000	0.0037	0.0020	0.0028
10000	0.0064	0.0024	0.0027
20000	0.0116	0.0022	0.0039
30000	0.0179	0.0031	0.0046
50000	0.0311	0.0035	0.0059
100000	0.0602	0.0054	0.0094
200000	0.1214	0.0084	0.0165
300000	0.1829	0.0104	0.0237

Table 5.5: Running Time (in sec) of Non-Comparison Based Sorting Algorithm on Random Data Averaged 10 Runs (Test Case 1)

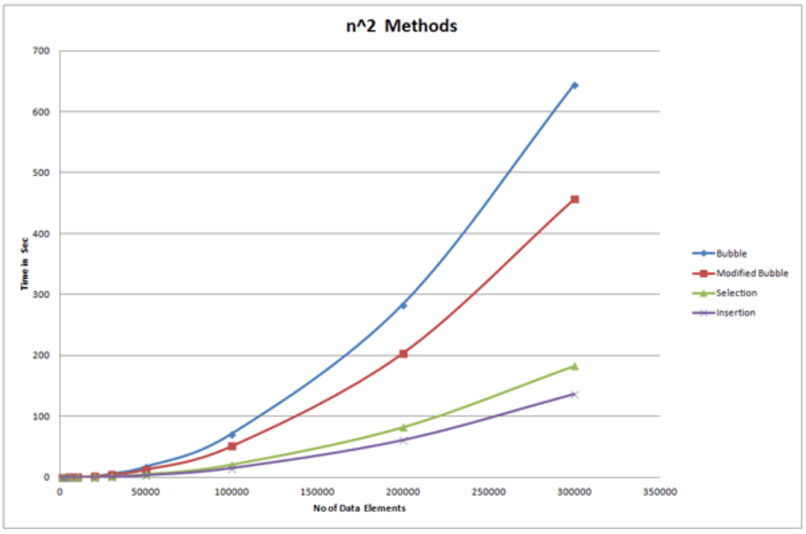


Figure 5.1: n^2 Sorting Class Performance Test Case 1

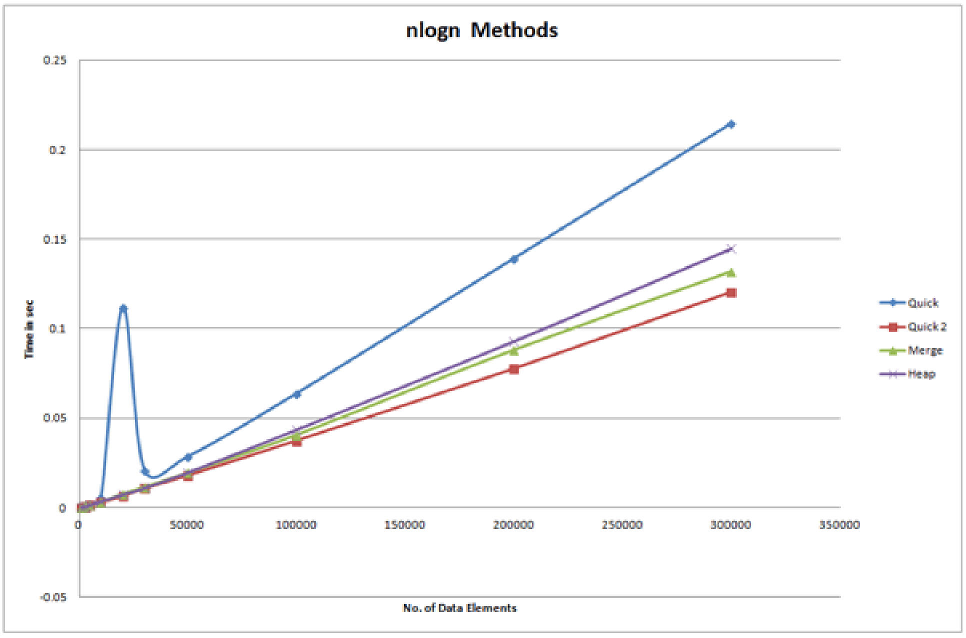


Figure 5.2: $n \log n$ Sorting Class Performance Test Case 1

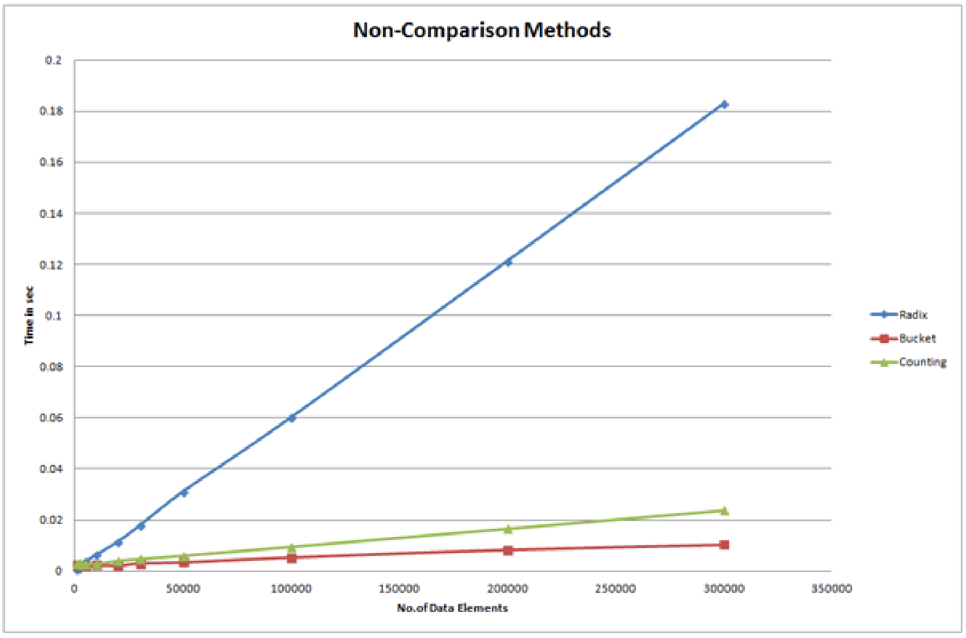


Figure 5.3: Non Comparison Based Sorting Perormance Test Case 1

5.1.2 Test Case 2

Results of Test case 2 are shown in Table 5.6. It shows the running times of sorting class algorithms. The size of the array elements are varied from 1000 to 300000 and the inputs were strings of size three.

No.of Data Elements	Bubble Sort	Modified Bubble Sort	Selection Sort	Insertion Sort
1000	0.1576	0.1172	0.0314	0.0382
2000	0.6328	0.4628	0.1206	0.1688
3000	1.4238	1.0446	0.2744	0.3766
5000	3.9930	2.9084	0.7768	1.0378
10000	16.2952	11.7172	3.0620	4.1084
20000	66.0238	47.4252	12.46	16.53
30000	145.2124	106.1016	27.5474	37.1234
50000	406.1814	296.647	77.4712	102.889
100000	1659.148	1184.134	309.8992	415.6938
200000	6715.638	4782.802	1252.576	1693.898
300000	15111.9	10631.1	2794.828	3699.648

Table 5.6: Running Time (in sec) of n^2 Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 2)

The results of $n \log n$ sorting class algorithms are shown in Table 5.7. The algorithms described in the table are quick sort, merge sort, heap sort and improved quick sort 2.

No.of Data Elements	Quick Sort	Quick Sort 2	Merge Sort	Heap Sort
1000	0.0028	0.0018	0.0084	0.0104
2000	0.004	0.0034	0.0072	0.0076
3000	0.0066	0.0058	0.013	0.018
5000	0.011	0.01	0.21	0.0376
10000	0.0254	0.0214	0.041	0.0768
20000	0.0534	0.0456	0.0884	0.1652
30000	0.0842	0.0692	0.1332	0.2632
50000	0.148	0.121	0.2364	0.454
100000	0.3228	0.257	0.5086	0.9974
200000	0.6782	0.5496	1.3014	2.27
300000	1.0566	0.85	1.944	3.3462

Table 5.7: Running Time (in sec) of $n \log n$ Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 1)

By observing the table of results we have plotted a graph for each algorithm to analyze the practical performance for Test Case 2. In the following, each figure describes the practical performance of sorting algorithms for randomly generated n elements. Each element is an array of string with size three. In Figure 5.4, it is observed that modified bubble sort takes less time when compared to standard inefficient bubble sort besides selection sort performs well when compared to other $O(n^2)$ Sorting algorithms, unlike an array of integers the algorithm's behavior has changed in an array of string . Moreover in Figure 5.5, the standard quick sort performs better than heap sort and merge sort, however improved version quick sort 2 outperforms and takes less time when compared with the rest and proved it is efficient. By observing the fact that the performance has changed when the array of input value changed from integer to string. Above all it still maintains the asymptotic range and this result proved that the theoretical assumption is legitimate.

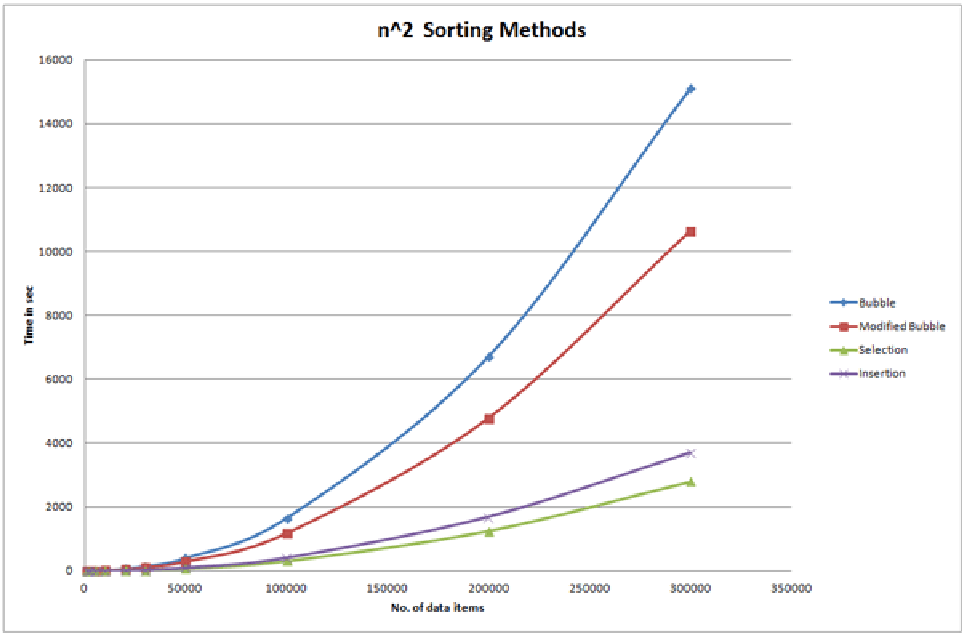


Figure 5.4: n^2 Sorting Class Performance Test Case 1

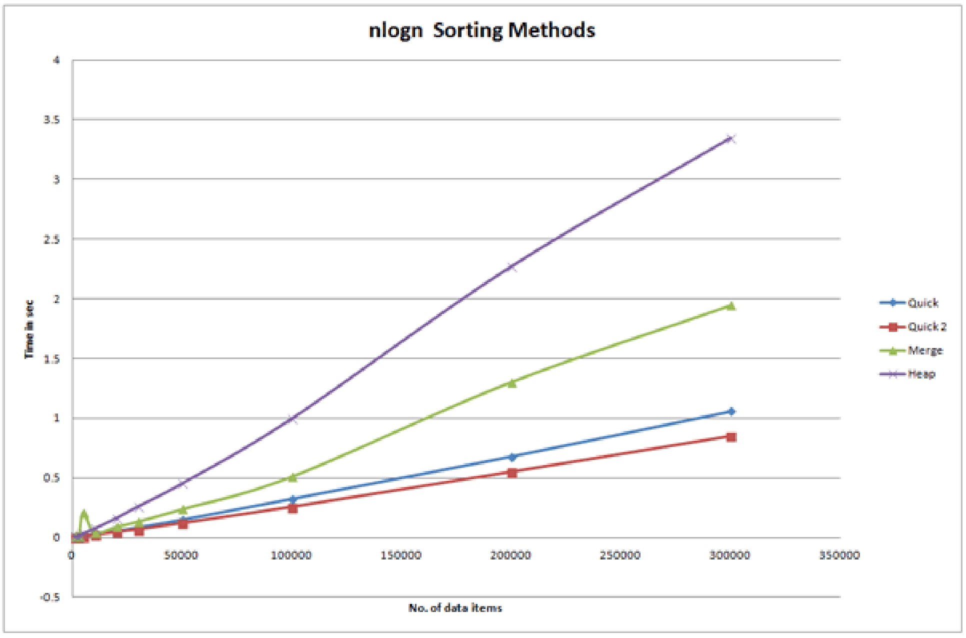


Figure 5.5: $n \log n$ Sorting Class Performance Test Case 1

5.1.3 Test Case 3

Results of Test case 3 are shown in Table 5.8. It shows the running times of n^2 sorting class algorithms. The size of the array elements are varied from 1000 to 300000 and the inputs were strings of size five.

No.of Data Elements	Bubble Sort	Modified Bubble Sort	Selection Sort	Insertion Sort
1000	0.1734	0.1168	0.043	0.0462
2000	0.6938	0.5028	0.158	0.1844
3000	1.5742	1.1132	0.3502	0.4046
5000	4.4826	3.157	1.014	1.1702
10000	18.3128	13.0202	3.9792	4.5846
20000	73.7136	51.6342	16.0676	18.7048
30000	165.3442	116.124	35.888	41.5144
50000	464.9382	325.657	100.3592	115.599
100000	1890.898	1308.372	406.5514	469.5514
200000	7574.37	5276.768	1607.726	1846.264
300000	17082.38	11630.3	3576.252	4129.718

Table 5.8: Running Time (in sec) of n^2 Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 3)

The results of $n \log n$ sorting class algorithms are shown in Table 5.9. The algorithms described in the table are quick sort, merge sort, heap sort and improved quick sort 2.

No.of Data Elements	Quick Sort	Quick Sort 2	Merge Sort	Heap Sort
1000	0.003	0.0028	0.0022	0.007
2000	0.0044	0.0044	0.007	0.0142
3000	0.0068	0.0064	0.013	0.019
5000	0.013	0.0122	0.022	0.0392
10000	0.027	0.0242	0.0424	0.0806
20000	0.061	0.0526	0.0886	0.1766
30000	0.0954	0.083	0.1422	0.2922
50000	0.1648	0.1408	0.2468	0.5034
100000	0.3642	0.308	0.5374	1.0838
200000	0.7712	0.6326	1.287	2.3348
300000	1.1718	0.9896	2.1554	3.5832

Table 5.9: Running Time (in sec) of $n \log n$ Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 3)

In the following, Figure 5.6 and 5.7 describes that there is not much difference in efficiency other than the increase in the running time of each algorithm in the same order equivalent to a string of size three. Moreover, it clearly shows that string of different sizes affects the running time of an algorithm even though it might not change the behavior of an algorithm.

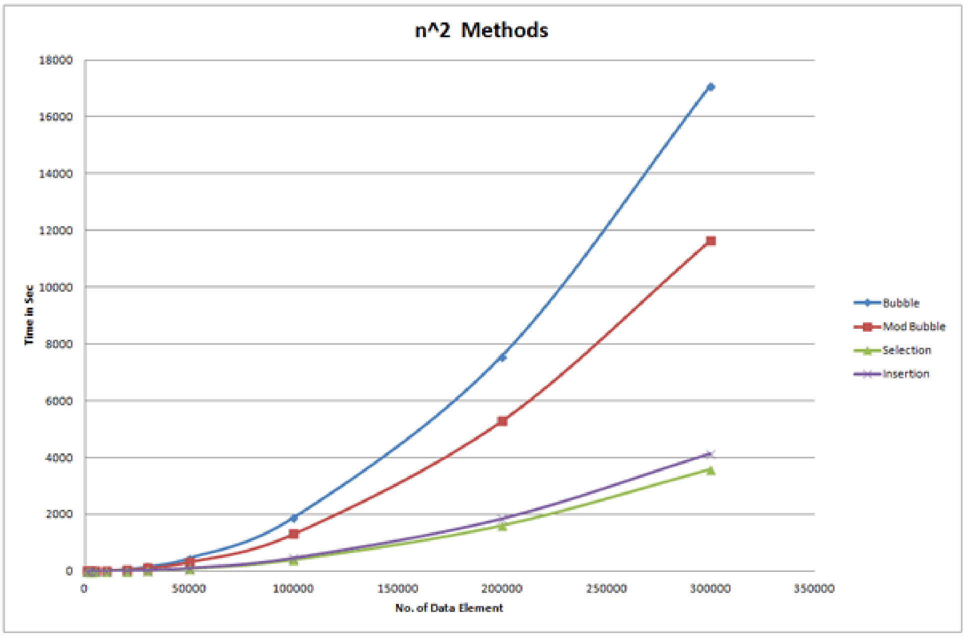


Figure 5.6: n^2 Sorting Class Performance Test Case 3

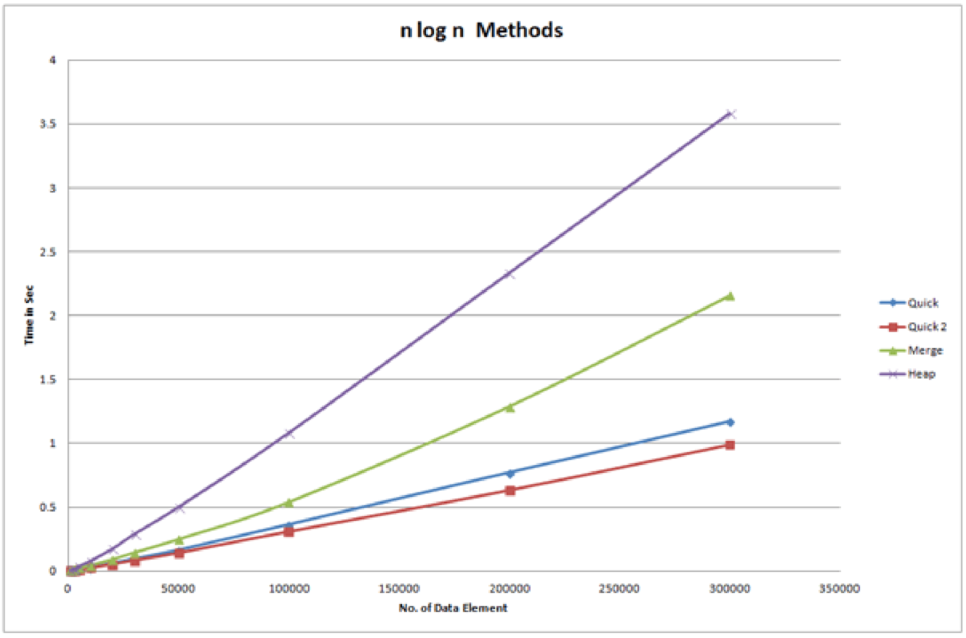


Figure 5.7: $n \log n$ Sorting Class Performance Test Case 3

5.1.4 Test Case 4

Results of Test case 4 are shown in the following Table 5.10, 5.11, 5.12, 5.13. It shows the number of comparison and assignment operations performed during the execution of each algorithm. The size of the array elements are varied from 1000 to 300000. Tables 5.10 and 5.11 delineate the comparison operation counts and Tables 5.12 and 5.13 delineates the assignment operation counts

No.of Data Elements	Bubble Sort	Modified Bubble Sort	Selection Sort	Insertion Sort
1000	999000	499500	499500	250963
2000	3998000	1999000	1999000	993193
3000	8997000	4498500	4498500	2239574
5000	24995000	12497500	12497500	6258267
10000	99990000	49995000	49995000	24998386
20000	399980000	199990000	199990000	99891338
30000	899970000	449985000	449985000	225521101
50000	2499950000	1249975000	1249975000	623929165
100000	9999900000	4999950000	4999950000	2501399429
200000	39999800000	19999900000	19999900000	9988712348
300000	89999700000	44999850000	44999850000	22512064014

Table 5.10: Comparison Operation Counts of n^2 Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 4)

In the above table, the result clearly shows that Bubble sort performs more comparison operation than other sorting methods and this was reduced to nearly half in the modified bubble sort which has performed equal comparison operations like selection sort. Likewise in the following table, the results clearly show that the merge sort does less comparison operations than quick sort, however this could be solved by implementing the quick sort efficiently and the result of quick sort 2 shows this improvement. Moreover the running time performance of heap sort is better than the standard quick sort even though it has done more comparison operations.

No.of Data Elements	Quick Sort	Quick Sort 2	Merge Sort	Heap Sort
1000	10430	7739	9187	20781
2000	24762	16577	20391	46515
3000	39601	27876	32250	74239
5000	71410	50815	57475	132807
10000	157705	103785	124934	290601
20000	347455	236892	269930	631183
30000	534813	360564	422888	990625
50000	967265	643677	740624	1744138
100000	2351571	1358119	1580866	3738462
200000	4360061	2869901	3361895	7976950
300000	6703656	4360804	5223527	12397123

Table 5.11: Comparison Operation Counts of $n \log n$ Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 4)

In the Following tables 5.12 and 5.13, the results of assignment operations performed are shown. In which there is not much difference in the improvement of bubble sort. Thus bubble and modified bubble sort have done almost same number of assignment operations. Moreover selection sort does very less assignment operations than any other sorting algorithms. When compared to comparison operations, merge sort did a more or similar number of assignment operations than quick sort.

Besides the improved version of quick sort has done less assignment operations than other sorting methods.

No.of Data Elements	Bubble Sort	Modified Bubble Sort	Selection Sort	Insertion Sort
1000	757310	757310	2997	252371
2000	2979579	2979579	5997	997191
3000	6718618	6718618	8997	2245572
5000	18723196	18723196	14997	6268265
10000	74987664	74987664	29997	25018384
20000	300187031	300001053	59997	99931336
30000	675974746	673854403	89997	225581099
50000	1875334135	1874924569	149997	623568478
100000	7521632091	7517520837	299997	2501599418
200000	29967010149	29995088793	599997	9989112346
300000	67567201527	67533912621	899997	22512664012

Table 5.12: Assignment Operation Counts of n^2 Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 4)

No.of Data Elements	Quick Sort	Quick Sort 2	Merge Sort	Heap Sort
1000	17856	7724	19951	27279
2000	40445	19333	43903	60464
3000	68039	26430	69807	96272
5000	115306	46676	123615	171300
10000	258784	101020	267231	372505
20000	566170	215879	574463	805123
30000	858905	336265	894463	1259569
50000	1584867	587134	1568927	1896410
100000	3351534	1251358	3337855	4723350
200000	6975982	2670396	7075711	10049815
300000	11030228	4162744	10951423	15588246

Table 5.13: Assignment Operation Counts of $n \log n$ Sorting Class Algorithm on Random Data Averaged 10 Runs (Test Case 4)

5.1.5 Test Case 5

Results of Test case 5 are shown in the following Tables 5.14, 5.15. It shows the amount of memory space utilized by each algorithm during the execution. The size of the array elements are varied from 1000 to 300000. When the memory usage of an algorithm exceeds the primary memory the algorithm will become slow and inefficient, thus it becomes more important.

No.of Data Elements	Bubble Sort	Modified Bubble Sort	Selection Sort	Insertion Sort	Quick Sort	Quick Sort 2	Merge Sort	Heap Sort
1000	308	304	304	300	308	308	304	300
2000	308	308	308	304	312	316	316	304
3000	312	316	316	312	320	320	316	308
5000	324	324	320	320	328	332	344	316
10000	344	344	340	344	348	348	376	344
20000	380	384	384	376	384	388	464	376
30000	424	424	424	420	428	420	534	420
50000	492	492	492	492	500	500	692	492
100000	692	696	688	680	700	698	1084	688
200000	1080	1080	1080	1080	1096	1092	1864	1080
300000	1480	1480	1480	1480	1484	1480	2648	1480

Table 5.14: Memory Space Utilized by Comparison Based Sorting Algorithm on Random Data, All values in KB (Test Case 5)

By observing the table of results presented above, most of the simple sorting algorithms are efficient in terms of memory usage, due to the fact that all sorting and swapping operations are done in the original data sets. Moreover these algorithms need only a constant amount of auxiliary space $O(1)$ for sorting an array of elements.

On the contrary, merge sort utilized more memory than other $n \log n$ sorting algorithms, due to the fact that it uses an extra array to sort elements, therefore it requires $O(n)$ auxiliary space. However, this could be solved by using a linked list data structure instead of arrays. Besides quick sort and heap sort are efficient in memory usage for the reason that both need only $O(1)$ auxiliary space.

In the following table 5.15, the results of non-comparison sort memory usage patterns are presented. By observing the table of data, all these sorting methods utilized more memory than comparison sorting methods. In which counting sort utilized more memory, by reason of it needs a bucket for each value between 0 and max and used an extra array additionally for sorting. Moreover radix sort used 10 buckets along with an extra array for sorting. This clearly shows that it used less memory like other simple sorting methods in the beginning and the memory usage gradually grows with respect to the number of data items.

Besides bucket sort does not need an extra array for sorting, however it created more buckets with respect to the number of data items. Thus radix and bucket sort are two useful abstractions of counting sort and have more in common with counting sort and each other as well.

No.of Data Elements	Radix Sort	Bucket Sort	Counting Sort
1000	304	1040	1048
2000	318	1048	1064
3000	324	1052	1064
5000	344	1064	1088
10000	376	1088	1124
20000	464	1128	1208
30000	532	1160	1280
50000	688	1240	1440
100000	1084	1436	1824
200000	1864	1824	2616
300000	2656	2224	3396

Table 5.15: Memory Space Utilized by Non-Comparison Based Sorting Algorithm on Random Data, All values in KB (Test Case 5)

By using the data in the above tables 5.14 and 5.15, we have plotted a graph to analyze the practical performance for Test Case 5. In the following, Figure 5.8 and 5.9 clearly shows that merge sort and counting sort has memory overhead, likewise we can clearly see that memory usage of radix sort grows gradually with number of data items.

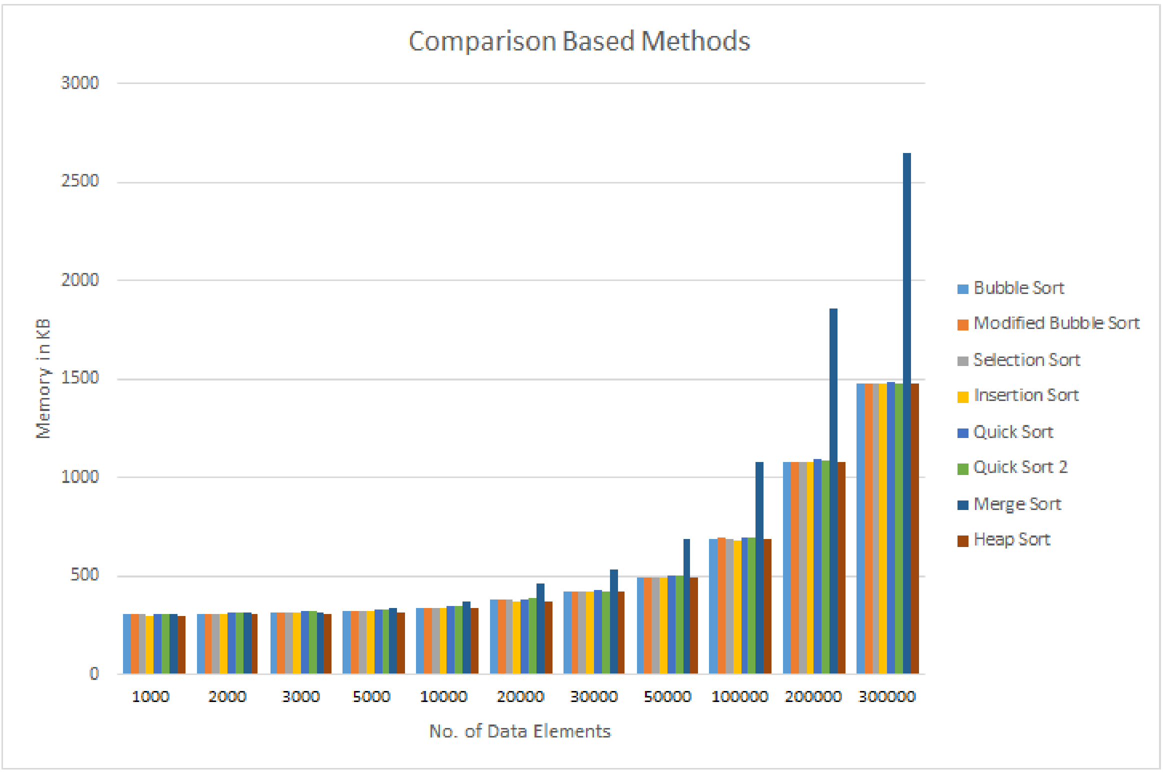


Figure 5.8: Comparison Based Sorting Algorithm Performance Test Case 5

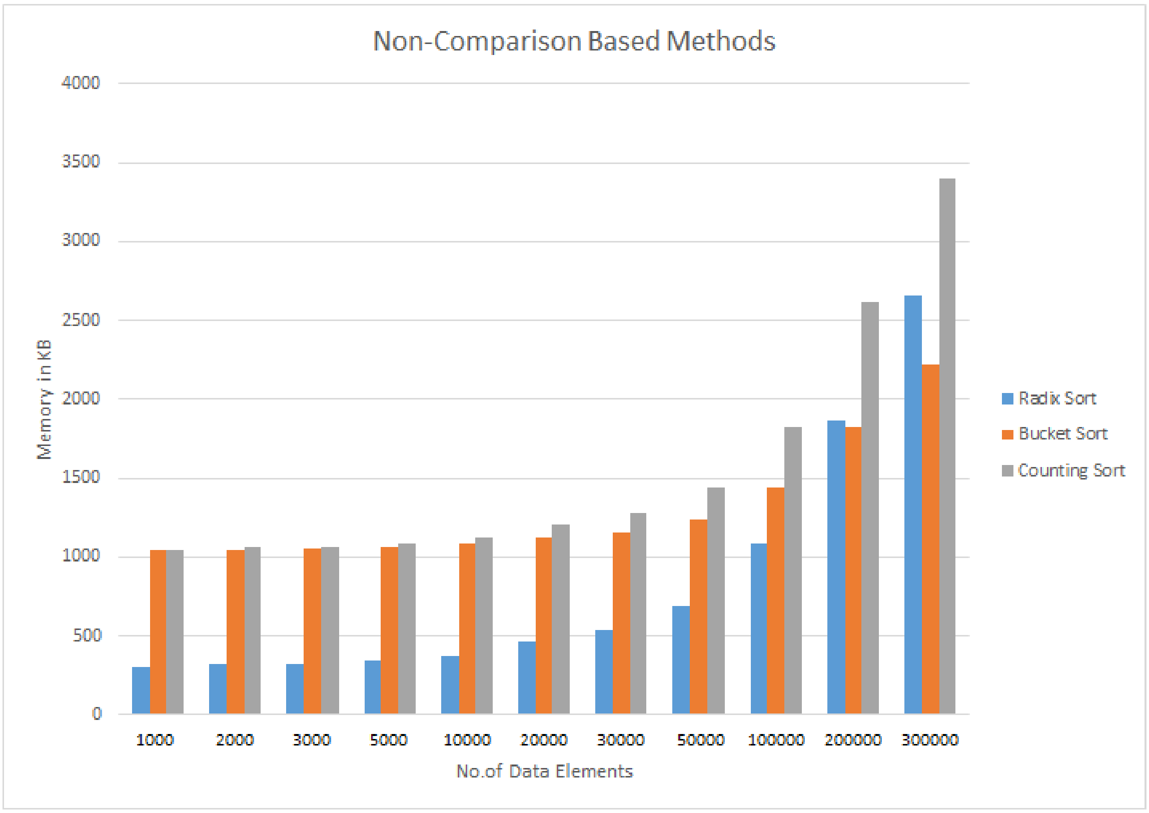


Figure 5.9: Non-Comparison Based Sorting Algorithm Performance Test Case 5

Chapter 6

Discussion and Summary

In this thesis, we have carried out several experiments and obtained the performance of each sorting algorithm. The study investigated the factor that affects the efficiency of each algorithm in terms of overall running time, number of comparisons and assignments and memory used. The results of experiments carried out shows that modified bubble sort is more recommended than standard bubble sort in terms of overall running time and the number of comparisons. Insertion sort had the least running time and made the minimum number of comparisons. Besides, it was simple to implement and thus it would be well suited for small data sets. Similarly selection sort was more efficient than modified bubble sort even though it had the same number of comparisons. However selection sort had a minimum number of assignments among quadratic methods. The relative performance of selection sort and insertion sort differed on different types of input items. For instance, insertion sort was more efficient than selection sort in integer data items and conversely in string data items.

When the number of data grew the improved quick sort had the least run time among the comparison based sorting technique. The standard quick sort was inefficient for integer data items and the performance improved for string data items. Comprehensively the improved quick sort was the most efficient in all the experiments carried out, however it might fail with worst case possibilities. Moreover merge sort also had the least running time, but less efficient than quick sort. Besides it used more memory due to the fact that it needs an auxiliary space for sorting. Heap sort is the least efficient sorting algorithm among the *nlogn* sorting techniques, for the reason that it had more number of comparisons and running time.

Likewise, in non-comparison based sorting methods, bucket sort had the least running time than radix sort and counting sort. Besides bucket sort used minimum memory space for large data sets. Similarly, counting sort also had the least running time, however it used more memory for sorting by reason of it created more buckets and used an extra array. This could be reduced for problem instances in which the maximum key value is significantly smaller than the number of items which results in a highly space efficient method. Moreover the efficiency of radix sort is uncertain due to the fact that keys of fixed length would make the radix sort more efficient than quick sort and merge sort and a distinct key gives at best a time complexity of $O(n \log n)$. However it used less memory than bucket and counting sort for small and medium data sets.

On this study, the empirically observed results show that when the size of data items doubles, the running time goes up by a factor of 3 for quadratic methods and takes slightly more than 2 for linear logarithmic methods. Furthermore, the running time of each algorithm changed relatively with different types of input data, such as integers and string of size three and five. The results show that string of size five takes maximum run time among the other inputs and integer data had least run time. However the different types of input data did not affect the asymptotic order in most methods.

6.1 Problems and their Suitable Sorting Algorithm

Each sorting algorithm is well suited for certain problem this fact validates that sorting algorithms are problem specific. In the following table, according to [18] we have specified some problems and suggested the suitable sorting algorithms.

Problem Definition	Sorting Algorithms
Problems which do not require any extra memory to sort the array of data.	Insertion Sort, Selection Sort,Bubble Sort.
Business applications and Database applications which required a significant amount of extra memory to store data.	Merge Sort.
Problems with the input is extracted from a uniform distribution of a range (0,1).	Bucket Sort.
To sort the record with multiple fields and alphabet with constant size.	Radix Sort
Problems with Small input data sets.	Insertion Sort.
Problems with Large input data sets.	Merge Sort, Quick Sort, Heap Sort.
Problems with the repeated data items in the input array.	Counting Sort.
To sort the record based on address.	Bucket Sort.
Problems with the repeated data items in the input array and the resultant output should maintain the relative order of the data items with equal keys.	Bubble Sort, Merge Sort, Counting Sort, Insertion Sort.
Problems with repeated data items in the input array and the resultant output does not need to be maintained the relative order of the data items with equal keys.	Quick Sort, Heap Sort, Selection Sort.

Table 6.1: Sorting Algorithms According to Problem

Chapter 7

Conclusion

This paper presented the survey and comparison of different sorting algorithms along with the results of practical performance. The main findings of this study are that three factors such as running time, number of swaps and memory used that are critical to efficiency of sorting algorithms. Experiment results clearly show that the theoretical performance behavior is relevant to the practical performance of each sorting algorithm. However there is a slight difference in the relative performance of algorithms in some test cases. In addition, the experiment results proved that algorithm's behavior will change according to the size of elements to be sorted and type of input elements. Which conclude that each algorithm has its own advantage and disadvantage.

In this study, three different types of performance behavior investigated such as $O(n^2)$ class and $O(n \log n)$ class and non-comparison or linear class. Each sorting class outperform the other class under certain conditions such as non-comparison based sorting algorithms outperforms comparison based sorting algorithm in large data-sets and $O(n \log n)$ class algorithm outperforms the $O(n^2)$ class algorithms in large data sets. With the evolution of new technology and the increased usage of the Internet, the data available on the Internet also increased. This creates the demand for fast and efficient sorting algorithm. However, this does not reduce the need of simple sorting algorithms in some cases due to the fact that many sophisticated algorithms use simple sorting algorithms such as insertion sort and shell sort despite of its quadratic performance [16]. As a result, there are more than two algorithms are used to solve a problem as a hybrid method.

Thus, selecting efficient algorithm depends on the type of problem. This concludes that sorting algorithm is problem specific. In the appendix, the implementation code was placed.

Appendix A

Bubble Sort

```
void bubble_sort()
{
    int temp;
    for(long i = 0; i < arraySize; i++)
    {
        for(long j = 0; j < arraySize-1; j++)
        {
            if (data[j] > data[j+1])
            {
                temp        = data[j];
                data[j]     = data[j+1];
                data[j+1]   = temp;
            }
        }
    }
}
```

Appendix B

Modified Bubble Sort

```
void mod_bubble_sort()
{
    int temp, flag = 0;
    for(long i = 0; i < arraySize; i++)
    {
        for(long j = 0; j < arraySize-i-1; j++)
        {
            if (data[j] > data[j+1])
            {
                temp        = data[j];
                data[j]     = data[j+1];
                data[j+1]   = temp;
                                flag = 1;
            }
        }
        if(flag == 0)
        {
            break;
        }
    }
}
```

Appendix C

Selection Sort

```
void selection_sort()
{
        for(long i=0; i<arraySize-1; i++)
        {
                int iMin = i;
                for(int j=i+1; j<arraySize; j++)
                {
                        if(data[j] < data[iMin])
                        {
                                iMin=j;
                        }
                }
                int temp = data[i];
                data[i] = data[iMin];
                data[iMin] = temp;
        }
}
```

Appendix D

Insertion Sort

```
void insertion_sort()
{
    int temp;
    for(long i = 1; i < arraySize; i++)
    {
        temp = data[i];
        long j;
        for(j = i-1; j >= 0 && data[j] > temp; j--)
        {
            data[j+1] = data[j];
        }
        data[j+1] = temp;
    }
}
```

Appendix E

Quick Sort

```
int partition (int start, int end)
{
        int pivot = data[end];
        int partitionIndex = start;
        for(int i = start; i < end; i++)
        {
                if(data[i] <= pivot)
                {
                        swap(data[i], data[partitionIndex]);
                        partitionIndex++;
                }
        }
        swap(data[partitionIndex], data[end]);
        return partitionIndex;
}

void quick_sort (int start, int end)
{
        if(start < end)
        {
                int partitionIndex = partition(start,end);
                quick_sort(start, partitionIndex -1);
                quick_sort(partitionIndex+1, end);
        }
}
```

Appendix F

Quick Sort 2

```
void quick_sort2(int start, int end)
{
    int i = start, j = end;
    int pivot = data[(start + end) / 2];
    while (i <= j)
    {
            while (data[i] < pivot)
                i++;
            while (data[j] > pivot)
                j--;
            if (i <= j)
            {
                swap(data[i], data[j]);
                i++;
                j--;
            }
    };
    if (start < j)
        quick_sort2(start, j);
    if (i < end)
        quick_sort2(i, end);
}
```

Appendix G

Merge Sort

```
void merge(int low, int pivot, int high)
{
    int h,i,j,k;
    h = low;
    i = low;
    j = pivot+1;

    while((h <= pivot)&&(j <= high))
    {
        if(data[h] <= data[j])
        {
            temp[i] = data[h];
            h++;
        }
        else
        {
            temp[i] = data[j];
            j++;
        }
        i++;
    }
    if(h > pivot)
    {
        for(k = j; k <= high; k++)
        {
            temp[i] = data[k];
            i++;
        }
    }
    else
    {
        for(k = h; k <= pivot; k++)
        {
            temp[i] = data[k];
            i++;
        }
    }
    for(k = low; k<=high; k++)
    {
                data[k] = temp[k];
    }
}


void merge_sort(int low, int high)
{
    int pivot;
    if(low < high)
    {
        pivot = (low+high)/2;
        merge_sort(low,pivot);
        merge_sort(pivot+1,high);
        merge(low,pivot,high);
    }
}
```

Appendix H

Heap Sort

```
void shift_down(int low, int high)
{
    int root = low;
    while ((root*2)+1 <= high)
    {
        int leftChild = (root * 2) + 1;
        int rightChild = leftChild + 1;
        int swapIndex = root;
        if (data[swapIndex] < data[leftChild])
        {
            swapIndex = leftChild;
        }
        if ((rightChild <= high) &&
            (data[swapIndex] < data[rightChild]))
        {
            swapIndex = rightChild;
        }
        if (swapIndex != root)
        {
            int temp = data[root];
            data[root] = data[swapIndex];
            data[swapIndex] = temp;
            root = swapIndex;
        }
        else
        {
            break;
        }
    }
    return;
}

void heapify(int low, int high)
{
    int midIndex = (high - low -1)/2;
    while (midIndex >= 0)
    {
        shift_down(midIndex, high);
        --midIndex;
    }
    return;
}

void heap_sort(int size)
{
        heapify(0, size -1);
    int high = size - 1;
    while (high > 0)
    {
        int temp = data[high];
        data[high] = data[0];
        data[0] = temp;
        --high;
        shift_down(0, high);
    }
    return;
}
```

Appendix I

Bucket Sort

```
void bucket_sort(long arraySize)
{
        int buckets [bSize];
        for(int j = 0; j < bSize; ++j)
        {
                buckets[j] = 0;
        }
        for(int i = 0; i < arraySize; ++i)
        {
                ++buckets[data[i]];
        }
        for(int i = 0, j = 0; j < bSize; ++j)
        {
                for(int k = buckets[j]; k > 0; --k)
                {
                        data[i++] = j;
                }
        }
}
```

Appendix J

Radix Sort

```
void radix_sort(long arraySize)
{
        int i, m = 0, exp = 1;
        for(i =0; i < arraySize; i++)
        {
                if(data[i] > m)
                {
                        m = data[i];
                }
        }
        while(m / exp > 0)
        {
                int bucket[BASE] = { 0 };
                for(i = 0; i < arraySize; i++)
                {
                        bucket[data[i] / exp % 10]++;
                }
                for(i = 1; i < BASE; i++)
                {
                        bucket[i] += bucket[i - 1];
                }
                for(i = arraySize - 1; i >= 0; i--)
                {
                        temp[--bucket[(data[i] / exp) % BASE]] =
                }
                for(i = 0; i < arraySize; i++)
                {
                        data[i] = temp[i];
                }
                exp *= BASE;
        }

}
```

Appendix K

Counting Sort

```
void counting_sort(long arraySize)
{
        int i, j, k = 0;
        int C[bSize];
        for(i = 0; i < arraySize; i++)
        {
                if(data[i] > k)
                {
                        k = data[i];
                }
        }
        for(i = 0; i <= k; i++)
        {
                C[i] = 0;
        }
        for(j = 0; j < arraySize; j++)
        {
                C[data[j]] = C[data[j]] + 1;
        }
        for(i = 1; i <= k; i++)
        {
                C[i] = C[i-1] + C[i];
        }
        for(j = arraySize-1; j >= 0; j--)
        {
                i = data[j];
                temp[C[i] - 1] = i;
                C[i] = C[i] - 1;
        }
        for(i = 0; i < arraySize; i++)
        {
                data[i] = temp[i];
        }
}
```

Bibliography

[1] Heap sort. `http://info.mcip.ro/?t=ts&p=7`, March 2014.

[2] Sorting in linear time. `http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap09.htm`, March 2014.

[3] Sorting: Runestone interactive. `http://interactivepython.org/runestone/static/pythonds/SortSearch/sorting.html`, March 2014.

[4] Arne Andersson and Stefan Nilsson. A new efficient radix sort. In *FOCS*, pages 714–721. IEEE Computer Society, 1994.

[5] Kenneth E. Batcher. Sorting networks and their applications. *Spring Joint Computer Conference, AFIPS Proc*, 32:307–314, 1968.

[6] Jon Louis Bentley and M. Douglas McIlroy. Engineering a sort function. *Softw., Pract. Exper.*, 23(11):1249–1265, 1993.

[7] Coenraad Bron. Merge sort algorithm [m1] (algorithm 426). *Commun. ACM*, 15(5):357–358, 1972.

[8] M.S. Garai Canaan.C and M. Daya. Popular sorting algorithms. *World Applied Programming*, 1:62–71, April 2011.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2009.

[10] Ian J. Davis. A fast radix sort. *Comput. J.*, 35(6):636–642, 1992.

[11] E.H.Friend. Sorting on electronic computers. *JACM 3(2)*, pages 34–168, 1956.

[12] Donald E.Knuth. *The Art of Computer Programming Second Edition*, volume 3. ADDISON-WESLEY, 1998.

[13] C.A.R. Hoare. Quicksort. *Commun.ACM*, 4:321, 1961.

[14] Daniel Jimenez. Complexity analysis. `http://www.cs.utexas.edu/users/djimenez/utsa/cs1723/lecture2.html`, June 1998.

[15] Omar khan Durrani, Shreelakshmi V, Sushma Shetty, and Vinutha D C. Analysis and determination of asymptotic behavior range for popular sorting algorithms. *Special Issue of International Journal of Computer Science Informatics(IJCSI), ISSN(PRINT):2231-5292, Vol-2, Issue-1,2.*

[16] Seth Long. Quick sort and insertion sort combo. `http://megocode3.wordpress.com/2008/01/28/8/`, November 2013.

[17] N. Santoro M.D. Atkinson, J.-R. Sack and T.Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, October 1986.

[18] A.D. Mishra and D. Garg. Selection of Best Sorting Algorithm. *International Journal of Intelligent Processing*, 2(2):363–368, July-December 2008.

[19] Samanta Debasis Samanta. *Classic Data Structure, Second Edition.* PHI, 2009.

[20] Pankaj Sareen. Comparison of sorting algorithms(on the basis of average case). *IJARCSSE*, 3:522–532, March 2013.

[21] Harold H Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, M.I.T, 1954.

[22] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis Third Edition.* Prentice Hall, April 2009.

[23] D. L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, July 1959.

[24] Shunichi Toida. Recursive algorithm. `http://www.cs.odu.edu/~toida/nerzic/content/recursive_alg/rec_alg.html`, February 2014.

[25] Ingo Wegener. Bottom-up-heap sort, a new variant of heap sort beating on average quick sort (if n is not very small). In Branislav Rovan, editor, *MFCS*, volume 452 of *Lecture Notes in Computer Science*, pages 516–522. Springer, 1990.

[26] Mark A. Weiss. *Data Structures Algorithms Analysis in C++ (Third Edition).* Prentice Hall, March 2006.

[27] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[28] Niklaus Wirth. *Algorithms+DataStructures=Programs.* Prentice Hall.

[29] Ray Wisman. The fundamentals: Algorithms, the integers, and matrices, June 1998.

[30] Gongjun Yan. Growth of function. `http://www.cs.odu.edu/~ygongjun/courses/cs381/cs381content/function/growth.html`, March 2014.