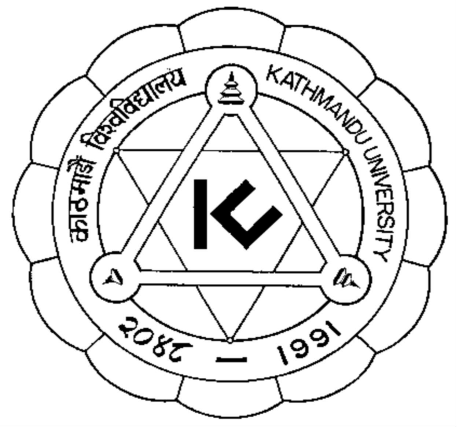
**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**

**A Project Report**

**On**

**“Sorting Techniques and Complexity”**

**[Code No: COMP 202]**

**(For the partial fulfillment of 2nd Year/ 1st Semester in Computer Science)**

**Submitted by:**

**Mandip Shrestha (48)**

**Vaibhav Sapkota (41)**

**Nikit K.C. (70)**

**Aadarsha Jung Basnet (6)**

**Suchit Kumar Mahatman (26)**

**Submitted to:**

**Dr. Gajendra Sharma**

**Department of Computer Science and Engineering**

**Submission Date: 28th February, 2020**

Abstract

Sortingalgorithm is one of the most basic research fields in computer science. Its goal is to make record easier to search, insert and delete. The complexity analysis is used to evaluate the suitable searching techniques that is used for each use-case with statistical methods backing up the claims. Used to represent data in more readable formats, Sorting is among the most common programming process. It is essential in handling of the elements by arranging them in certain order than handling the random elements where the data is meaningfully structured. Sorting is used to rearrange a list of elements in an ordered sequence as in:   
maintaining the information and facilitating commonly used CRUD (create, read, update and delete) operations is crucial. Efficient algorithms like binary search can greatly benefit from ordered data structure in any data-driven application. Also, the data structure, number of elements, algorithm design constraints and computational time-complexity is considered for each use-case. In this report, we have explained, implemented and compared some commonly used sorting techniques. Some sorting algorithms are stable by its nature such as insertion sort, merge sort, bubble sort, while some sorting algorithms are not, such as quick sort, any given sorting algorithm which is not stable can be modified to be stable.

**Keywords:** *Algorithms, Sorting, Time Complexity*

**Contents**

[Abstract II](#_Toc33739488)

[Acronyms IV](#_Toc33739489)

[Chapter 1: Introduction 1](#_Toc33739490)

[1.1 Background 1](#_Toc33739491)

[1.3 Motivation and Significance 2](#_Toc33739492)

[Chapter 2: Procedure, methods and implementation 2](#_Toc33739493)

[Overview: 2](#_Toc33739494)

[2.1 Explanation 3](#_Toc33739495)

[2.1.1 Bubble Sort 3](#_Toc33739496)

[2.1.1.1 Working Mechanism: 3](#_Toc33739497)

[2.1.1.2 Implementation and worst-case analysis: 3](#_Toc33739498)

[2.1.1.3 Example 4](#_Toc33739499)

[2.1.1.4 Complexity analysis on all cases 6](#_Toc33739501)

[2.1.2 Selection Sort 6](#_Toc33739502)

[2.1.2.1 Working Mechanism: 6](#_Toc33739503)

[2.1.2.1 Implementation and worst-case analysis 7](#_Toc33739504)

[2.1.2.2 Example 8](#_Toc33739505)

[2.1.2.3 Graph for different cases 9](#_Toc33739506)

[2.1.2.4 Complexity analysis on all cases 9](#_Toc33739507)

[2.1.3 Insertion Sort 9](#_Toc33739508)

[2.1.3.1 Working Mechanism: 10](#_Toc33739509)

[2.1.3.2 Implementation and worst-case analysis: 10](#_Toc33739510)

[2.1.3.3 Example 11](#_Toc33739511)

[2.1.3.4 Graph (N vs. Time) for different cases 13](#_Toc33739512)

[2.1.3.5 Complexity analysis on all cases 13](#_Toc33739513)

[2.1.4 Merge Sort 14](#_Toc33739514)

[2.1.4.1 Working Mechanism with example 14](#_Toc33739515)

[2.1.4.2 Implementation and case analysis: 18](#_Toc33739516)

[2.1.4.3 Graph (N vs. Time) for different cases 18](#_Toc33739517)

[2.1.5 Heap Sort 19](#_Toc33739520)

[2.1.5.1 Working Mechanism: 19](#_Toc33739521)

[2.1.5.2 Implementation and case analysis: 19](#_Toc33739522)

[2.1.5.3 Graph (N vs. Time) for different cases 22](#_Toc33739523)

[References 23](#_Toc33739524)

**List of Figure**

**Figure Page No.**

[Figure 2.1.1 Implementation of Bubble Sort 4](file:////Users/skelliger7/Desktop/miniReport.docx#_Toc33574257)

[Figure 2.1.2 Example and N vs Time-Graph of Bubble Sort 5](file:////Users/skelliger7/Desktop/miniReport.docx#_Toc33574258)

[Figure 2.2.1 Implementation of Selection Sort 7](file:////Users/skelliger7/Desktop/miniReport.docx#_Toc33574259)

[Figure 2.2.2 Example and N vs Time-Graph of Selection Sort 8-9](file:////Users/skelliger7/Desktop/miniReport.docx#_Toc33574260)

[Figure 2.3.1 Implementation,Example and N v TimeGraph of Insertion Sort 10-13](file:////Users/skelliger7/Desktop/miniReport.docx#_Toc33574261)

[Figure 2.4.1 Implementation,Example and N vs Time-Graph of Merge Sort 15](file:////Users/skelliger7/Desktop/miniReport.docx#_Toc33574262)-18

[Figure 2.5.1 Implementation,Example and N vs Time-Graph of Heap Sort 20-22](file:////Users/skelliger7/Desktop/miniReport.docx#_Toc33574263)

Acronyms

**CRUD- Create Read Update Delete**

# Chapter 1: Introduction

## 1.1 Background

Large number of algorithms have been developed to improve sorting, each of them having a different mechanism to reorder elements which increase the performance and efficiency of the practical applications and reduce time complexity of each one. As years goes by, data has been growing rapidly. Exponential growth of data and information leads to increasing development of sort algorithms. Developing sorting algorithms with improved performance and decreasing complexity has attracted a great deal of research. Any effect of sorting algorithm enhancement of the current algorithms or implementation of better techniques minimizes performance heads on further processing of data, especially in fields like big-data, machine learning and so on.

There are factors like *Time Complexity, Stability* and *Space Complexity* that must be taken in consideration when comparing between various sorting algorithms. Sorting algorithms work for a specific environment where some of them work best on small data-sets, while others work better on medium-sized or larger ones. The time complexity of an algorithm can be determined by the amount of time taken by an algorithm to run. The complexity of algorithmic is generally written in a form known as Big-O (N) notation, where the O represents the complexity of the algorithm and a value N represents the size of the data the algorithm is run against. Stability, being another factor means that the algorithm should keep elements with equal values in the same relative order in the output as they were in the input. The third factor is memory space, algorithm that used recursive techniques need more copies of sorting data that affect to memory space. The many previous researches have been suggested to enhance the sorting algorithm to maintain memory and improve efficiency.

We explain machine-independent mathematical reasoning and visualize a machine-dependent comparison for different sized data to suggest a suitable sorting technique for each use-case (best, average and worst cases). Here, we only consider time complexity to benchmark the algorithms.

**1.2** **Objectives**

* To benchmark different sorting algorithms based on time complexity.
* To experiment and suggest a performance-based sorting algorithm.
* Visualize an idea of data structure specific algorithm designs.

## 1.3 Motivation and Significance

Data science and analysis is often regarded as one of the most prominent job in entire lineup of jobs this century. Statistical analysis and prediction are still an under-developed sector in computer science as per increasing data demands and such methods can greatly benefit from small improvements in algorithms where our main motivation is directed towards.

This study holds significant importance: Firstly, it helps us to understand and compare algorithms based on time complexity. Moreover, it gives us an idea of data structure specific algorithm designs. Finally, it makes a significant contribution to the existing literature as it relates to computational data processing methodologies.

# Chapter 2: Procedure, methods and implementation

## Overview:

Our project compares sorting algorithms in terms of time complexity. We have implemented following algorithms: bubble sort, selection sort, insertion sort, quick sort, merge sort and heap sort. Then, we generate up to N linearly spaced elements. We checked each case, the elements were generated such that they were either sorted, random or reversely sorted integers to maintain consistency while comparing algorithms. We then structure the measured time period and then plot the number of elements vs time taken graph to visually compare different sorts.

## 2.1 Explanation

### 2.1.1 Bubble Sort

Bubble sort is the sorting algorithm largely used in computer science. The primary concept used in Bubble Sort is based on the idea of repeatedly comparing corresponding adjacent elements repeatedly and swapping their positions in case they exist in unsorted order.

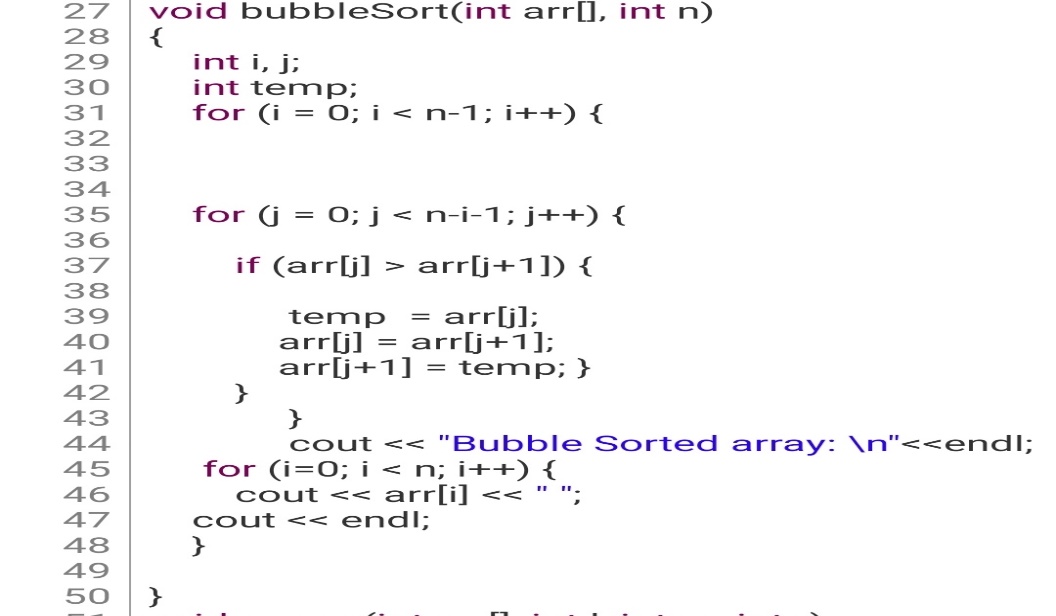
#### 2.1.1.1 Working Mechanism:

1. Starting with the first element (index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element**.**RepeatStep1.

#### 2.1.1.2 Implementation and worst-case analysis:

Bubble sort algorithm contains two computations:

1. Comparison
2. Swapping



In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

(N -1) + (N -2) + (N -3) + ..... + 3 + 2 + 1

Sum = N (N -1)/2

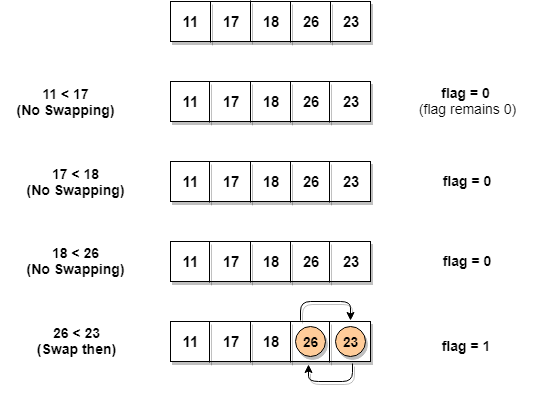
i.e O(N^2)

Hence, we conclude that worst running time for bubble sort is definitely O(N^2).

#### 2.1.1.3 Example

Let's consider an array with values {11, 17, 18, 26, 23}

Below, we have a pictorial representation of how the optimized bubble sort will sort the given array.



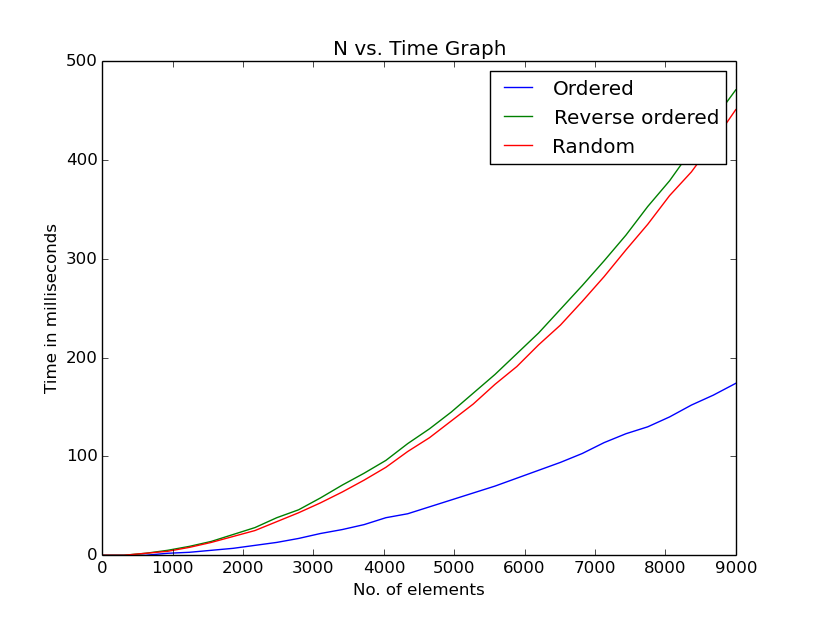
As we can see, in the first iteration, swapping took place, hence we updated our flag value to 1, as a result, the execution enters the for loop again. But in the execution will break out of loop.2.1.1.3 Graph for different cases:

Figure .1.2 Bubble Sort on Sorted, reversely sorted and random cases

#### 

#### 2.1.1.4 Complexity analysis on all cases

Worst Case Time Complexity [ Big-O ]: **O(N2)**

Best Case Time Complexity [Big-omega]: **Ω(N)**

Average Time Complexity [Big-theta]: **Θ(N2)**

Space Complexity: **O(1)**

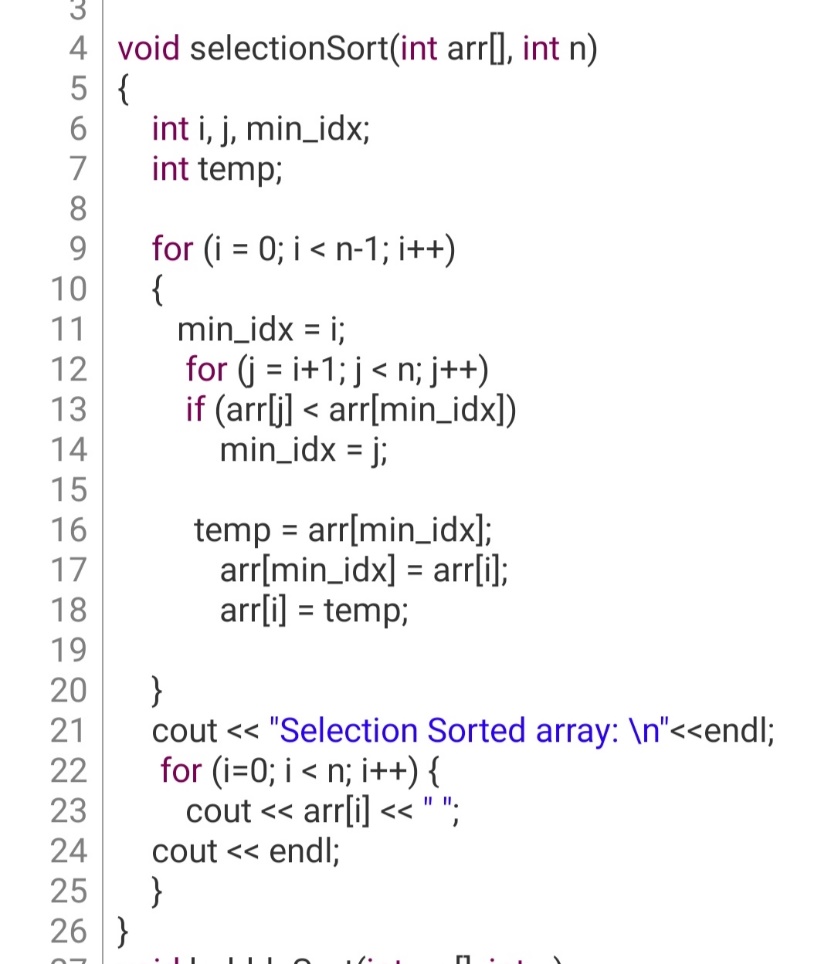
### 2.1.2 Selection Sort

Selection sort is conceptually the simplest sorting algorithm. This algorithm will first find the smallest element in the array and swap it with the element in the first position, then it will find the secondsmallest element and swap it with the element in the second position, and it will keep on doing this until the entire array is sorted.

#### 2.1.2.1 Working Mechanism:

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the sub array, starting from index 1, till the last index.
3. We replace the element at the second position in the original array, or we can say at the first position in the sub array, with the second smallest element.
4. This is repeated, until the array is completely sorted.

#### 2.1.2.1 Implementation and worst-case analysis



Selection sort algorithm contains two kinds of computation:

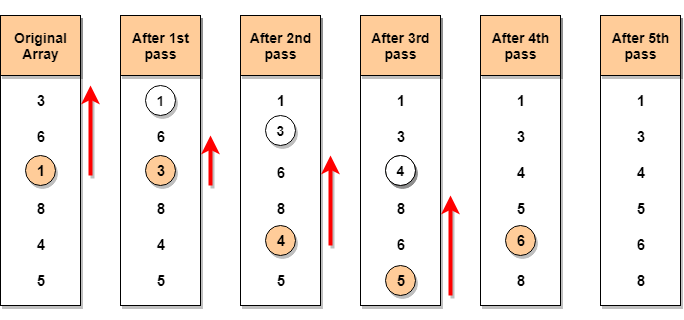
1. Comparisons
2. Swaps

Selecting the lowest element requires scanning all n elements (this takes N-1 comparisons) and then swapping it into the first position.  
   
Finding the next lowest element requires scanning the remaining N - 1 elements and so on,   
= (N-1) + (N-2) + ... + 2 + 1 = N(N-1)/2   
**= O(N^2) comparisons.**

#### 2.1.2.2 Example

Let's consider an array with values {3, 6, 1, 8, 4, 5}

Below, we have a pictorial representation of how selection sort will sort the given array.

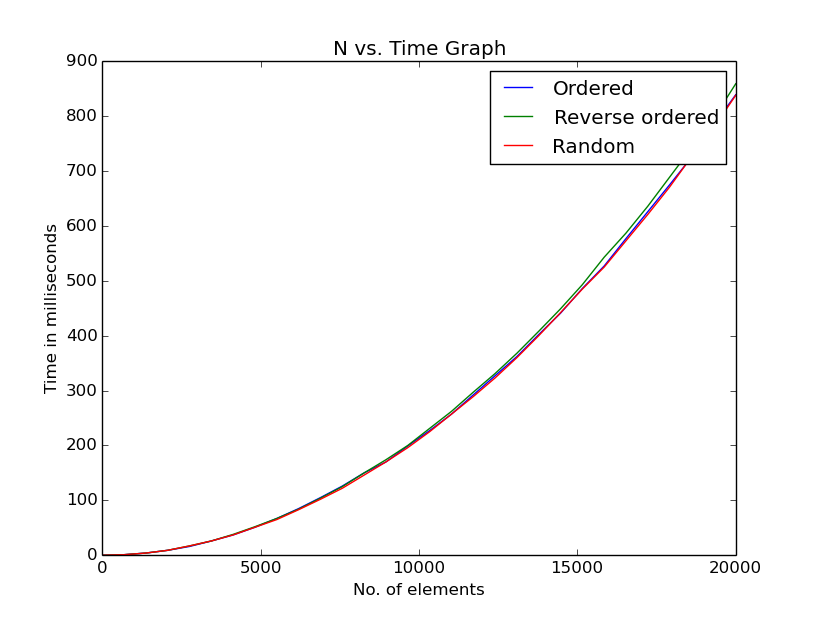


In the **first** pass, the smallest element will be 1, so it will be placed at the first position.

Then leaving the first element, **next smallest** element will be searched, from the remaining elements. We will get 3 as the smallest, so it will be then placed at the second position.

Then leaving 1 and 3(because they are at the correct position), we will search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

#### 2.1.2.3 Graph for different cases



#### 2.1.2.4 Complexity analysis on all cases

Worst Case Time Complexity [ Big-O ]: **O(N2)**

Best Case Time Complexity [Big-omega]: **Ω(N2)**

Average Time Complexity [Big-theta]: **Θ(N2)**

Space Complexity: **O(1)**

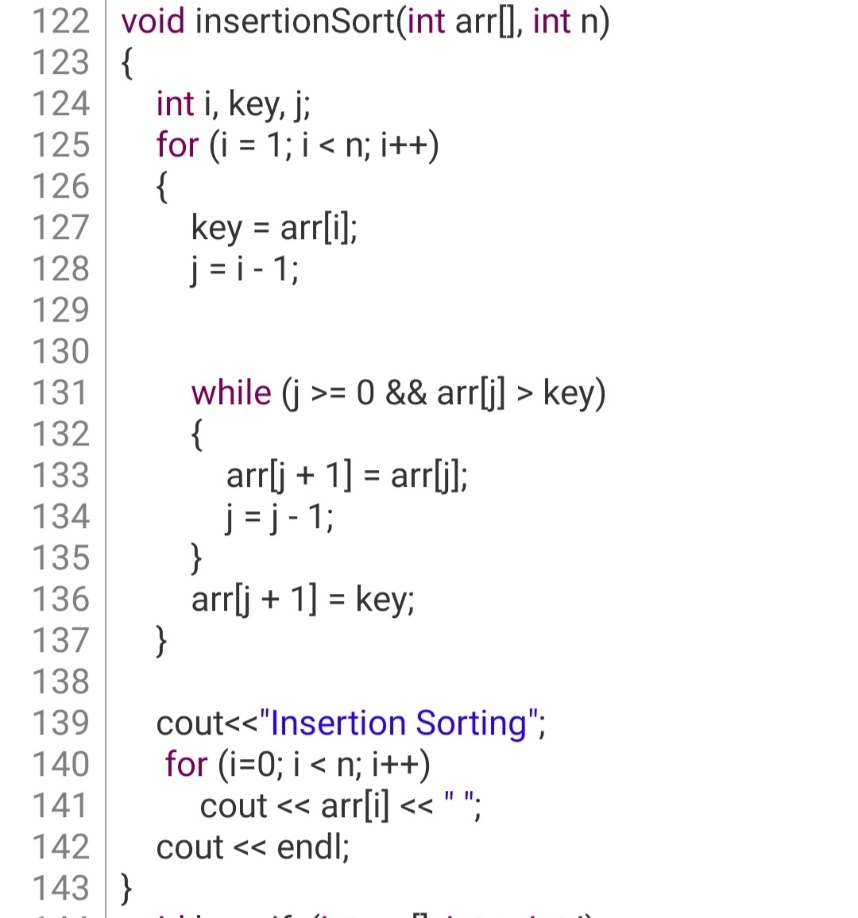
### 2.1.3 Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

#### 2.1.3.1 Working Mechanism:

1. The first step involves the comparison of the element in question with its adjacent element.
2. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
3. The above procedure is repeated until all the element in the array is at their apt position.

#### 2.1.3.2 Implementation and worst-case analysis:



Insertion sort algorithm contains four kinds of computation:

1. Comparisons
2. Shifts
3. Insertion
4. Removal

The worst case for insertion sort will occur when the input list is in decreasing order. To insert the last element, we need at most *n*−1 comparisons and at most *n*−1 swaps. To insert the second to last element, we need at most *n*−2 comparisons and at most *n*−2 swaps, and so on. The number of operations needed to perform insertion sort is therefore: 2×(1+2+⋯+*n*−2+*n*−1). To calculate the recurrence relation for this algorithm, use the following summation:

*q*=1∑*p*​*q*=2*p*(*p*+1) ​.

It follows that:

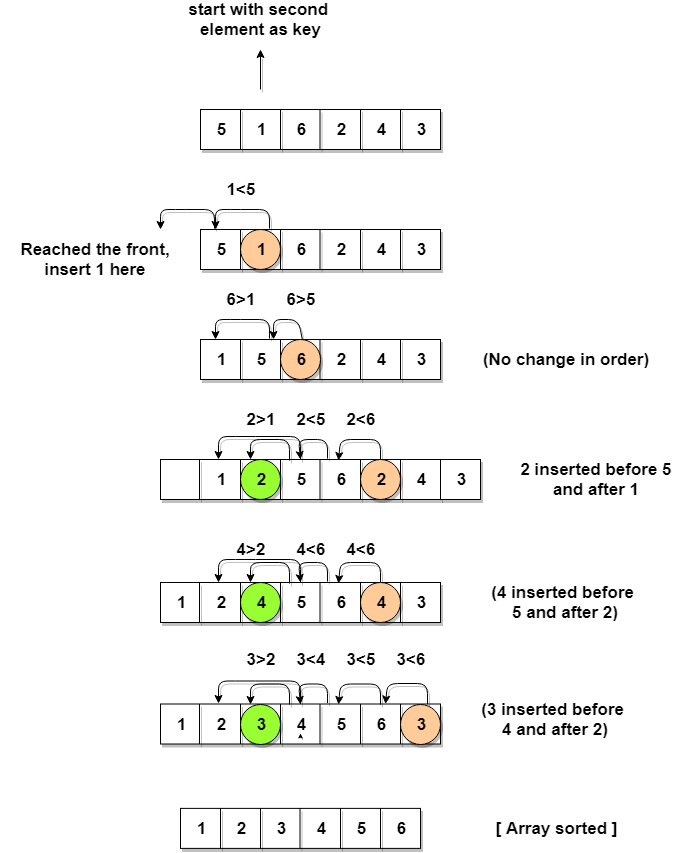
(2(*n*−1) (*n*−1+1))/2 = n(n-1).

Use the master theorem to solve this recurrence for the running time. As expected, the algorithm's complexity is *O*(*n^*2). When analyzing algorithms, the average case often has the same complexity as the worst case. So insertion sort, on average, takes *O*(*n^*2) time.

#### 2.1.3.3 Example

Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array.



As you can see in the diagram above, after picking a key, we start iterating over the elements to the left of the key.

We continue to move towards left if the elements are greater than the key element and stop when we find the element which is less than the key element.

And, insert the key element after the element which is less than the key element.

#### 2.1.3.4 Graph (N vs. Time) for different cases

#### 2.1.3.5 Complexity analysis on all cases

Worst Case Time Complexity [ Big-O ]: **O(n2)**

Best Case Time Complexity [Big-omega]: **Ω(N)**

Average Time Complexity [Big-theta]: **Θ(N2)**

Space Complexity: **O(1)**

### 2.1.4 Merge Sort

**Merge sort** is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sub-list consists of a single element and **merging** those sub-lists in a manner that results into a **sorted** list.

#### 2.1.4.1 Working Mechanism with example

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two subarrays with 3 elements each.

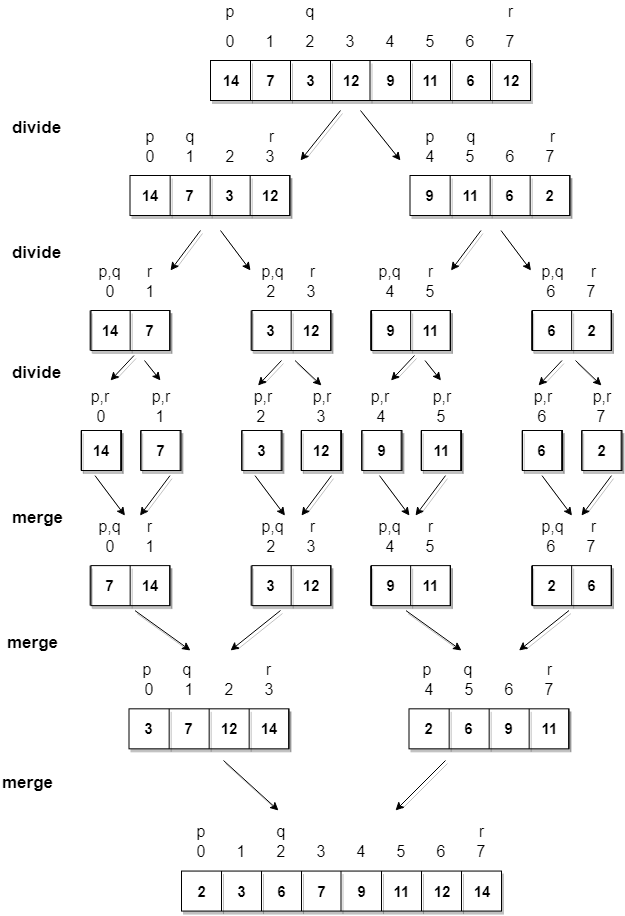
But breaking the orignal array into 2 smaller subarrays is not helping us in sorting the array.

So we will break these subarrays into even smaller subarrays, until we have multiple subarrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into subarrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted subarrays, step by step to form one single sorted array.

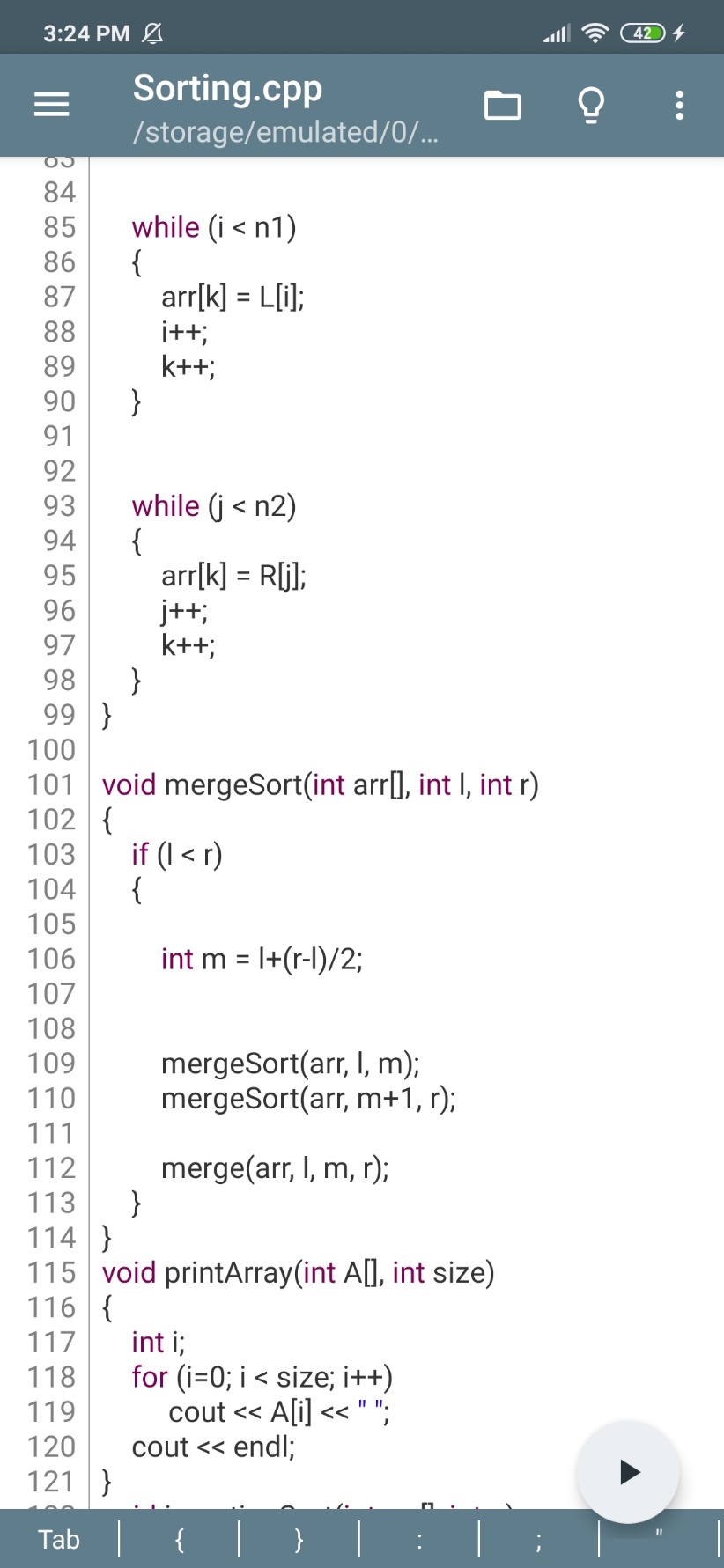
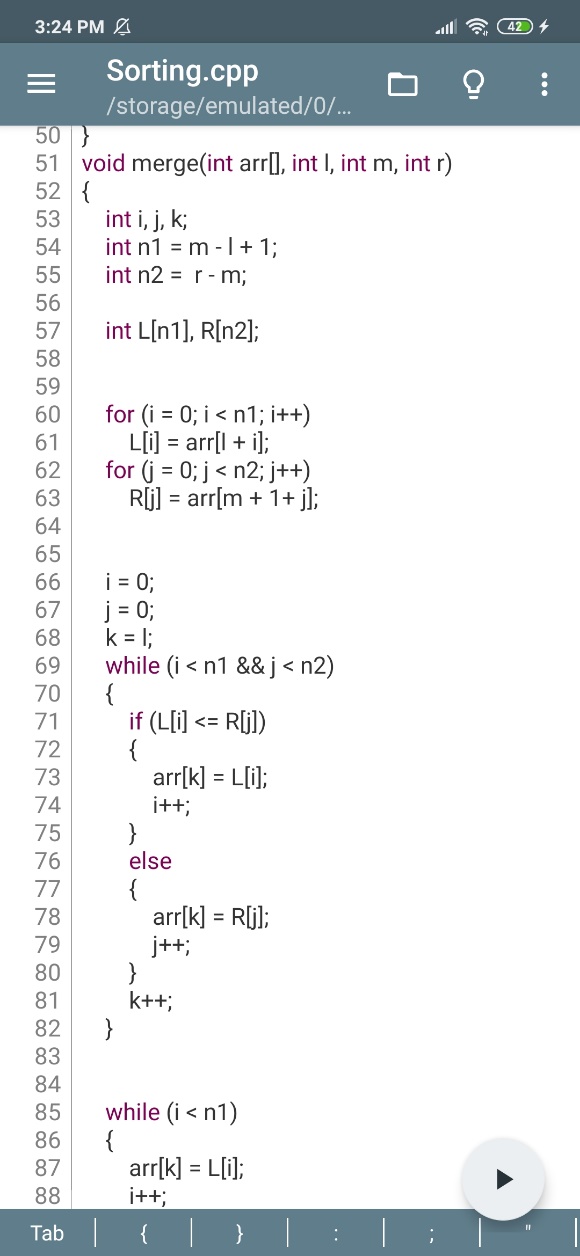
Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}

Below, we have a pictorial representation of how merge sort will sort the given array.



In merge sort we follow the following steps:

1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
2. Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two subarrays, from p to q and from q + 1 to r index.
3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.



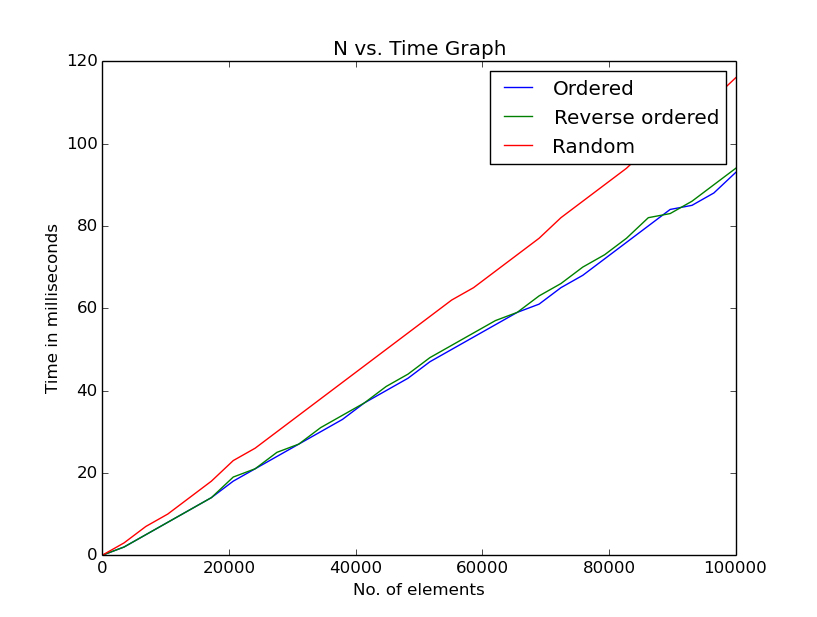
#### 2.1.4.2 Implementation and case analysis:

The master theorem tells us that the solution to this recurrence is

*T* (*n*) =*O* (*n* log *n*).

Mergesort runs in *O* (*n* log *n*) time in its best case, worst case, and average case. That means that no matter what the input, mergesort will operate in *O*(*n* log*n*) time.

#### 2.1.4.3 Graph (N vs. Time) for different cases



### 

### 

### 2.1.5 Heap Sort

Heap sort is a comparison based **sorting** technique based on Binary **Heap** data structure. It is similar to selection **sort** where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

#### 2.1.5.1 Working Mechanism:

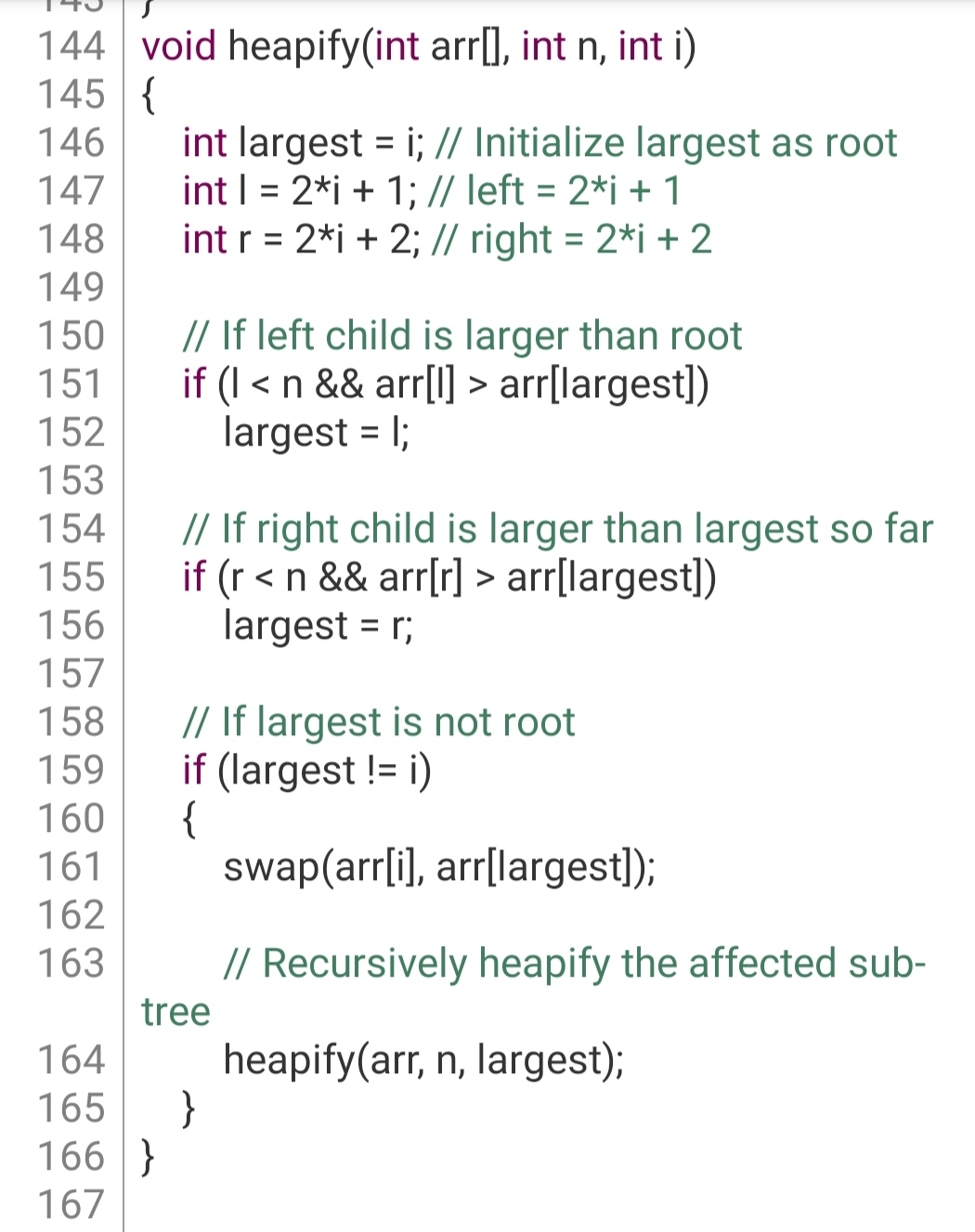
Heap sort algorithm is divided into two basic parts:

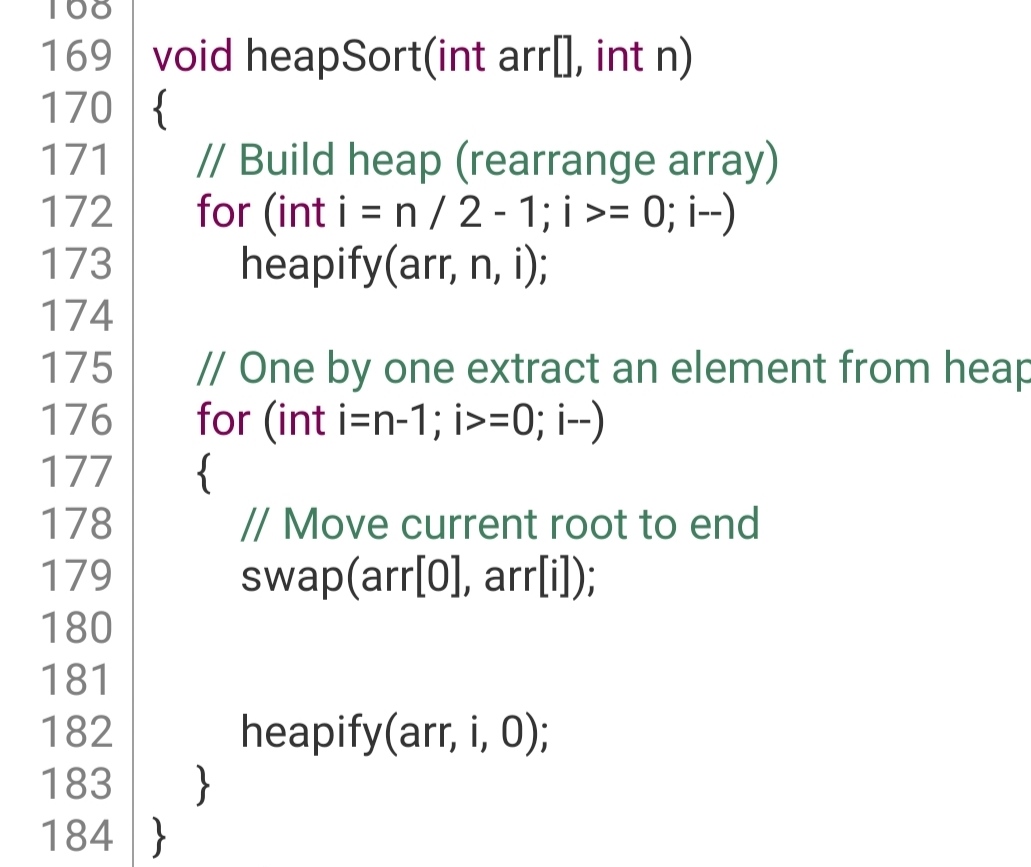
* Creating a Heap of the unsorted list/array.
* Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.

In the below algorithm, initially heapsort() function is called, which calls heapify() to build the heap.

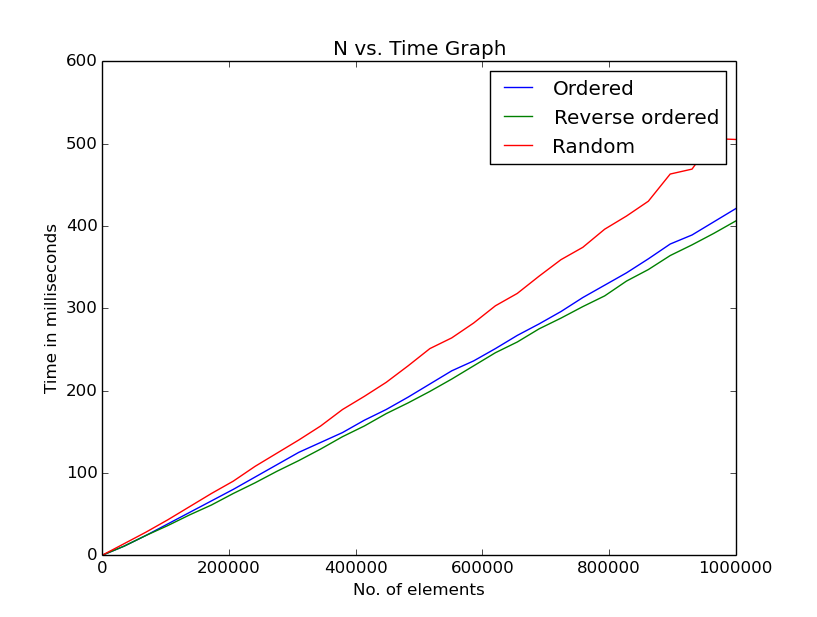
#### 2.1.5.2 Implementation and case analysis:





Heapsort has a worst- and average-case running time of *O*(*n*log*n*) like mergesort, but heapsort uses *O*(1) auxiliary space (since it is an in-place sort) while mergesort takes up *O*(*n*) auxiliary space, so if memory concerns are an issue, heapsort might be a good, fast choice for a sorting algorithm. Quicksort has an average-case running time of *O*(*n*log*n*) but has notoriously better constant factors, making quicksort faster than other *O*(*n*log*n*)-time sorting algorithms. However, quicksort has a worst-case running time of *O*(*n*^2) and a worst-case space complexity of *O*(log*n*), so if it is very important to have a fast worst-case running time and efficient space usage, heapsort is the best option. Note, though, that heapsort is slower than quicksort on average in most cases.

#### 2.1.5.3 Graph (N vs. Time) for different cases



**Chapter 3: Conclusion and Recommendations**

**3.1 Conclusion:**

In this report, we have discussed the best, average and worst case complexity of different sorting techniques with possible scenarios.

**3.2 Recommendation:**

# References

[1] S. Shastri, V. Mansotra, and A. Sharma, “Sorting Algorithms and their Run-Time Analysis with C#,” 2015.

[2] I. M. Alturani, A. Mahmoud, and I. Alturani, “Review on Sorting Algorithms A Comparative Study,” *Int. J. Comput. Sci. Secur.*, 2013.

[3] S. Jadoon, S. F. Solehria, P. Dr, S. Rehman, and P. H. Jan, “Design and Analysis of Optimized Selection Sort Algorithm.” .

[4] M. Khan, S. Shaheen, and F. Qureshi, “Comparative Analysis of five Sorting Algorithms on the basis of Best Case, Average Case, and Worst Case,” *Int. J. Inf. Technol. Electr. Eng. 2306-708X*, vol. 3, pp. 1–10, 2014.

[5] Cisco, “Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2015 -- 2020,” 2016.

[6] A. Kazim, “A comparative study of well known sorting algorithms,” *Int. J. Adv. Res. Comput. Sci.*, vol. 8, no. 1, 2017.