Computer Engineering
CSE 222/505 - Spring 2023
Homework # Report

Name: Ahmet Furkan Ekinci
Student Number: 200104004063

1. Introduction

Purpose of Document

This document is report for this project. The report is going to explain you what stages the homework was going through.

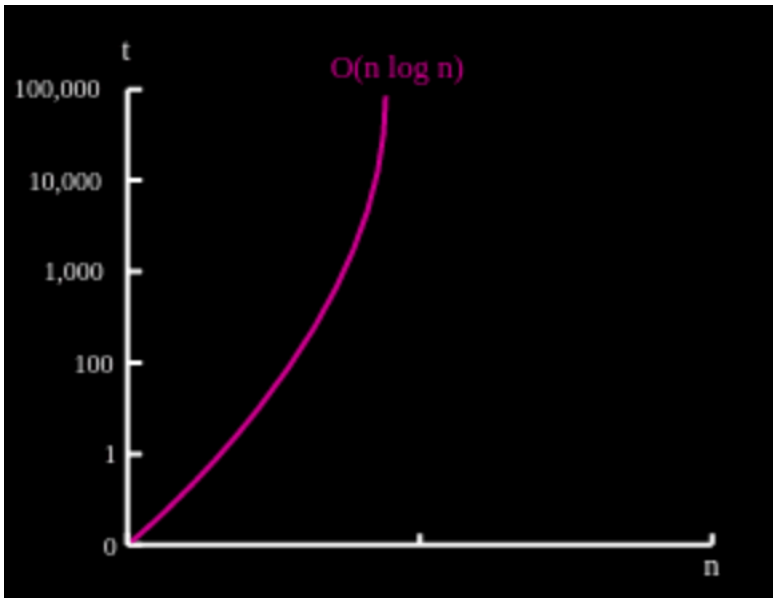
Purpose of Project

The purpose of this assignment is to create some sorting algorithm using the map data structure. When we do and understand the whole of this assignment, we will actually understand the issues of map and sorting algortihm.

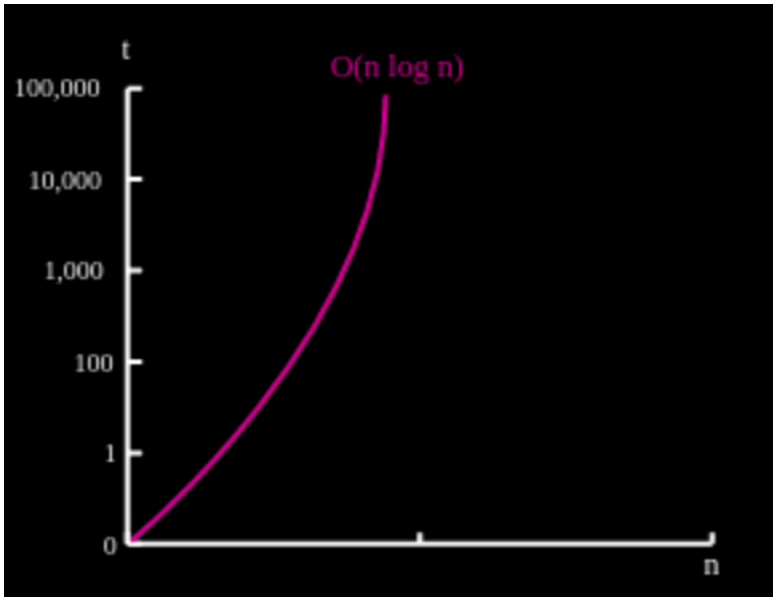
2. Time Complexity Analysis

1. Merge Sort

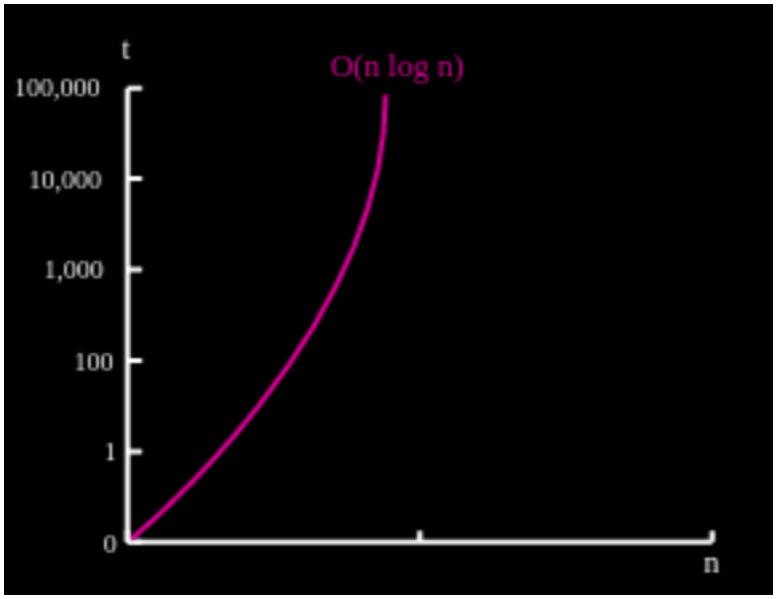
Best Case: O(n(log n))



Avarage Case: O(n(log n))



Worst Case: O(n(log n))



Merge Sort is a "divide and conquer" algorithm. The working logic of the algorithm is based on first dividing the sequence that needs to be sorted in half and then merging the segmented sequences. This concatenation operation combines the fragmented arrays to form a larger ordered array.

At each stage of the Merge Sort, the sequence is split into two, and this process continues in a logarithmic manner. At each split stage,

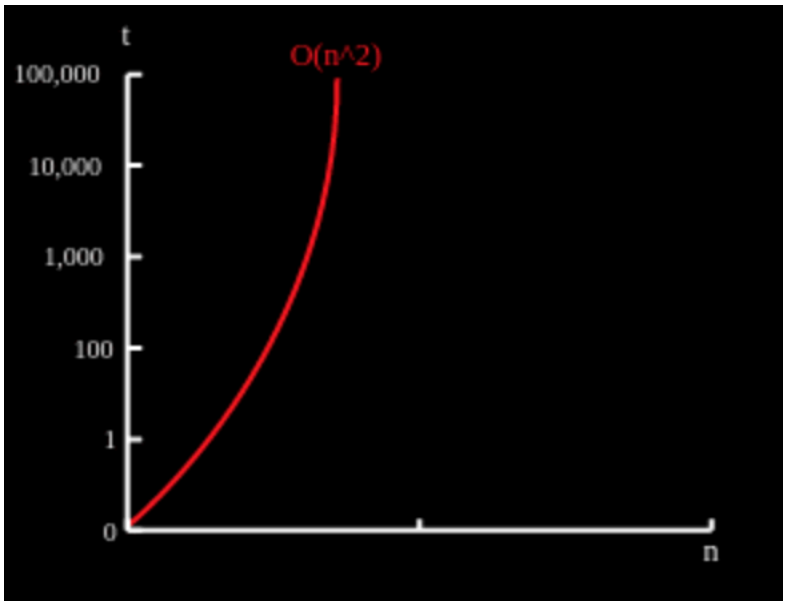
the size of the array is halved. Hence, division operations are performed log n times (where n represents the initial size of the array).

After the division operation, the merge operation is performed. In this process, as the segmented arrays are combined, the elements are compared and a new array is created in a sequential manner. The merge operation includes the merge operations where each element is compared at most once, and this operation is performed for n elements.

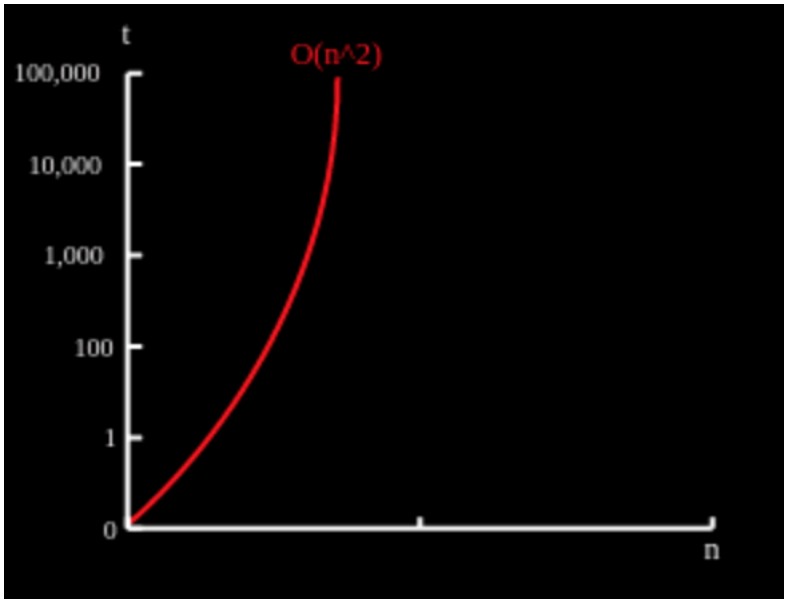
As a result, whole stages of Merge Sort are executed log n times, and each stage is processed on n elements. Therefore, the best, average, and worst-case time complexity of the Merge Sort is the same as O(n log n). The algorithm performs the same regardless of the initial state of the array or the ordering of the elements.

2. Selection Sort

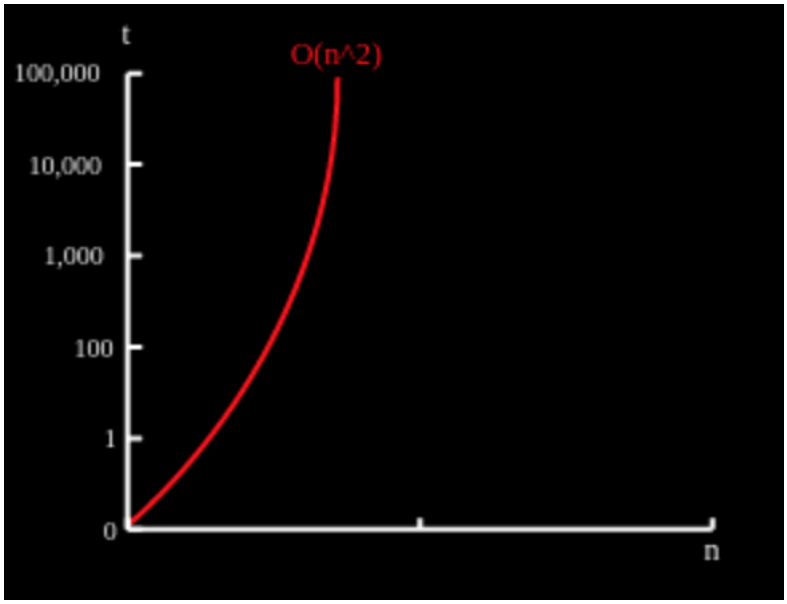
Best Case: O(n^2)



Avarage Case: O(n^2)



Worst Case: O(n^2)



The reason the Selection Sort has the same best, average, and worst-case time complexity is because the algorithm does the same number of operations in each case.

Selection Sort performs sorting by finding the smallest element of the array and replacing it in each iteration. At each iteration, the rest of the array is scanned to find the minimum element and this element is placed at the beginning of the array. This operation is performed n times depending on the size of the array, where n is the number of

elements of the array.

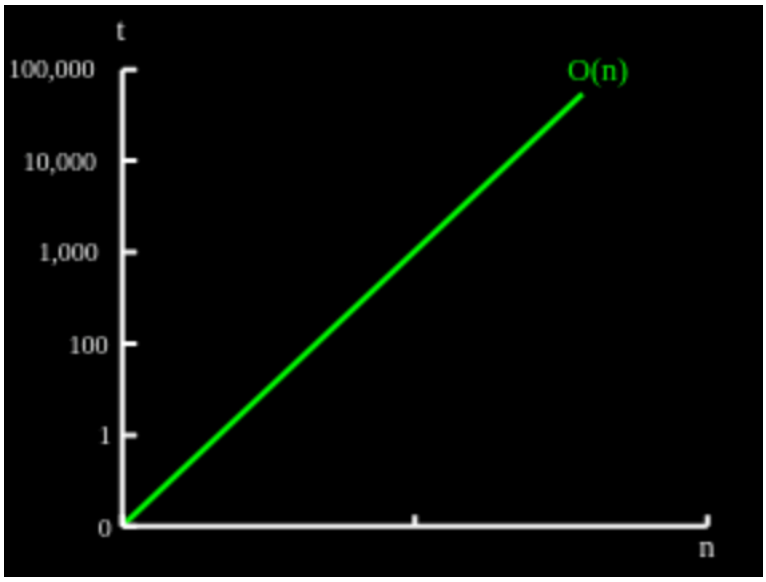
In the best case, suppose the array is already sorted. Although scanning is performed to find the minimum element in each iteration, the minimum element found each time will already be at the beginning of the array. Therefore, scans to find the minimum element in each iteration will be of no use and only displacement will be performed. In this case, the optimal time complexity of the Selection Sort will still be (n^2).

In the average and worst cases, each element of the array is scanned and selected as the minimum element and the displacement operation is performed. In this case, the average and worst-case time complexity of the Selection Sort will be (n^2).

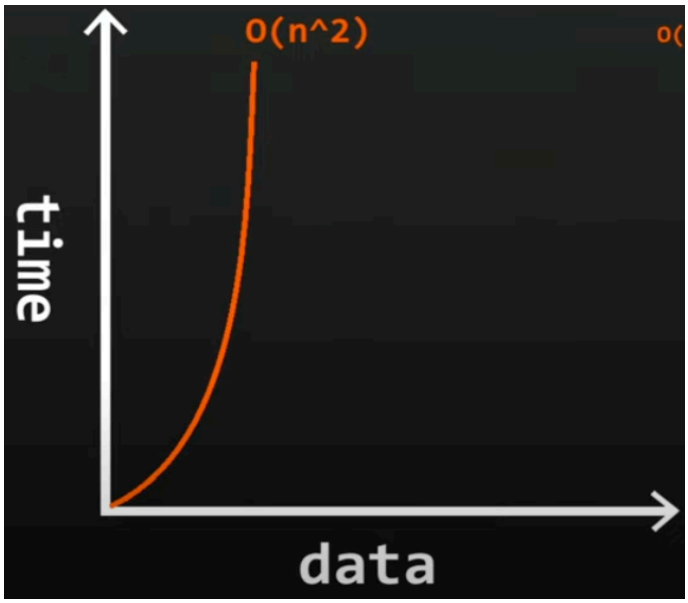
As a result, the best, average, and worst-case time complexity of the Selection Sort is the same, because the algorithm does the same number of operations in all cases, and the scans are of no benefit other than to find the minimum element.

3. Insertion Sort

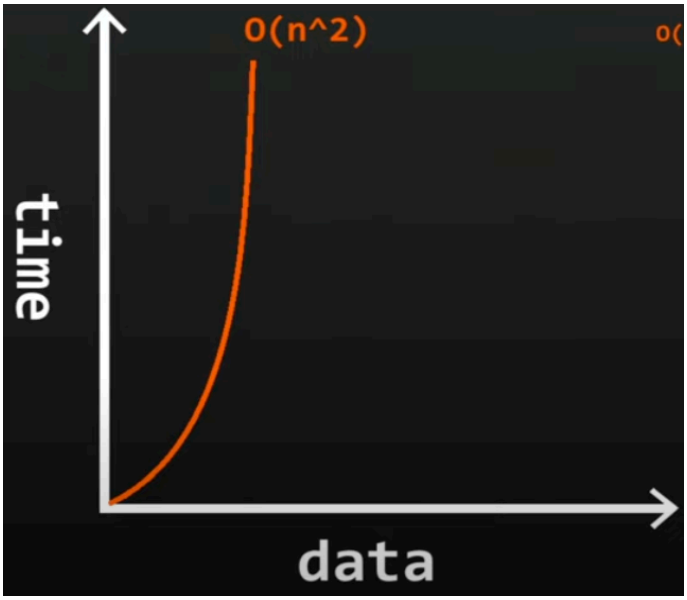
Best Case: O(n)



Avarage Case: O(n^2)



Worst Case: O(n^2)



Below is a picture of the part of the code that makes the difference in Time complexity.

```
while (j >= 0 && originalMap.getMap().get(aux[j]).getSize() >
    originalMap.getMap().get(temp).getSize())
```

If the code is already sequential this while loop won't work and so in the best case time complexity will remain O(n).

Insertion Sort performs the sorting process by traversing the elements of the array to be sorted one by one, placing each element in the appropriate position.

Best Condition: In the best case, suppose the array is already sorted. In this case, only one comparison is made to place each element, and each time the element is placed in its correct position. Therefore, only one comparison is made for each element and no replacement is required. In this case, the optimal time complexity of the Insertion Sort is O(n).

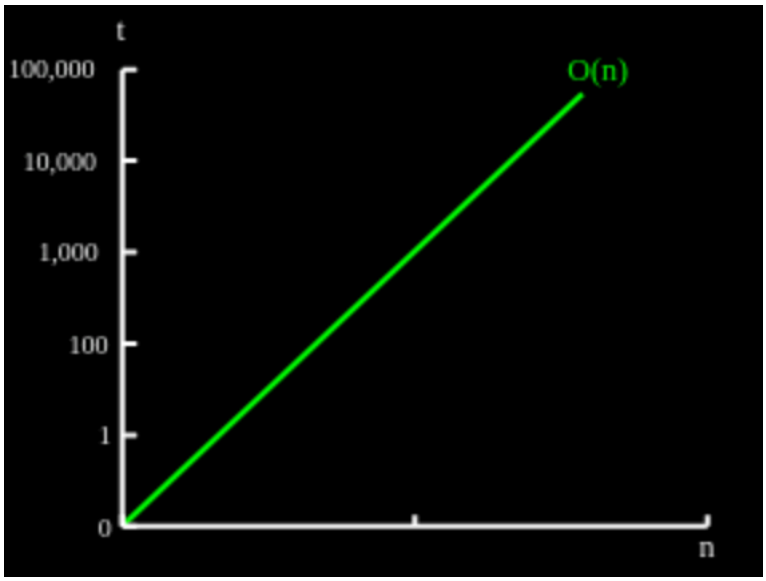
Average Situation: In the average case, the elements of the array are randomly ordered or mixed. To place each element in the correct position, on average, up to half the element in the order so far is compared and repositioned as needed. In the average case, the Insertion Sort has a time complexity of O(n^2).

Worst: In the worst case, suppose the array is in reverse order. In this case, it is necessary to compare all previous elements to place each element in the correct position. That is, n-1 comparisons are made for each element and displacement is performed when necessary. In this case, the worst-case time complexity of the Insertion Sort is O(n^2).

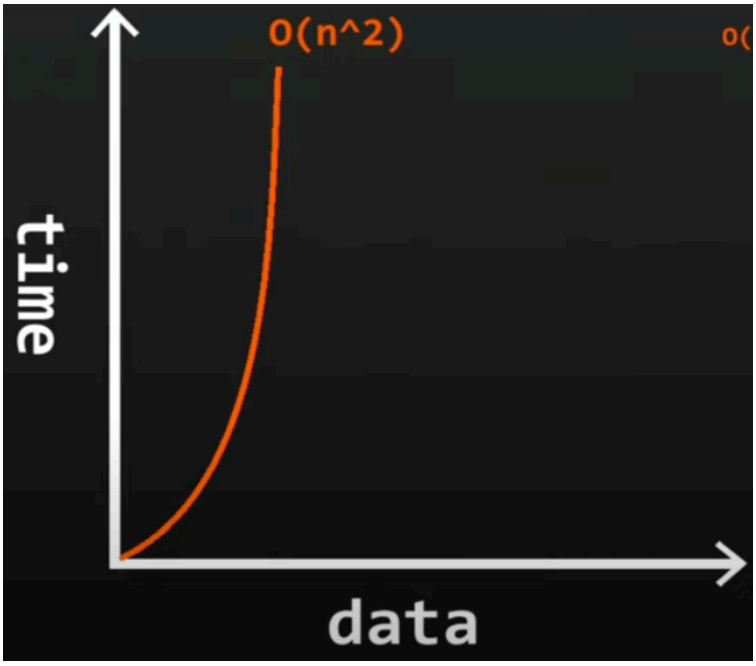
As a result, the best-case time complexity of the Insertion Sort differs from O(n), its average and worst-case time complexity of O(n^2). This difference depends on the ordering of the elements and the comparisons made for the nesting operations.

4. Bubble Sort

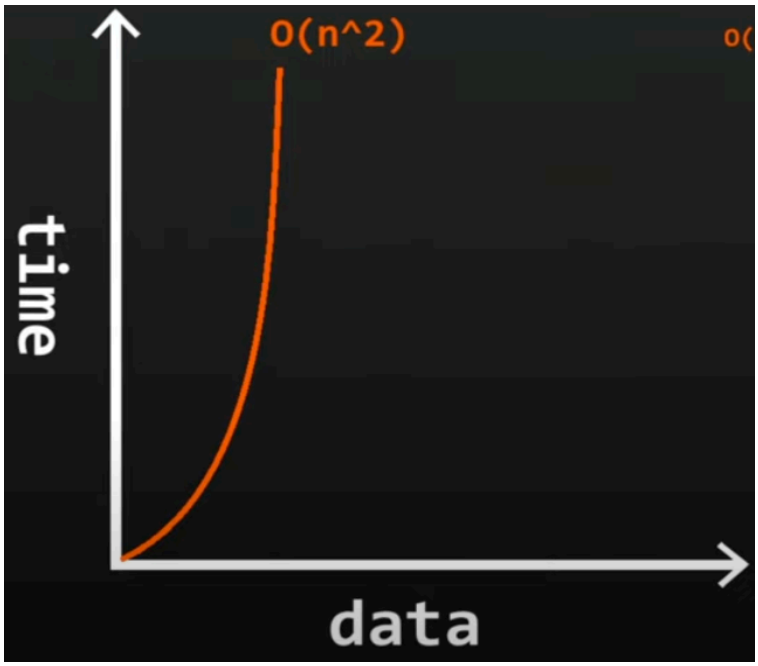
Best Case: O(n)



Avarage Case: O(n^2)



Worst Case: O(n^2)




```
for (int i = 0; i < (originalMap.getMapSize() - 1); i++)
{
    sorted = false;
    for (int j = 0; j < originalMap.getMapSize() - 1 - i; j++)
    {
        if (originalMap.getMap().get(aux[j]).getSize() >
            originalMap.getMap().get(aux[j + 1]).getSize())
        {
            temp = aux[j];
            aux[j] = aux[j + 1];
            aux[j + 1] = temp;
            sorted = true;
        }
    }
    if (sorted == false)
    {
        return;
    }
}
```

The best case for the bubble sort algorithm will occur if the array is sorted, and if the array is already sorted, the best case time complexity will be O(n) since the variable sorted will return false from the second for loop.

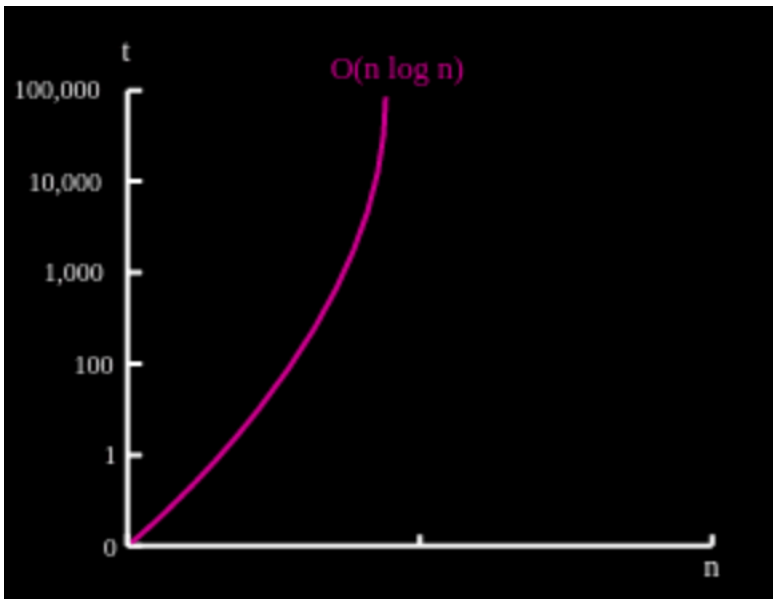
The first for loop represents each step of the sorting process. The second for loop compares the elements of the array and swaps them. Thanks to nested for loops, all elements are compared with adjacent elements and sorting is performed.

In the best case, suppose the array is already sorted. No displacements will be required in each iteration, because adjacent elements are already in the correct order. In this case, the best-case time complexity of Bubble Sort will be O(n).

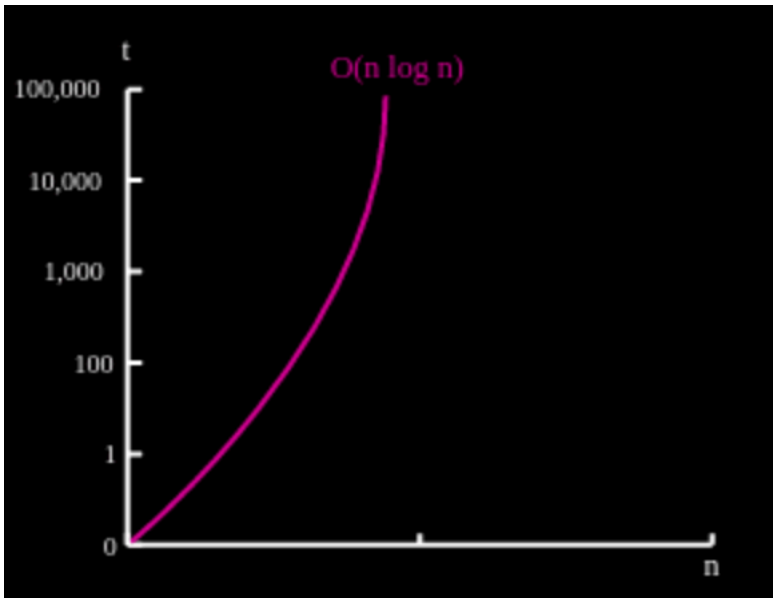
On average and worst cases, each element has to run from the beginning of the array to the end of the array at worst. In each iteration, n elements are processed and n-1 comparisons are performed. In this case, the average and worst-case time complexity of the Bubble Sort will be O(n^2).

5. Quick Sort

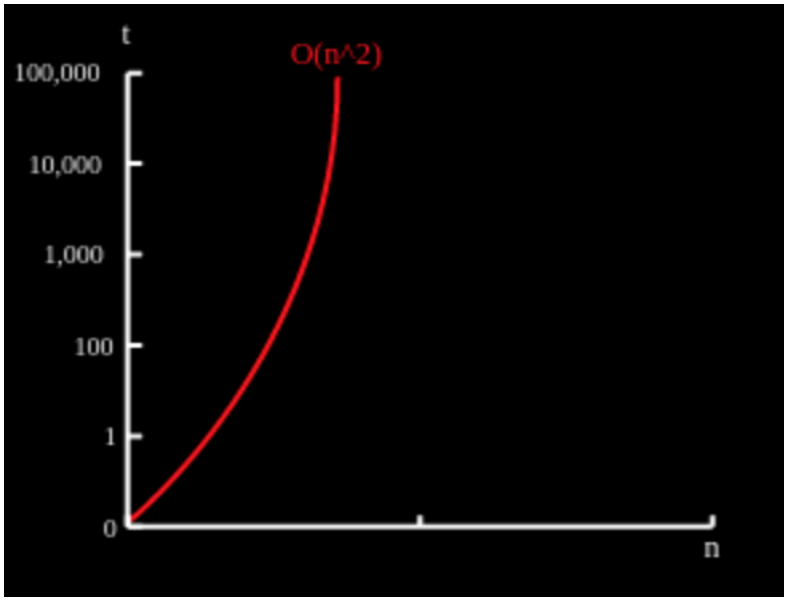
Best Case: O(n(log n))



Avarage Case: O(n(log n))



Worst Case: O(n^2)



Quick Sort is a divide and conquer algorithm and is commonly known as a quick sort algorithm. The algorithm selects an element (pivot) and divides the array into two parts smaller and larger than that element. Then it repeats the sorting process, sorting both parts separately.

Best Condition: In the best case, the time complexity of Quick Sort is O(n log n). This happens when the pivot element is in the middle of the array every time. Thus, each split operation splits the array into almost two equal parts. In this case, log n levels of division are performed at each level, and each division is performed on n elements. As a result, the time complexity is O(n log n).

Average Situation: In the average case, the time complexity of Quick Sort is still O(n log n). This happens when the pivot element is chosen randomly or the median element is used. In the average case, log n levels of division are performed at each level, and each division is performed on n elements. Therefore, the time complexity is calculated as O(n log n).

Worst: In the worst case, the time complexity of Quick Sort is O(n^2). This happens when the pivot element is always selected as the smallest or largest element of the array. In this case, division is performed with only one element at each level, and operations are performed on n elements in each division. In this case, since the division operations are not sufficiently balanced, the sorting process slows down and the time complexity becomes O(n^2).

In the best and average cases, Quick Sort works efficiently and its time complexity is O(n log n). However, in the worst case, the algorithm performance drops significantly and the time complexity rises to O(n^2). This is due to irregularities in the selection of the pivot element. Therefore, the efficiency of Quick Sort is closely related to the choice of the pivot element, and the choice of the pivot element is important.

3. Running Time

Inputs:

Worst case: "abc cbc ctt ttt"
Avarage case: "acb cbc ctt ttt"
Best case: "ttc cbc ttt acb"

1. Merge Sort

Worst case: 94373 ns
Avarage case: 90333 ns
Best case: 87000 ns

2. Selection Sort

Worst case: 47479 ns
Avarage case: 36459 ns
Best case: 283638 ns

3. Insertion Sort

Worst case: 39417 ns
Avarage case: 31917 ns
Best case: 24959 ns

4. Bubble Sort

Worst case: 42142 ns
Avarage case: 32000 ns
Best case: 25859 ns

5. Quick Sort

Worst case: 85163 ns
Avarage case: 41961 ns
Best case: 39235 ns

4. Comparison of Sorting Algorithm

Bubble Sort is simple to understand and implement, but it has poor performance for large data sets due to its quadratic time complexity. It is a costly sorting algorithm when sorting unordered arrays, as it will have to make a lot of changes within the array.

Selection Sort is also simple, but it performs better than Bubble Sort in terms of the number of swaps. However, it still has a quadratic time complexity, making it inefficient for large data sets.

Insertion Sort is efficient for small data sets or partially sorted arrays. It works by iterating through the array and inserting each element into its correct position. It performs better than Bubble Sort and Selection Sort for small inputs.

Merge Sort is a divide-and-conquer algorithm that consistently has a time complexity of O(n log n) regardless of the input data. It divides the array into smaller subarrays, sorts them, and then merges them to obtain the final sorted array. Merge Sort is efficient and stable but requires additional memory space for the merging process.

Quick Sort is a widely used sorting algorithm known for its average-case performance. It works by selecting a pivot element, partitioning the array, and recursively sorting the subarrays. Quick Sort's worst-case time complexity can be mitigated by using randomized pivot selection or other techniques, making it efficient in practice.

Merge sort and quick sort algorithms are better than other algorithms in terms of time complexity, but since they contain recursive methods, the working time of the methods took longer than other sorting algorithms as seen above.

When there is already a sorted array, insertion sort and bubble sort work very fast because their algorithms are very good for this situation.

When we measure by giving larger data instead of the input data I have given, we will actually get more appropriate data. When we enter larger data, we will see the fastest algorithm as quick sort, and then we see the merge sort algorithm. These two algorithms cannot fully show their performance with less data because they work recursive and recursive methods are at a disadvantage in terms of time because they call a method constantly, but the situation changes when larger data are entered and these two algorithms are the fastest algorithms. Among the other 3 sorting algorithms, insertion sort stands out in big data in terms of speed.

In summary, Merge Sort and Quick Sort are generally preferred for their average-case time complexity of O(n log n), which makes them efficient for large data sets. Insertion Sort can be useful for small or partially sorted arrays. Bubble Sort and Selection Sort are simple to understand but have poor performance compared to the others, especially for large data sets. The choice of sorting algorithm depends on the specific requirements and characteristics of the input data.

5. The Special Question

The logic of the merge sort selection sort and bubble sort algorithms is established to look at the elements in the adjacent indexes and replace the two if there is a mistake in their places. Therefore, if the values of the two indexes are equal, whichever comes first in the normal array stays ahead because there is no error in their place.

But since this is not the case in quick sort and selection sort, the order of indexes with the same value may be distorted. In these two algorithms, comparison operations are made not only between two adjacent elements, but also between elements that are far from each other.

For example, let's have a series and we will sort it with the selection sort algorithm. Since all the elements in the array will not be compared side by side and the place will not be changed, the order of the elements with the same size may not be the same as the order in the normal array, and I explained this with a visual below.