Duke University

ECE Department

ECE 590 Algorithms

Project1: Sorting

Presented By

Zhihang Shen, zs149

Kaijian Huang, kh492

Professor: Jian Pei

Fall 2022

**Abstract**

In this project, we designed and implemented five sorting algorithms to obtain an array of ascending order. Here, we will have a through discussion on the performances of different algorithms on both sorted and unsorted test cases. Also, we will show you the plots of runtime versus input size.

# Introduction and Background

## Selection Sort

The **selection sort** algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

* The subarray which already sorted.
* The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

**Time Complexity**: The time complexity of Selection Sort is as there are two nested loops:

* One loop to select an element of Array one by one =
* Another loop to compare that element with every other Array element =

Therefore, overall complexity, [1]

## Insertion Sort

**Insertion sort** is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

**Time Complexity:** [2]

## Bubble Sort

**Bubble Sort** is the simplest [sorting algorithm](https://www.geeksforgeeks.org/sorting-algorithms/) that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

**Time Complexity**:  [3]

## Merge Sort

The **Merge Sort** algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

**Time Complexity:**,  Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation:

[4]

## Quick Sort

Like [Merge Sort](https://www.geeksforgeeks.org/merge-sort/), **QuickSort**is a[Divide and Conquer algorithm](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/). It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

* Always pick the first element as a pivot.
* Always pick the last element as a pivot (implemented below)
* Pick a random element as a pivot.
* Pick median as the pivot.

The key process in **quickSort**is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

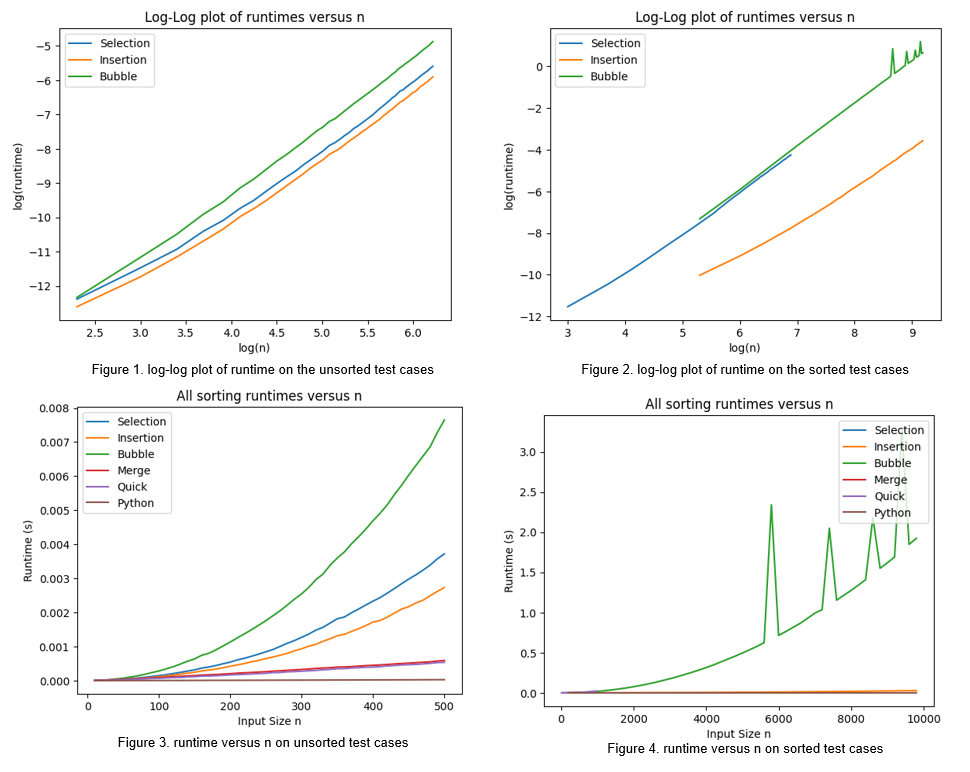
For average and best case, according to . For worst case, pivot is the smallest or the largest one of all the elements in every partition, the time complexity is .[5]

# Discussion

From part one, we can list a table to show the time complexity of each algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| Sorting Algorithm | Worst Case | Average Case | Best Case |
| Selection Sort |  |  |  |
| Insertion Sort |  |  |  |
| Bubble Sort |  |  |  |
| Merge Sort |  |  |  |
| Quick Sort |  |  |  |

Also, we got some output plots from the code.



From the figures above, we can answer following questions.

• Do your algorithms behave as expected for both unsorted and sorted input arrays?

Answer:

Yes, the algorithms behave as expected, we derived form the slopes of the fitted lines.

• Which sorting algorithm was the best (in your opinion)? Which was the worst? Why do you think that is?

Answer:

Quick Sort algorithm was the best in my opinion, and bubble sort works the worst. It is related to the time complexity that quick sort works better.

• Why do we report theoretical runtimes for asymptotically large values of n?

Answer:

when the input size is large, extreme cases would have smaller influence on the runtime, so the slope becomes more stable and closer to the theoretical value. Therefore, asymptotically large values of *n* make better sense and should be taken into consideration.

• What happens to the runtime for smaller values of n? Why do you think this is?

Answer:

when n is small, the computer finishes the task in a really short time. If the computer finishes a task in 100ns versus 50ns, then the difference doesn’t matter.

• Why do we average the runtime across multiple trials? What happens if you use only one trial?

Answer:

it is meaningful and significant to average the runtime across multiple trials, rather than use only one trial. The reason is that some algorithms can have best cases and worst cases other than averages cases. If we only use one trial and it happens to be a best or worst case, the experimental runtime would be more likely to deviate from the theoretical value. We can largely avoid this drawback by taking a number of measurements and averaging them.

• What happens if you time your code while performing a computationally expensive task in the background (i.e., opening an internet browser during execution)?

Answer:

The computing power will be assigned to other tasks running in the background. The runtime of our code will also account for the programs that run in the background. Therefore, it is very likely that we will get some inaccurate result, and it does not really reflect the runtime of our code.

• Why do we analyze theoretical runtimes for algorithms instead of implementing them and reporting actual/experimental runtimes? Are there times when theoretical runtimes provide more useful comparisons? Are their times when experimental runtimes provide more useful comparisons?

Answer:

In most cases, we analyze the theoretical runtime instead of the actual runtime, when n is large or when the runtime increases very fast, it’s impossible for us to get the actual time. And Big-O notation abstracts away unimportant distinctions caused by factors such as processors and goes to the key differences between algorithms. when n is small, the computer finishes the task in a really short time. If the computer finishes a task in 100ns versus 50ns, then the difference doesn’t matter.

# References

[1] https://www.geeksforgeeks.org/selection-sort/

[2] https://www.geeksforgeeks.org/insertion-sort/

[3] https://www.geeksforgeeks.org/bubble-sort/

[4] <https://www.geeksforgeeks.org/merge-sort/>

[5] https://www.geeksforgeeks.org/quick-sort/