# Project 2 report

Comparison of sorting methods

20B11806
Saw Kay Khine Oo

## 1. Purpose of the project

The main purpose of the project is to compare and analyze a few sorting methods available. Thus, in this report, I will focus on the analysis of the resulting graphs. Even though I will go through the algorithms, they may not be explained in details.

## 2. Basic Functions

We will need three main functions in the following sorting algorithms: creating random array, swapping and checking if the array is sorted.

### 2.1 Creating random array

We use the sample from random to produce a random array of given size.

### 2.2 Swapping

For swapping, we will introduce a new variable tmp to store the value temporarily so that we can easily swap the value at two indexes.

### 2.3 is_sorted function

In this project, we want the array to be sorted in ascending order, so we check if the i-1 value of the array is less than i value of the array. If that is true for all the elements in the array, then the function will return True.

## 3. Sorting Methods

### 3.1 Insertion Sort

There are two loops in insertion sort. The outer loop is for going through every element in the array. The inner one loops backwards and it is for comparing the elements and swapping if necessary. Comparing and swapping will continue until the element finds its right place. Since the elements will already be sorted in the inner loop, when the first comparison fails, we do not need to compare with the rest and the loop can break.

### 3.2 Quick Sort

Since we are going to use a recursive function, we will build the base case first. If there is only one element in the array, we can just return the array. If there is 2, we can just do one comparison. However, if there is more than 2, we will choose a random pivot, in this project, it will be the first element in each recursion. Then, we can have 2 arrays split by the pivot, either smaller or larger than the pivot. These two new arrays will be sorted by the quick sort method by calling recursion.

### 3.3 Merge Sort

Merge sort will be mainly divided into two recursive parts. The first recursion is for splitting into two parts from the middle and the second is for merging the two split parts. For the first part, we will set the left section from 0 to middle and right from middle to last element and use recursion to keep splitting until there is only 1 or 2 elements left (the base

case). For the second part, we can just concatenate two arrays in the base case when both arrays are empty or when the last element on the left array is smaller than the first element of the right array (assuming we want an ascending order). Other than that, we need to use recursion by comparing the first element of both arrays, take out the smaller one and keep recurring with the left array (without the one already taken out) and the right array (same).

### 3.4 Bubble Sort

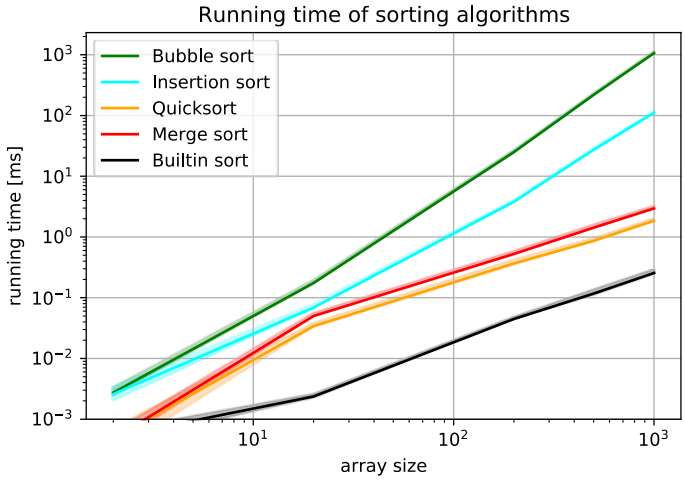
This algorithm is very similar to insertion sort, having two loops, one for all the elements in the array and the other for comparing and swapping. The outer loop will loop from length of the array to 0. The inner loop will have to go through all the unsorted elements and compare to find the most favorable one (either smallest or largest). It can't stop the comparison half way because unless it is sorted there is no way to know if the current element is the most suitable one. However, if the remaining elements from the outer loop already contribute to a sorted array, then there will be no need for the comparison.

## 4. Comparisons of different sorting methods

For the graph, I used 50 repetitions to get better result since the arrays will be produced randomly. Furthermore, since some algorithms will have similar running time result for small test size, we need to analyze the result with fairly big test size as well. So, I chose the biggest as 500. I could have chosen bigger test size, but it would take so long for the bubble sort.

The graph below shows the running time of 4 different sorting algorithms explained above in Section 3 plus the built-in sorting method in Python. Obviously, the built-in sort in python is the most efficient one given that running time is always the lowest. However, there are some interesting points to note about the other 4 sorting algorithms.

As we can see in the graph here, time taken for insertion sort and bubble sort (both $O(n^2)$ swaps in average scenario) has little difference and the same applies to merge sort and quick sort (both O(nlogn) swaps in average scenario). So, these similar methods will be compared first and then insertion sort and quick sort will be compared and explained shortly.



### 4.1 Merge Sort vs. Quick Sort Comparison

Merge sort and quick sort are quite similar, but the efficiency of quick sort prominently depends on the first and subsequent choices of pivots. On the other hand, merge sort's efficiency is only dependent on the initial permutation as it doesn't include random choice of pivots. When the number of array size is small, there is a high chance that quick sort will have the optimal split, hence results in almost the same efficiency as merge sort. Thus, theoretically, quick sort might seem to be more efficient for a small array size, but for really large arrays, it would be better to use merge sort. This is what I initially thought. Nevertheless, I could not get a graph of merge sort being more efficient than quick sort despite trying different combinations.

I searched for a while and found out why quick sort is generally considered to be better than merge sort. Merge sort has to store the sorted left and right array before merging. So, quick sort actually wins because of its lower consumption in memory. This plays a big factor as well when the array size gets bigger.

### 4.2 Insertion Sort vs. Bubble Sort Comparison

Insertion sort and bubble sort are similar in a way or so. They extend the sorted part one element at a time. In the bubble sort, the iterations needed for comparison will get shorter and shorter as the elements are getting sorted (the first iteration is the longest). Optimal situation is when the remaining list is already sorted. However, this situation will only likely to occur near the end (for short iterations). So, number of comparisons is roughly equal to n^2/2.

In insertion sort, iterations gradually become longer (the first iteration is the shortest). However, if the element can find its correct place faster, then the efficiency increases. This usually happens in the middle of iteration number. So, number of comparisons is approximately n^2/4. We can say that the optimal situation for insertion sort occurs at a higher possibility than that for bubble sort. Thus, insertion sort is more efficient.

### 4.3 Insertion Sort vs. Quick Sort Comparison

Insertion sort is a $O(n^2)$ time complexity algorithm while quick sort is a O(nlogn) time complexity algorithm. Thus, even if the running time of the two algorithms might be close for small n, as n gets bigger and bigger, the running time gap between these two algorithms will only grow larger. This can be seen in the graph.

## 5. Conclusion

Through this project, I learnt different kinds of sorting algorithms and for the first time actually paid attention to the time complexity of each algorithm. Since bubble sort and insertion sort take the longest, when I run and test the main program, I would sometimes take out those two to compare merge sort and quick sort only. Although I could not include all the available sorting algorithms in this project, I am looking forward to learning new sorting algorithm as well as checking out the python built-in sort, which I think is amazing given its efficiency.