Student: Tran Ngoc Son (陳玉山)
ID: 0710185

HOMEWORK 1 REPORT

Contents

I. Introduction **2**
II. Methology **2**
III. Results **5**
IV. Discussion **7**

Student: Tran Ngoc Son (陳玉山)
ID: 0710185

I. Introduction

In this homework, we will implement the two very famous sorts, which are insertion sort and merge sort. While insertion sort will sort every element from the beginning to the end of the array, merge sort will divide the array into multiple sub-array to decrease the size of the problem by using the recursive method. In addition, we will try to figure out the executing time and discuss the big O notation of each function and conclude which one is faster in different input array sizes.

II. Methodology

1. Insertion sort

The idea of this sorting method is to take each element, starting from the second element to the end, of the array to compare it with the previous values of that array. Take index 2 for example, we will compare the value a[2] with a[1] and a[0], if a[2] is smaller then we do the swap (inserting) process, or we keep staying the same place if a[2] is equal or larger.

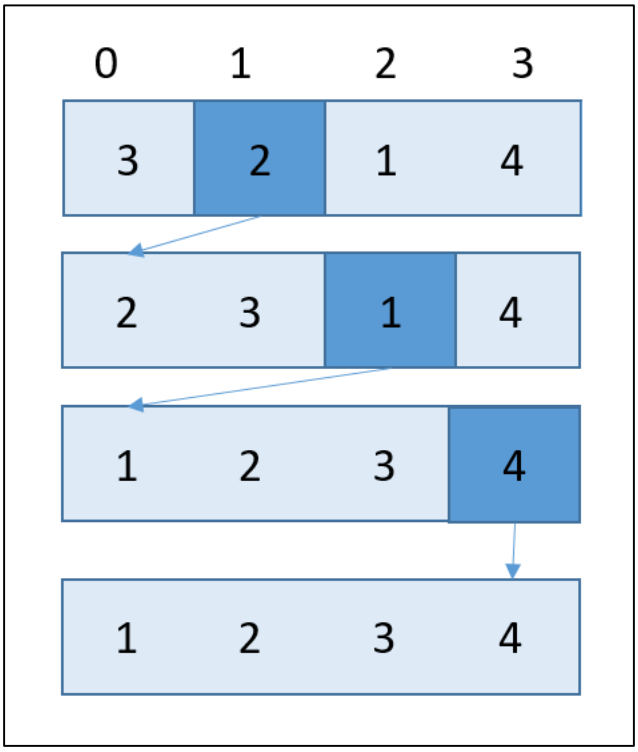


Figure 1: Insertion sort process

To implement this insertion sort, we should have two for loops, the outer for loop is used for taking each element as a key value for comparing, and the inner loop is to go through every index that stands before the one we take as a key index to do the comparing process. This process is kept executing repeatedly until all the elements are sorted.

```
INSERTION-SORT (A)
1 for j <- 2 to length[A]
2 do key <- A[j]
3 Insert A[j] into the sorted sequence A[1 . . j - 1].
4 i <- j - 1
5 while i > 0 and A[i] > key
6 do A[i + 1] <- A[i]
7 i <- i - 1
8 A[i + 1] <- key
```

Figure 2: Insertion-sort pseudo code

2. Merge sort

The main idea of this sort is to divide the array into many sub-arrays, it is also known as the divide and conquer method, usually, it is divided by 2. Until the sub-array has 1 element only, we do the merge process to merge these sub-arrays into a full array. In the merge process, we also need to make sure that the part after merging is sorted already. This process keeps repeating until we achieve the full sorted array.

In general, we can observe what happens when we call the merge function from the figure below. The red number is noted for the procedure of processing inside the computer.

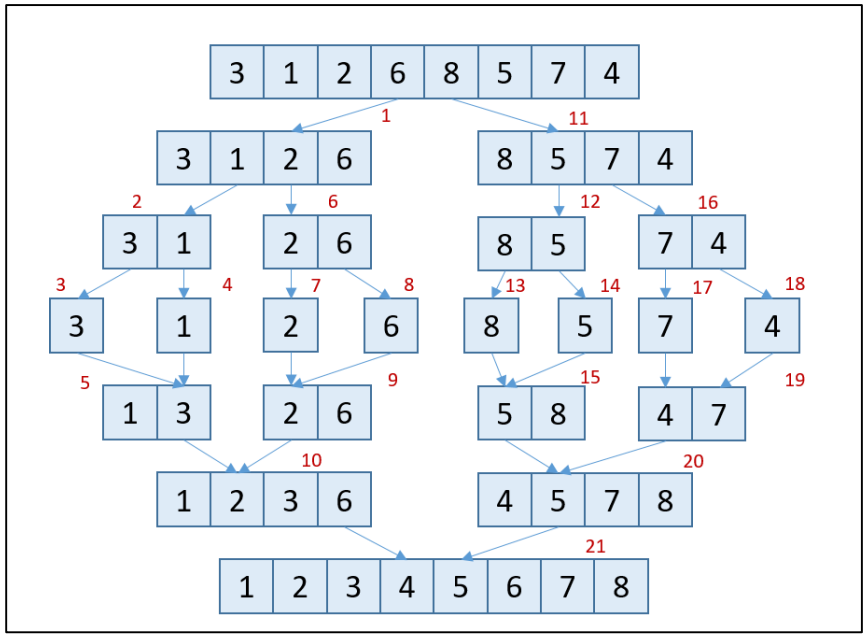


Figure 3: Merge sort procedure

Student: Tran Ngoc Son (陳玉山)
ID: 0710185

Merge function

The main purpose of the merge function is to merge the sub-array into the larger array and finally achieve a fully sorted array. To do it, we will use two variables (i, j) to control the index of two arrays then compare each element in the two sub-arrays, the smaller one would be chosen for putting in the sorted array and then increase the index of that array by one. The process is repeated until (i, j) are out of range of the two sub-arrays, then we will get the sorted array.

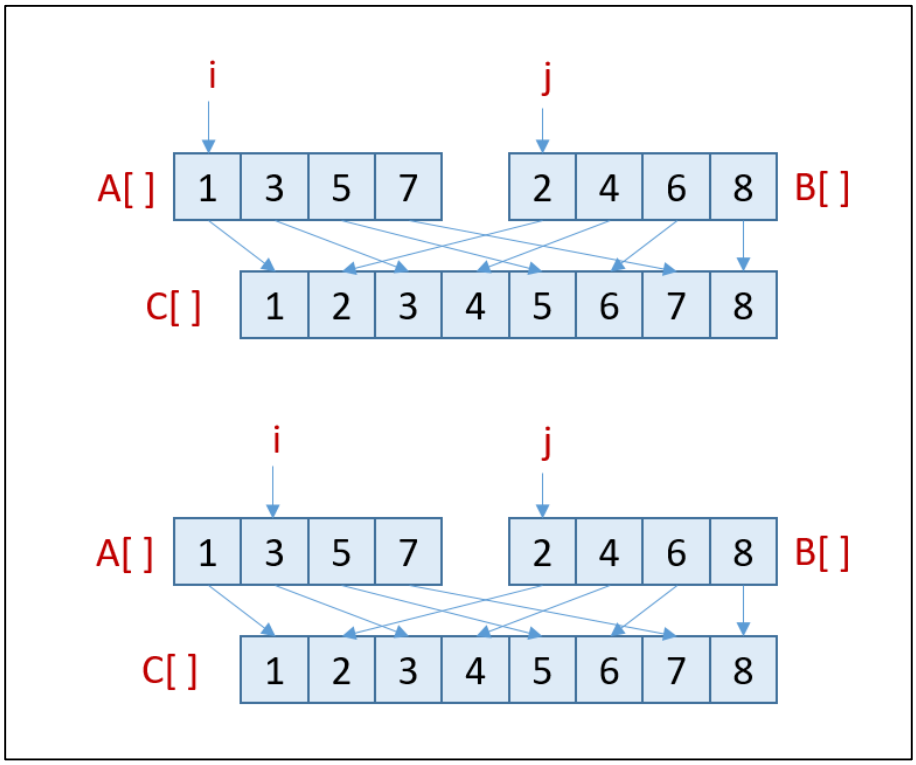


Figure 4: Merge function procedure

i, j initially are equal to 0, we compare A[i] and B[j], the smaller one is picked to be an element of array C, then increase the recently picked index by 1 and keep repeating the process.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Figure 5: Merge sort pseudo code (recursive part)

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 . . n_1 + 1] and R[1 . . n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Figure 6: Merge sort pseudo code (merge part)

III. Results

a. Big O discussions

Insertion sort

In the insertion sort algorithm, two loops need to be implemented; therefore, in the worst case, its big O should be $O(n^2)$.

Merge sort

With the method of dividing the array into multiple sub-arrays, we can know the height of that recursive tree by using $\log_2 n$ function. Furthermore, each layer will executive with the time of O(n); hence, this algorithm would have $O(n\log_2 n)$.

b. Executing time

Input array size	Insertion sort (microseconds)	Merge sort (microseconds)
50	0	997
100	231	497
650	2035	2003
1.000	4517	1002
10.000	220 625	5652
100.000	23 339 462	45 552

Table 1: Executing time comparison

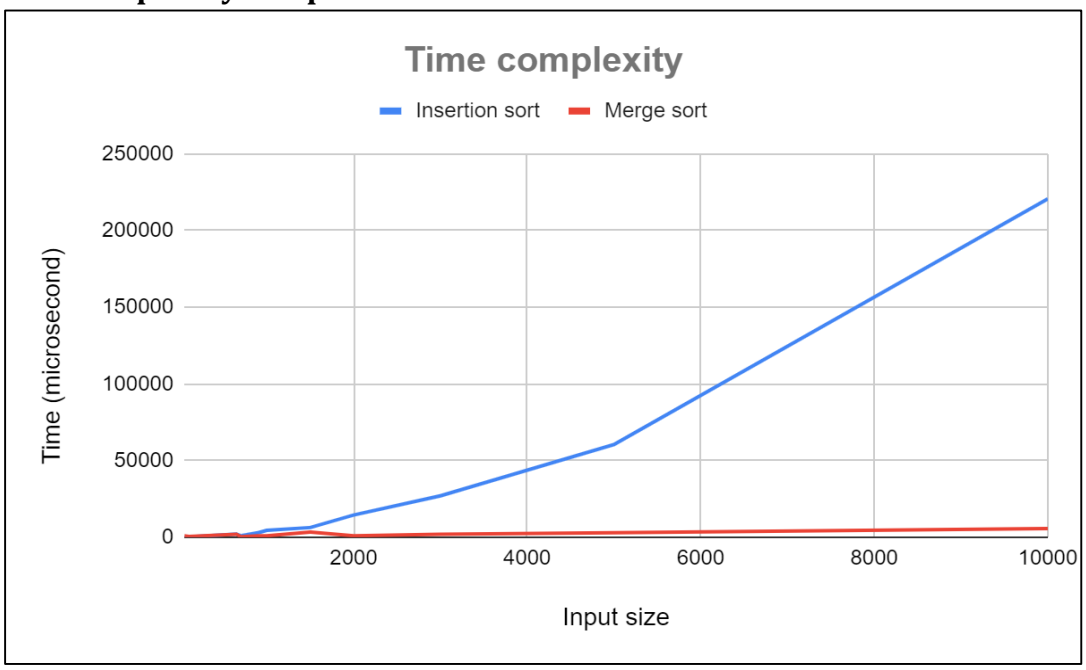
As we can observe from table 1, the Insertion sort performs pretty well in terms of executing time when the input array size is still small, especially less than 650. The executing time of insertion sort in with a small input size is less than merge sort in many cases. The differences can be seen since we increase the input size to 1000, insertion sort is about the fourth time slower than merge sort. There is a huge difference between the two executing times when we set the input size to 100.000, the data shows that merge sort sends out a much better performance in this case, 45.552 microseconds compared to 23.339.462 microseconds of the insertion sort function.

Note: Because of the random property when creating an input array and the differences in performance of the computer at different times, therefore the above table cannot tell the exact executing time when we input the same size. However, it shows clearly that as input size n becomes significantly large, merge sort performs much better.

c. Big O verification

Let's take an input size of 100,000 for example. We assume that the insertion sort has to deal with the worst-case and has $O(n^2)$ with executing time of 2.339.462 microseconds. Therefore, n would be roughly equal to 4831 microseconds after n^2 is taken square root. We then apply this number to $O(n\log_2 n)$ of the merge function then we get $n\log_2 n$ = 59 122,2 microseconds compared to 45 552 microseconds from the data in table 1, and this value of time is quite reasonable since $n\log_2 n$ is an upper bound for merge sort function.

d. Time complexity comparison



Student: Tran Ngoc Son (陳玉山)
ID: 0710185

IV. Discussion

At first, I was not clear about why we needed to use different sorting methods with sending out the same result. However, when implementing and comparing the executing time of these sorting algorithms. I found out that each of them has its advantages and disadvantages when dealing with different input sizes. For instance, insertion sort works noticeably well when the input size is relatively small, even faster than merge sort. In contrast, for the significantly large input size, it takes a lot of time for the insertion sort to have work done but the merge sort can deal with it easily. Finally, this homework helps me to have a different perspective on the sorting algorithms and figure out the pros and cons of the two algorithms.