Sorting Algorithm

Chinchuthakun Worameth

January 31, 2019

1 Introduction

This project compares the actual running time of well-known sorting algorithms of both types, comparison sort and non-comparison sort, under the same environment; including bubble sort, insertion sort, selection sort, merge sort, quick sort, and radix sort. The mathematical analysis is also included in the report.

2 Comparison Sorting Algorithm

This section will explain the principles behind each comparison sorting algorithms in the project including possible optimizations.

2.1 Comparison sort

A comparison sort is a class of sorting algorithm relying on a **comparison operator** to determines the order of every pair of elements. The valid operator must satisfy the following properties:

1. if $a \leq b$ and $b \leq c$, then $a \leq c$
2. For every pair (a, b), either $a \leq b$ or $b \leq a$ must hold.

In case $a \leq b$ and $b \leq a$, the order is not relevant. However, in **stable sort**, the order of a and b in the sorted sequence is determined by their order in the original sequence. Note that sort algorithms that are not in this class are denoted as **non-comparison sorts**.

2.2 Bubble Sort

Let $(a_1, a_2, ..., a_n)$ be a sequence of random integers of length n and $i \in [1, n-1]$ be an index such that $a_i > a_{i+1}$. The bubble sort repeatedly swap a_i and a_{i+1}

until i does not exist. The pseudo-code of the algorithm is shown below.

Algorithm 1 Bubble Sort

```
1: function SORT(A = (a_1, ..., a_n))
2:     for i = 1, ..., n do
3:         for j = i, ..., n − 1 do
4:             if a_j > a_{j+1} then
5:                 swap(a_j, a_{j+1})
6:             end if
7:         end for
8:     end for
9: end function
```

From the pseudo-code, it is obvious that the time complexity of the bubble sort is $\mathrm{O}(n^2)$. Note that in the best case, the input is already sorted, the time complexity is reduced to $\mathrm{O}(n)$.

2.3 Insertion Sort

Let $(a_1, a_2, ..., a_n)$ be a sequence of random integers of length n. For every i in interval $[1, n]$, insert the integer a_i at the end of non-decreasing subsequence of length $i-1$, $(a_1, a_2, \ldots, a_{i-1})$. Then, repeatedly swap the integer to its correct position to obtain non-decreasing subsequence of length i, $(a_1, a_2, \ldots, a_i)$. The algorithm proceeds $i = 1, 2, ..., n$ chronologically. The pseudo-code of the algorithm is shown below.

Algorithm 2 Insertion Sort

```
1: function SORT(A = (a_1, ..., a_n))
2:     for i = 1, ..., n do
3:         for j = n, ..., 1 do
4:             if a_j < a_{j−1} then
5:                 swap(a_{j−1}, a_j)
6:             end if
7:         end for
8:     end for
9: end function
```

From the pseudo-code, it is obvious that the time complexity of the bubble sort is $\mathrm{O}(n^2)$. Note that in the best case, the input is already sorted, the time complexity is reduced to $\mathrm{O}(n)$.

2.4 Selection Sort

Let $(a_1, a_2, ..., a_n)$ be a sequence of random integers of length n. For every i in interval $[1, n]$, swap the least integer in subsequence of length $n - i$, $a_{i+1}, a_{i+2}, \ldots, a_n$, with a_i. The algorithm proceeds $i = 1, 2, ..., n$ chronologically. The pseudo-code of the algorithm is shown below.

Algorithm 3 Selection Sort

```
1: function SORT(A = (a_1, ..., a_n))
2:     for i = 1, ..., n do
3:         x = index such that a_x is the minimum number among a_i, ..., a_n
4:         swap(a_i, a_x)
5:     end for
6: end function
```

From the pseudo-code, it is obvious that the time complexity of the bubble sort is $\mathrm{O}(n^2)$.

2.5 Merge Sort

Merge sort algorithm is a sorting algorithm that utilize the divide and conquer technique. Let $(a_1, a_2, ..., a_n)$ be a sequence of random integers of length n. For merge sort algorithm, in order to sort the integer in the interval $[l, r]$, the algorithm divides the interval into 2 sub-intervals: $[l, m]$ and $[m + 1, r]$ where $m = \lfloor \frac{l+r}{2} \rfloor$. The merge sort algorithm is then applied to both sub-intervals; creating two sorted sequences. In the end, the algorithm merges two sorted sequences into one sorted sequence. Note that this can be achieved in $\mathrm{O}(n)$. The pseudo-code of the algorithm is shown below.

Algorithm 4 Merge Sort

```
1: function SORT(A = (a_1, ..., a_n))
2:     if length of A is 1 then
3:         return A
4:     else
5:         m = n/2
6:         SORT((a_1, a_2, ..., a_m))
7:         SORT((a_{m+1}, a_{m+2}, ..., a_n))
8:         return MERGE((a_1, a_2, ..., a_m),(a_{m+1}, a_{m+2}, ..., a_n))
9:     end if
10: end function
```

Let $T(n)$ denotes the total time for the algorithm on an input of size n. From the pseudo-code, we have $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$. Since the recurrence relation is in the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, the master theorem is applied to

solve the relation. From $c = log_2 2$, we have $f(n) = O(n) = \Theta(n^c \log^k n)$ for $k = 0$. Therefore, $T(n) = \Theta(n^c \log^{k+1} n) = \Theta(n \log n)$

2.6 Quick Sort

Quick sort is another sorting algorithm that utilize the divide and conquer technique. Let $(a_1, a_2, ..., a_n)$ be a sequence of random integers of length n. The sequence is then separated into 2 sub-sequences $(a_1, a_2, ..., a_{p-1})$ and $(a_{p+1}, a_{p+2}, ..., a_n)$ such that every elements in the first sub-sequence is not greater than a_p while every elements in the second sub-sequence is not less than a_p. Both sub-sequences are then sorted by quick sort; appending together after that. Note that a_p is called pivot.

The pivot can be selected in many ways e.g. the first element, the last element, the median, or even a random element. Random element approach is used in this experiment which has average time complexity of $\mathrm{O}(n \log n)$.

Algorithm 5 Quick Sort

```
1: function SORT(A = (a_1, ..., a_n))
2:     if length of A is 1 then
3:         return A
4:     else
5:         p =PARTITION(1, n)
6:         SORT((a_1, a_2, ..., a_{p-1}))
7:         SORT((a_{p+1}, a_{p+2}, ..., a_n))
8:     end if
9: end function
```

3 Non-comparison Sorting Algorithm

Because of the $\Omega(nlogn)$ lower bound of time complexity of comparison sort, it is interesting to investigate the alternative solution: non-comparison sort. This section will explain the principles behind one of well-known non-comparison sort, radix sort.

3.1 Radix Sort

Radix sort's principle is based on repeatedly grouping **integers** into several groups (usually 10) by using queues based on their digits - from the least significant digit to the most significant digit chronologically. Note that every integers has infinite number of leading zero. The algorithm proceeds $i = 1, 2, ..., n$ chronologically. The pseudo-code of the algorithm is shown below.

Algorithm 6 Radix Sort

```
1: function SORT(A = (a_1, ..., a_n))
2:     for i = 1, ..., d do
3:         q_0, ..., q_b are empty queues. B is an empty list.
4:         for j = 1, ..., n do
5:             Push a_j in the a_j[i]-th queue
6:         end for
7:         for j = 1, ..., b do
8:             for k = 1, ..., q_j do
9:                 Append q_j[k] at the end of list B
10:            end for
11:        end for
12:        A = B
13:    end for
14: end function
```

From the pseudo-code, if the maximum number of digit of input data is d, it is obvious that the time complexity of the radix sort is O(nd). If the value of d is fixed i.e. d is constant, it can be derived that the time complexity of radix sort is O(n).

4 Experiment

4.1 Detail of Experiment

The experiment focuses on 6 sorting algorithms: bubble sort, insertion sort, selection sort, merge sort, quick sort, and radix sort. The test data is a sequence of integers with length n. The test size is (10,100,500,1000,2000,5000) to observe the behavior of the algorithms at both small and large size of input data. Note that each execution of certain algorithm against certain size of input data is repeated 10 times.

4.2 Result and Discussion

The result of the experiments are shown below. There are 3 interesting observations from the graphs:

1. The O(n^2) class algorithms execute faster than the O($nlogn$) class algorithms against small input datas. However, at a certain value of n, the opposite happens; the O(n^2) class algorithms are faster than the O($nlogn$) ones. This comes in to agreement with the calculation that $\lim_{n\to\infty}(an^2 - bn\log n) = \infty$ where $a, b > 0$.

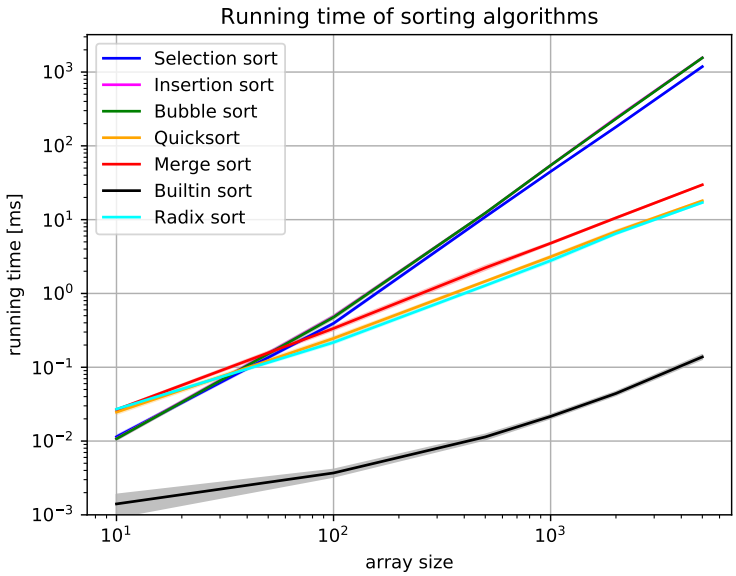

Figure 1: Running time of sorting algorithms against random sequences

2. Poor implementation can affect the running time; making it longer than it should be. For example, compare two attached files: **mergesort.py** and **mergesort_bad.py**. The first code only uses indices in the array as its parameters while the second one actually sends the separated arrays as its arguments. As shown in the Fig 2, the running time of the first one is slightly decreased when compared to the second one.

3. The pivot selection in the quick sort greatly affects the running time of the algorithm. If the test sequence is specifically selected, the running time can also be dramatically different as shown in the Fig 3. In the graph, the quick sort used in Fig. 1 (red) uses random pivot approach while another quick sort (blue) uses the first pivot approach. The test sequences are guaranteed to be non-decreasing sequence. From complexity analysis, we have the time complexity of the red quick sort and blue quick sort against this particular type of test sequence are $\mathrm{O}(n \log n)$ and $\mathrm{O}(n^2)$ respectively.

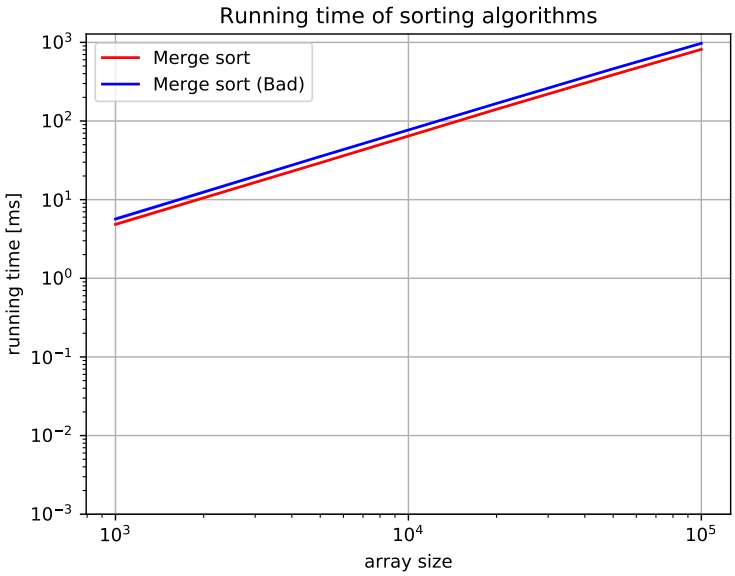

Figure 2: Running time of two merge sort implementations

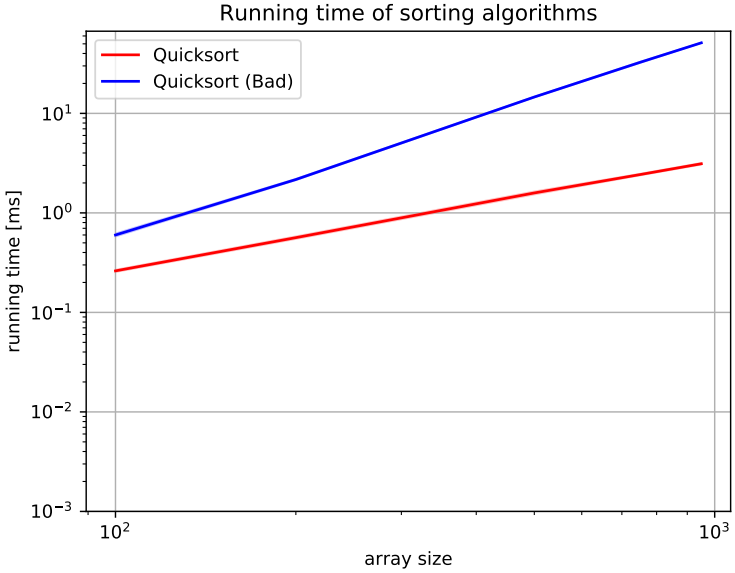

Figure 3: Running time of two quick sort implementation