EE 3980 Algorithms

Homework 1. Quadratic Sorts Report

105061110 周柏宇

2020/3/13

1. Introduction

In this homework, we implemented 4 sorting algorithms: selection sort, insertion sort, bubble sort and shaker sort. To measure the performance, we run each of the algorithm to sort different number of English words in-place and in ascending order for 500 times to calculate the average CPU run time. Finally, we print out the sorted list, name of sorting algorithm, size of inputs and the average execution times.

2. Analysis & Implementation

2.1 Selection Sort

```
Algorithm SelectionSort(A, n)
{
    for i := 1 to n do {
        j := i;                               // initialize j to be i
        for k := i + 1 to n do       // find the smallest in A[i + 1 : n]
            if (A[k] < A[j]) then j := k;    //found, record the index
        t := A[i]; A[i] := A[j]; A[j] := t; // swap A[i] and A[j]
    }
}
```

In selection sort, we first search for the smallest element in A[i+1, n] and switch the smallest element found with A[i]. As i increase, we gradually finish the sort.

The outer loop will execute n times and the inner loop will execute n – i times.

$$\sum_{i=1}^{n} n - i = n^2 - \frac{n(n+1)}{2} = \frac{n^2 - n}{2}$$

Therefore, the selection sort has time complexity $O(n^2)$ and space complexity $O(1)$ since there is no extra memory required relating to the input size.

2.2 Insertion Sort

```
Algorithm InsertionSort(A, n)
{
    for j := 2 to n do {    // assume A[1 : j - 1] already sorted
        item := A[j];       // move A[j] to its proper place
        i := j - 1;         // initialize i to be j - 1
        while ((i >= 1) and (item < A[i])) do {
                                // find i such that A[i] <= A[j]
            A[i + 1] := A[i];   // move A[i] up by one position
            i := i - 1;
        }
        A[i + 1] := item;       // move A[j] to A[i + 1]
    }
}
```

In insertion sort, we start at the second element of A and compare it with the elements to its left (smaller index). If the elements to its left is bigger than A[j], then move it to the right. When the while loop stops, it means we either encountered a element smaller than A[j] or A[j] is the smallest among the elements to its left. We fill A[i + 1] with A[j].

The outer loop executes n – 1 times and the while loop has the worst case of j – 1 times execution.

$$\sum_{j=2}^{n} j - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

which is $O(n^2)$ in time complexity and $O(1)$ in space complexity.

2.3 Bubble Sort

```
Algorithm BubbleSort(A, n)
{
    for i := 1 to n - 1 do {        // find the smallest item for A[i]
        for j := n to i + 1 step -1 do {
            if (A[j] < A[j - 1]) {   // swap A[j] and A[j - 1]
                t := A[j]; A[j] := A[j - 1]; A[j - 1] := t;
            }
        }
    }
}
```

In bubble sort, we keep comparing the contiguous elements starting from the right and move the smaller one to the left. After the inner loop is done, we can make sure the smallest element in A[i : n] has moved to A[i].

The outer loop executes n – 1 times and the inner loop executes n – i times. The calculation is similar to selection sort, which gives the bubble sort a $O(n^2)$ time complexity and $O(1)$ space complexity.

2.4 Shaker Sort

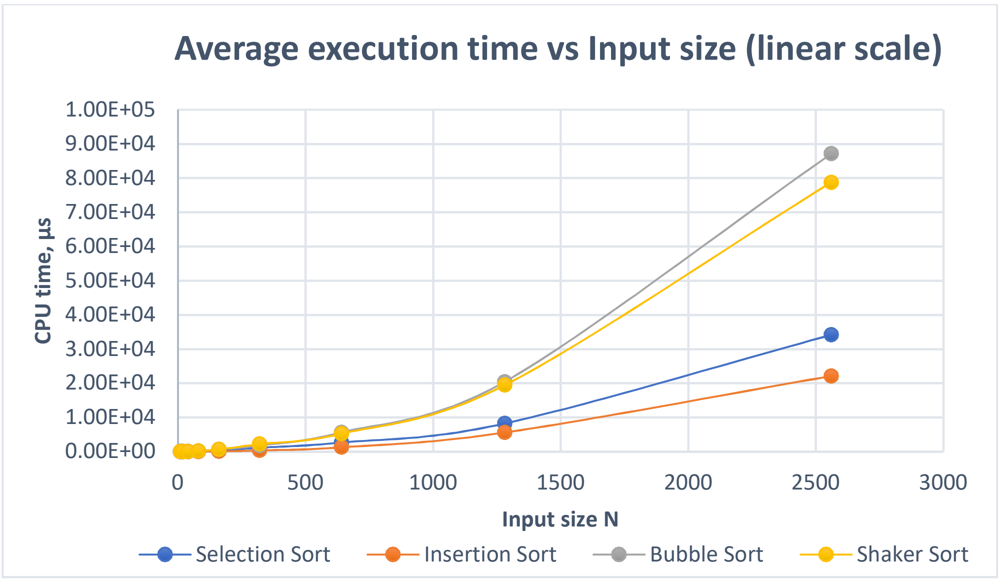
```
Algorithm ShakerSort(A, n)
{
    l := 1; r := n;
    while l <= r do {
        for j := r to l + 1 step -1 do {// element exchange from r to l
            if (A[j] < A[j - 1]) {       // swap A[j] and A[j - 1]
                t := A[j]; A[j] := A[j - 1]; A[j - 1] := t;
            }
        }
        l := l + 1;
        for j := l to r - 1 do {        // element exchange from l to r
            if (A[j] > A[j + 1]) {       // swap A[j] and A[j + 1]
                t := A[j]; A[j] := A[j + 1]; A[j + 1] := t;
            }
        }
        r := r - 1;
    }
}
```

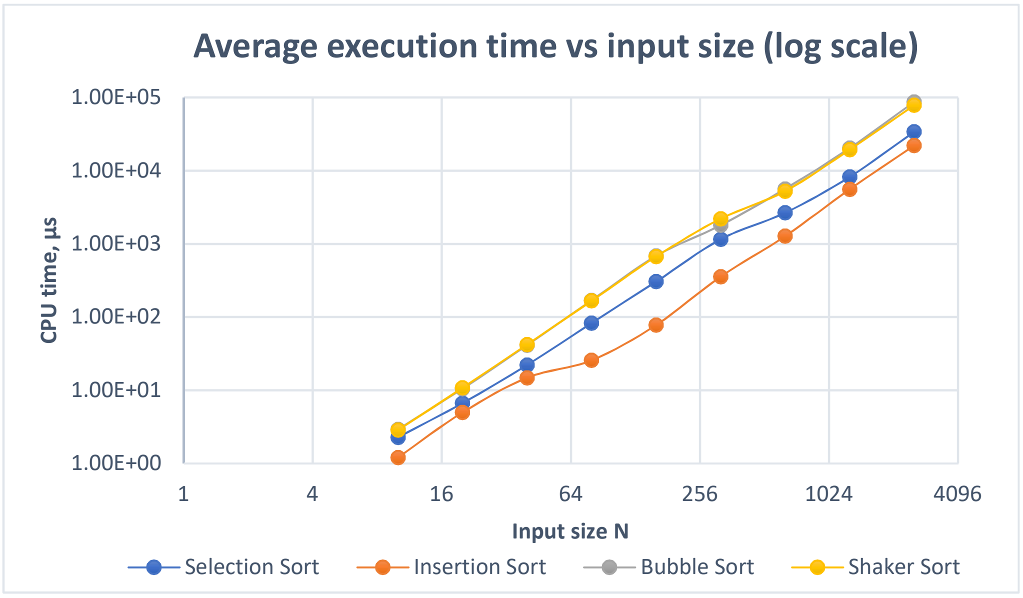
The shaker sort is very similar to the bubble sort. Inside the while loop it performs two bubble sorts – one from the right and one from the left. After each bubble sort, the unsorted area reduces by one. Although under some cases it can be more efficient than bubble sort, for the worst case shaker sort has the same time and space complexity as bubble sort – $O(n^2)$ and $O(1)$.

3. Result and Observation

Input size N	Selection Sort	Insertion Sort	Bubble Sort	Shaker Sort
10	2.28357	1.21212	2.92778	2.85769
20	6.68621	4.98009	10.5400	10.8159
40	22.1720	14.8222	41.3857	42.2621
80	82.8619	25.6281	169.110	166.694
160	305.348	78.0401	684.662	673.282
320	1154.24	357.700	1798.58	2205.04
640	2660.69	1275.13	5642.05	5257.94
1280	8227.41	5581.26	20420.3	19429.4
2560	34145.1	22112.4	87123.1	78771.4

Table 1. Average execution time [μ s] vs input data size





Although the 4 algorithms have same time complexity i.e. the quadratic trend, the actual run time varies. For the bubble sort and shaker sort, since they perform similar operations but only in different orders, we expect them to have similar performance over random inputs. But inside the loop they may need to do multiple times of swapping, it contributes to longer execution time comparing to selection sort and insertion sort, which only do at most two assignments. As for insertion sort, the inner loop has a chance to exit early compared to that of selection sort, which has to run n – i times deterministically. Therefore, we can infer that insertion sort will execute faster than selection sort in our implementation.