Homework 1. Quadratic Sorts report

● Introduction

In this homework, we are requested to implement four sorting algorithms, analyze and compare their performance. The four sorting algorithms are **selection sort**, **insertion sort**, **bubble sort**, and **shaker sort**. The concepts of the four sorting algorithms are shown below:

Algorithm	Concept (We want to sort a list in **non-decreasing** order)
Selection sort	Selection sort is the most intuitive sorting algorithm. From those elements that are currently unsorted, find the smallest and place it next to the sorted list.
Insertion sort	Insertion sort removes one element from the currently unsorted input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.
Bubble sort	Bubble sort compares adjacent elements and swaps them if they are in the wrong order. In each iteration, we try to "bubble" (or squeeze) the smaller elements to the bottom of the list.
Shaker sort	Shaker sort extends Bubble sort by operating in two direction. That is, we try to squeeze the smaller elements to the bottom of the list and squeeze the larger elements to the top of the list in each iteration.

Table 1 The concepts of the four sorting algorithms.

To test and analyze these four algorithms, a main function should be implemented as **Algorithm 1.5** and execute each sorting function one by one to compare the performance using the nine wordlist files. The time complexities of these four algorithms should be analyzed and compare that to the measured CPU time.

● Approach

The following are pseudocodes of four sorting algorithms and their time complexity analysis:

```
// Sort the array A[1 : n] into nondecreasing order.
// Input: A[1 : n], int n
// Output: A, A[i] ≤A[j] if i < j.
1 Algorithm SelectionSort(A,n)
2 {
3       for i := 1 to n do { // for every A[i]
4           j := i; // Initialize j to i
5           for k := i+ 1 to n do // Search for the smallest in A[i+ 1 : n].
6               if (A[k] < A[j]) then j := k; // Found, remember it in j.
7           t := A[i]; A[i] := A[j]; A[j] := t; // Swap A[i] and A[j].
8       }
9 }
```

Algorithm 1 Selection sort

statement	s/e	frequency	total steps
1 Algorithm SelectionSort(A,n)	0	--	0
2 {	0	--	0
3 for i := 1 to n do {	1	n+1	$\Theta(n)$
4 j := i;	1	n	$\Theta(n)$
5 for k := i+ 1 to n do	1	$\Sigma_{i=1 \text{ to } n}$ (n-i+1)	$\Theta(n^2)$
6 if (A[k] < A[j]) then j := k;	1	$\Sigma_{i=1 \text{ to } n}$ (n-i)	$\Theta(n^2)$
7 t := A[i]; A[i] := A[j]; A[j] := t;	3	n	$\Theta(n)$
8 }	0	--	0
9 }	0	--	0
Total			$\Theta(n^2)$

Table 2 Asymptotic complexity of SelectionSort (Algorithm 1)

From Table 2, the asymptotic complexity of SelectionSort is $\Theta(n^2)$. Besides, we can come to the conclusion that the asymptotic complexity is determined by the total steps of the innermost loop. It is the bottleneck of this algorithm. This conclusion can help us simplify the calculation of asymptotic time complexity.

There is also a simpler way to analyze the asymptotic time complexity without using step table. What the selection sort does is simply selecting the minimum element from n elements (taking n-1 comparisons), then selecting the minimum element from n-1 elements (taking n-2 comparisons), and so on. The total number of comparisons is (n-1) + (n-2) + … + 2 + 1 = 0.5 * n * (n-1). Thus, the time complexity is $\Theta(n^2)$.

```
// Sort A[1 : n] into nondecreasing order.
// Input: array A, int n
// Output: array A sorted.
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; // Init 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].
    }
}
```

Algorithm 2 Insertion sort

statement	s/e	frequency		total steps	
		worst case	best case	worst	best
1 Algorithm InsertionSort(A,n)	0	--	--	0	0
2 {	0	--	--	0	0
3 for j := 2 to n do {	1	n	n	O(n)	Ω(n)
4 item := A[j];	1	n-1	n-1	O(n)	Ω(n)
5 i := j −1;	1	n-1	n-1	O(n)	Ω(n)
6 while ((i ≥ 1) and (item < A[i])) do {	1	$\Sigma_{i=1 \text{ to } n-1} (i+1)$	0	$O(n^2)$	0
7 A[i + 1] := A[i];	1	$\Sigma_{i=1 \text{ to } n-1} i$	0	$O(n^2)$	0
8 i := i−1;	1	$\Sigma_{i=1 \text{ to } n-1} i$	0	$O(n^2)$	0
9 }	0	--	--	0	0
10 A[i + 1] = item;	1	n-1	n-1	O(n)	Ω(n)
11 }	0	--	--	--	--
12 }	0	--	--	--	--
iTotal				$O(n^2)$	Ω(n)

Table 3 Asymptotic complexity of InsertionSort (Algorithm 1)

From Table 3, InsertionSort has the best case complexity $\Omega(n)$ and the worst case complexity $O(n^2)$. Actually, it has the average case complexity $\Theta(n^2)$ (from Wikipedia). Besides, compared to SelectionSort, it has the larger loop body. We can conclude that although they have the same average case complexity $\Theta(n^2)$, InsertionSort may be slower.

```
// Sort A[1 : n] into nondecreasing order.
// Input: array A, int n
// Output: array A sorted.
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;
            }
        }
    }
}
```

Algorithm 3 Bubble sort

I want to try to analyze the time complexity without using the step table. From the conclusion above, the asymptotic time complexity will be determined by the innermost loop steps. The inner most loop steps in BubbleSort is $\Sigma_{i=1 \text{ to } n-1}$ $(n – (i + 1) + 1) = \Theta(n^2)$. Thus, the time complexity of BubbleSort is $\Theta(n^2)$.

```
// Sort A[1 : n] into nondecreasing order.
// Input: array A, int n
// Output: array A sorted.
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 down 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;
    }
}
```

Algorithm 4 Shaker sort

The first inner loop executes (n-1) + (n-3) + (n-5) + …… times. The second inner loop executes (n-2) + (n-4) + (n-6) + … times. Thus, the total steps of inner loop is $\Sigma_{i=1 \text{ to } n-1}$ $(n – i) = \Theta(n^2)$. The time complexity of ShakerSort is $\Theta(n^2)$.

The ShakerSort is the advanced version of BubbleSort. Although they have the same time complexity $\Theta(n^2)$, ShakerSort will achieve the slightly better performance than BubbleSort. The reason is that BubbleSort only passes through the list in one direction. However, ShakerSort can pass through the list in two directions. Thus, one ShakerSort pass should be counted as two BubbleSort passes. I thought this is a little bit similar to the loop unrolling concept which is introduced in Computer Architecture Course.

```
// main function for Homework 1.
// Input: read wordlist from stdin
// Output: sorted wordlist and CPU time used.
1 Algorithm main(void)
2 {
3       Read n and a list of n words and store into A array ;
4       t0 := GetTime();
5       repeat 500 times {
6           Copy array A to array list;
7           Call one of the three sort function ;
8       }
9       t1 := GetTime();
10      t := (t1 −t0)/500;
11      write (sorted list);
12      write (sorting method, n , CPU time) ;
13 }
```

Algorithm 5 main function for Homework 1.

Results and Observation

The following figures are the results of four algorithms:

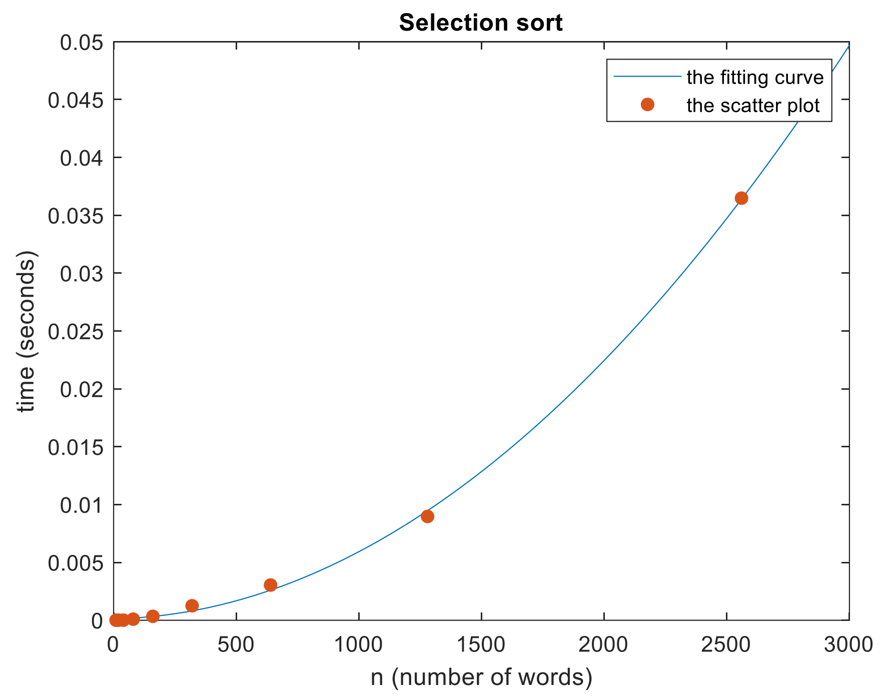

Figure 1 Selection sort result. The fitting curve is a quadratic curve. The result consists with the time complexity Θ(n^2) which I analyzed before.

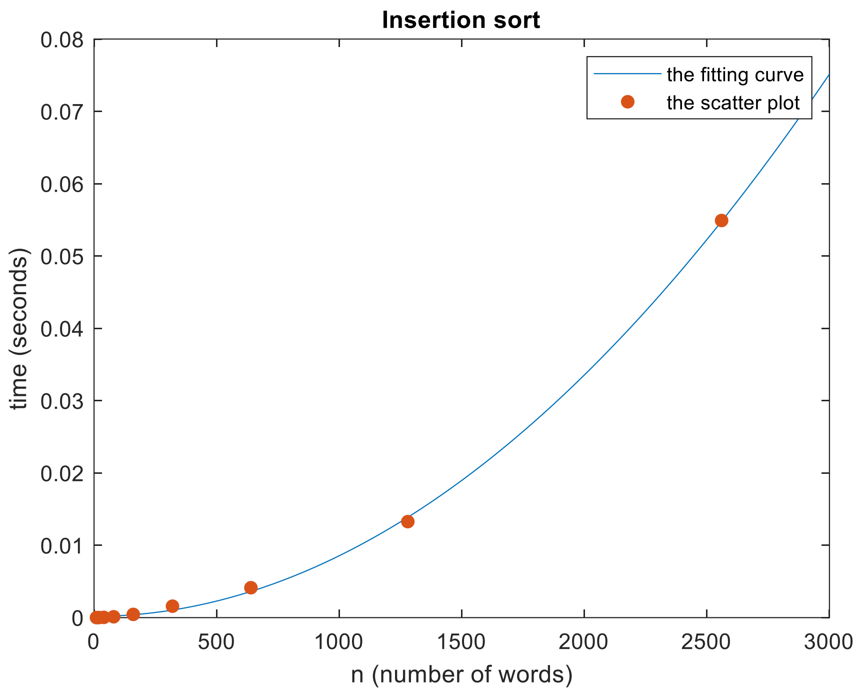

Figure 2 Insertion sort result. The fitting curve is a quadratic curve. The result consists with the average case time complexity Θ(n^2) which I mentioned before.

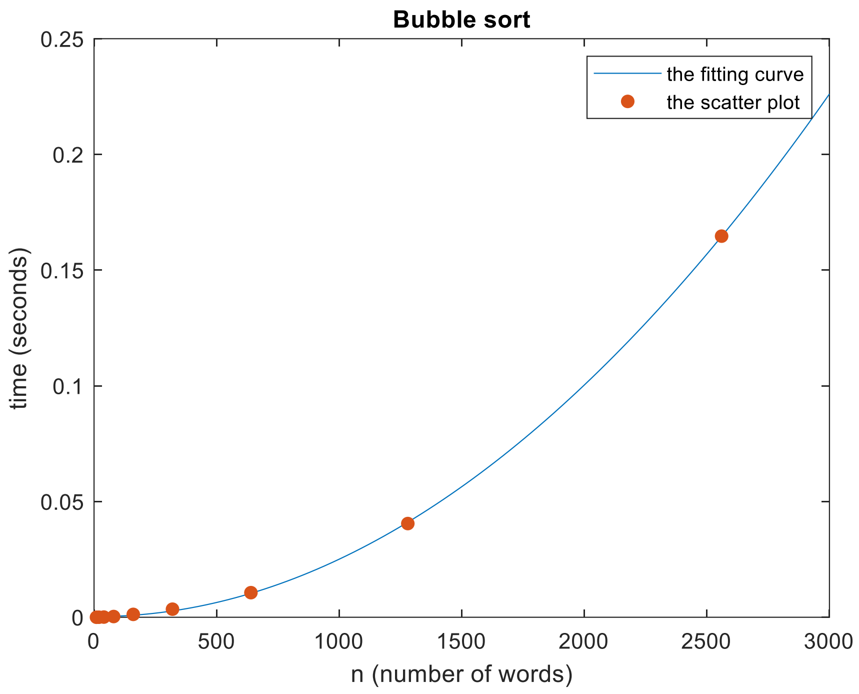

Figure 3 Bubble sort result. The fitting curve is a quadratic curve. The result consists with the time complexity Θ(n^2) which I analyzed before.

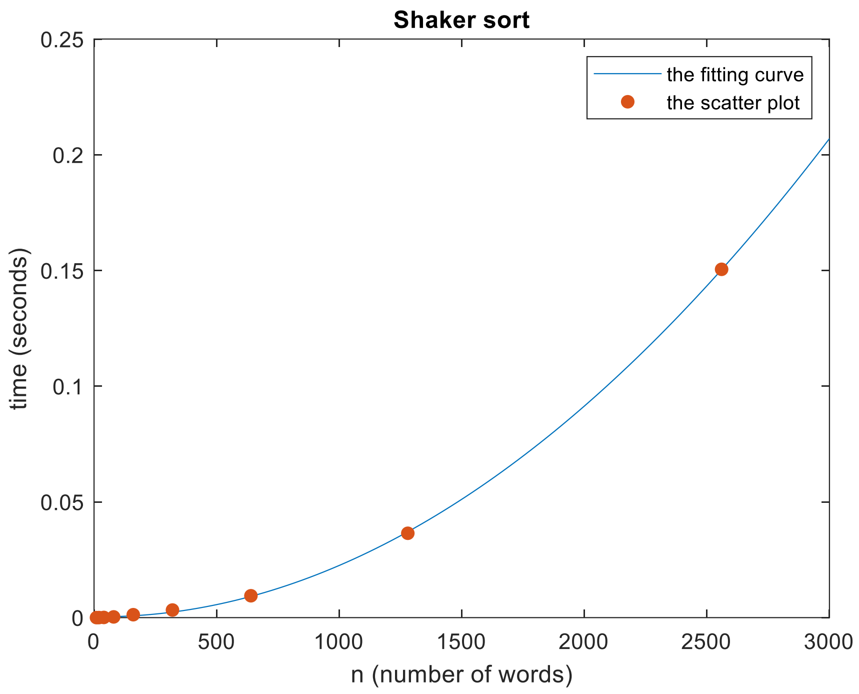

Figure 4 Shaker sort result. The fitting curve is a quadratic curve. The result consists with the time complexity Θ(n^2) which I analyzed before.

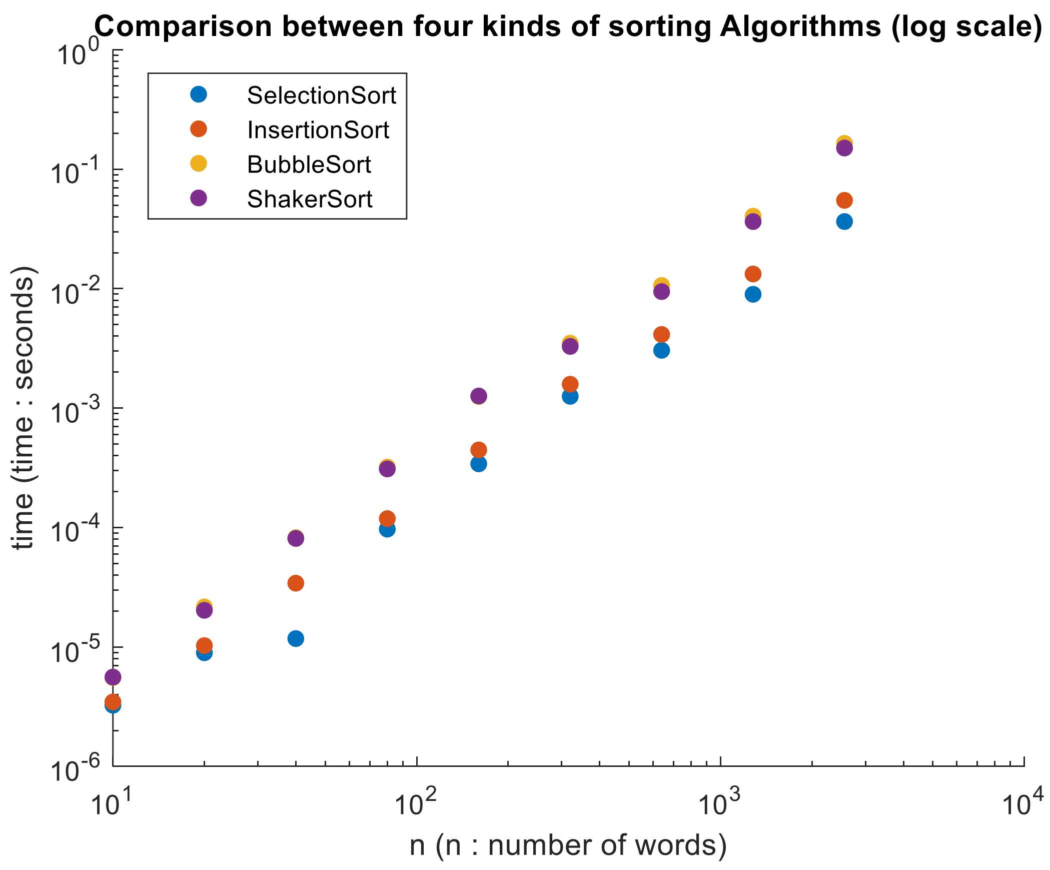

Figure 5 Comparison between four kinds of sorts.

From Figure 5, it seems that SelectionSort is the fastest, InsertionSort is the second, ShakerSort is the third, and BubbleSort is the last (the third and the last are very close because ShakerSort and BubbleSort are nearly the same algorithm). I have already mentioned the reason why SelectionSort is faster than InsertionSort and the reason why ShakerSort is faster than BubbleSort before. The only relation I have not explained is that InsertionSort is faster than BubbleSort (or ShakerSort).

From **Algorithm 2** and **Algorithm 3**, it seems that **Algorithm 3** has the larger innermost loop body (4 statements) than that of **Algorithm 2** (2 statements). Besides, BubbleSort must have to "bubble" the smallest element to the front of the list. However, InsertionSort only needs to put the unsorted element into the right place in the sorted part. Thus, the times innermost loop executed in InsertionSort is more likely to be less than those of BubbleSort. We can come to the conclusion that InsertionSort is faster than BubbleSort, and so is ShakerSort.