10802EE 398000 Algorithms

Homework 3. Heap Sort report

● Introduction

In this report, I will implement the heap sort algorithm, then comparing with four kinds of quadratic sorting algorithms in homework 1: selection sort, insertion sort, bubble sort, and shaker sort. The heap sort algorithm is introduced below :

Algorithm	Concept
Heap sort	The max heap is a complete binary tree with the property that the value at each node is bigger or equal to the values of its children. There are mainly two steps in the heap sort: Heapify and keep interchanging. Heapify is to construct the max heap. Keep interchanging is to interchange the new maximum with the element at the end of the array repeatedly. Then we can get an array with non-decreasing order.

Table 1 The concepts of the heap sort algorithm.

I will also explain the correctness of the heap sort algorithm. After that, I will analyze the time complexity and space complexity of it and compare with those of the other four algorithms theoretically. Then I will explain what kind of dataset can cause the best-case time complexity and the worst-case time complexity happen. Finally, I will implement them and compare the implemented results with the theoretical results to see whether they are the same or not.

● Approach

The following are pseudocodes of the heap sort algorithm, including Algorithm HeapSort, Algorithm Heapify, and Algorithm Adjust, its explanation, and its time and space complexity analysis. I will compare its time and space complexity with the other four kinds of sorting algorithms at the end of this section.

```
// Sort A[1 : n] into nondecreasing order.
// Input: Array A with n elements
// Output: A sorted in nondecreasing order.
1 Algorithm HeapSort(A, n)
2 {
3         Heapify(A, i, n); // Transform the array into a heap
4         for i := n to 2 step −1 do { // Repeat n−1 times
5                 t := A[i]; A[i] := A[1]; A[1] := t; // Move maximum to the end.
6                 Adjust(A, 1, i−1); // Then make A[1 : i−1] a max heap.
7         }
8 }
```

Algorithm 1 HeapSort psedocode

```
// Transfom A[1 : n] into a max heap.
// Input: Array A with n elements
// Output: An array represents the max heap.
1 Algorithm Heapify(A, n)
2 {
3         for i := ⌊n / 2⌋ to 1 step −1 do { // A bottom-up method to make A[1 : n] a max heap.
4                 Adjust(A, i, n);
5         }
6 }
```

Algorithm 2 Heapify psedocode

```
// To enforce max heap property for n-element heap A with root i.
// Input: size n max heap array A, root i
// Output: updated A.
1 Algorithm Adjust(A,i,n)
2 {
3     j := 2×i; // A[j] is the lchild.
4     item := A[i];
5     done := false ;
6     while ((j ≤ n) and ( not done )) do { // A[j + 1] is the rchild.
7         if ((j < n) and (A[j] < A[j + 1])) then
8             j := j + 1; // A[j] is the larger child.
9         if (item >= A[j]) then // If larger than or equal to children, done.
10            done := true ;
11        else { // Otherwise, continue.
12            A[⌊j / 2⌋] := A[j];
13            j := 2×j;
14        }
15    }
16    A[⌊j / 2⌋] := item;
17 }
```

Algorithm 3 Adjuct psedocode

In HeapSort, the first thing we have to do is to make our dataset into the max heap. This can be done by Heapify. The Heapify is a bottom-up method to make all the subtrees from small to large into the max heap. It repeatedly calls the function Adjust. The Adjust is the most important function which traces the tree from the root to the leaf. If the root value is less than the maximum value of the children then swap the value and go to the next level repeatedly, else we can stop the loop.

After heapifing, we keep interchanging our root values to the end of the array then maintaining the max heap structure. Because the root values of the max heap is the maximum, and we keep putting them to the end of the array, we can get the non-decreasing array at the end.

The space complexity of the Heapsort is equal to O(n) because it is characterized by n, the number of elements in the list.

For the time complexity analysis, we analyze the time complexity of the Adjust function first. Its time complexity is equal to the height of the tree which is log n for a complete binary tree. Thus the time complexity of the Adjust function is O(log n).

The time complexity of the Heapsort is determined by two steps: the "Heapify" and "keep interchanging". The time complexity of "keep interchanging" step is determined by the Adjust whose time complexity is O(log n). Since it repeats nearly n times in the Heapsort. The time complexity of "keep interchanging" step is O(n * log n). The time complexity of "Heapify" step is determined by the Adjust either. Besides, it repeats around n / 2 times. Thus, the time complexity of it is also O(n * log n). Finally, we get the time complexity of the Heapsort is O(n * log n).

The pseudocodes of other four kinds of sorting algorithm are shown below:

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

Algorithm 4 Selection sort

```
// Sort A[1 : n] into nondecreasing order.
// Input: array A, int n
// Output: array A sorted.
1 Algorithm InsertionSort(A,n)
2 {
3       for j := 2 to n do { // Assume A[1 : j −1] already sorted.
4           item := A[j ]; // Move A[j ] to its proper place.
5           i := j −1; // Init i to be j −1.
6           while ((i ≥ 1) and (item < A[i])) do { // Find i such that A[i] ≤ A[j ].
7               A[i + 1] := A[i]; // Move A[i] up by one position.
8               i := i−1;
9           }
10          A[i + 1] = item; // Move A[j ] to A[i + 1].
11      }
12 }
```

Algorithm 5 Insertion sort

```
// Sort A[1 : n] into nondecreasing order.
// Input: array A, int n
// Output: array A sorted.
1 Algorithm BubbleSort(A,n)
2 {
3       for i := 1 to n−1 do { // Find the smallest item for A[i].
4           for j := n to i + 1 step −1 do {
5               if (A[j ] < A[j −1]) { // Swap A[j ] and A[j −1].
6                   t = A[j ];
7                   A[j ] = A[j −1];
8                   A[j −1] = t;
9               }
10          }
11      }
12 }
```

Algorithm 6 Bubble sort

```
// Sort A[1 : n] into nondecreasing order.
// Input: array A, int n
// Output: array A sorted.
1 Algorithm ShakerSort(A,n)
2 {
3         ℓ := 1; r := n;
4         while ℓ ≤ r do {
5               for j := r to ℓ + 1 step −1 do { // Element exchange from r down to ℓ
6                     if (A[j ] < A[j −1]) { // Swap A[j ] and A[j −1].
7                           t = A[j ]; A[j ] = A[j −1]; A[j −1] = t;
8                     }
9               }
10              ℓ := ℓ + 1;
11              for j := ℓ to r−1 do { // Element exchange from ℓ to r
12                    if (A[j ] > A[j + 1]) { // Swap A[j ] and A[j + 1].
13                          t = A[j ]; A[j ] = A[j + 1]; A[j + 1] = t;
14                    }
15              }
16              r := r−1;
17        }
18 }
```

Algorithm 7 Shaker sort

Now, lets consider what kinds of dataset can lead to the best case and the worst case of five kinds of algorithms.

In insertion sort, it is obvious. If the dataset is in non-decreasing order, it results in the least execution step of the innermost loop (line 6 to 9 in **Algorithm 5**) which is 1. Thus, the best-case complexity of insertion sort is O(n). The worst-case occurs when the dataset is in non-increasing order because it leads to the largest number of swapping. The worst-case time complexity is $O(n^2)$.

In bubble sort and shaker sort, if the dataset is in non-decreasing order, it can lead to the least times of swapping. Oppositely, if the dataset is in non-increasing order, it can lead to the most times of swapping. However, the times innermost loop executed are the same. Thus the best-case time complexity and the worst-case time complexity is both O(n^2).

In selection sort, the best case is the dataset already with non-decreasing order because it can lead to the least operation step of j := k operation in line 6, **Algorithm 4**. However, the worst case is not only one. There exists many worst cases in the selection sort. For example, if we want to arrange the database which consists the number 1 to 5 in non-decreasing order. The most steps of selection sort algorithm will be 52 steps. There are many possible databases which can reach 52 steps, such as (5, 4, 3, 2, 1), (5, 4, 1, 3, 2), (4, 5, 3, 2, 1), etc. However, the non-increasing order of database can reach the least steps absolutely. Thus, I will use the database with the non-increasing order for the worst-case CPU time.

In heap sort, the condition will become more stranger. Consider the database which consists the number 1 to 5, we want to arrange them into non-decreasing order. The most steps of the algorithm is 83 steps, and the database is (1, 4, 3, 2, 5). The least steps of the algorithm is 68 steps. There are many possible databases which can reach 68 steps such as (5, 3, 4, 2, 1), (5, 3, 4, 1, 2), or (5, 1, 4, 3, 2). Also, I find that the larger steps doesn't mean that the CPU time is longer. There must some issues about CPU time that steps method doesn't consider. I will explain more details in **Results and Observations**. Anyway, I will use the input with non-increasing order for the best-case CPU time and the input with non-decreasing order for the worst-case CPU time. The reason is the input with non-increasing order is easier to heapify since it's already a max heap. Also, it can be easier to do the "keep interchanging" step because it's more likely to have the maximum value which is already at the root. It can help us execute less steps in Adjust function. Oppositely, the input with non-decreasing order is more difficult to heapify. Also, it's harder to "keep interchanging".

	Space complexity	Time complexity		
		Best-case	Worst-case	Average-case
Selection sort	O(n)	O(n^2)	O(n^2)	O(n^2)
Insertion sort	O(n)	O(n)	O(n^2)	O(n^2)
Bubble sort	O(n)	O(n^2)	O(n^2)	O(n^2)
Shaker sort	O(n)	O(n^2)	O(n^2)	O(n^2)
Heap sort	O(n)	O(n * log n)	O(n * log n)	O(n * log n)

Table 2 the space and time complexities of five algorithms

● Results and Observation

I. Average CPU Time :

n	Average CPU time				
	Heap	Selection	Insertion	Bubble	Shaker
10	1.02186E-06	1.35994E-06	8.62122E-07	1.44196E-06	1.60980E-06
20	3.89814E-06	4.28009E-06	2.52581E-06	5.28812E-06	4.64010E-06
40	8.69989E-06	1.26300E-05	7.04813E-06	1.93000E-05	1.86620E-05
80	2.06981E-05	4.52261E-05	2.34680E-05	7.60822E-05	6.26798E-05
160	4.42057E-05	1.18864E-04	8.96401E-05	2.40814E-04	2.34026E-04
320	1.01862E-04	4.46586E-04	3.04694E-04	8.76876E-04	7.60240E-04
640	2.11130E-04	1.73523E-03	1.04000E-03	3.30586E-03	2.94297E-03
1280	4.14534E-04	7.12039E-03	3.97279E-03	1.32251E-02	1.14446E-02
2560	8.29392E-04	2.85398E-02	1.67243E-02	5.38587E-02	4.65332E-02

Table 3 Average CPU time of five kinds of sorting algorithm

Table 3 consists with the theoretical result shown in **Table 2**. The speed of the growth in CPU time in heap sort is the slowest. Also, we can observe that when n is double, the CPU time in heap sort is nearly double. However, the CPU time in other four sorting algorithms is about four times which indicates that their time complexities are quadratic.

The following figures are the comparison of four algorithms in average CPU time:

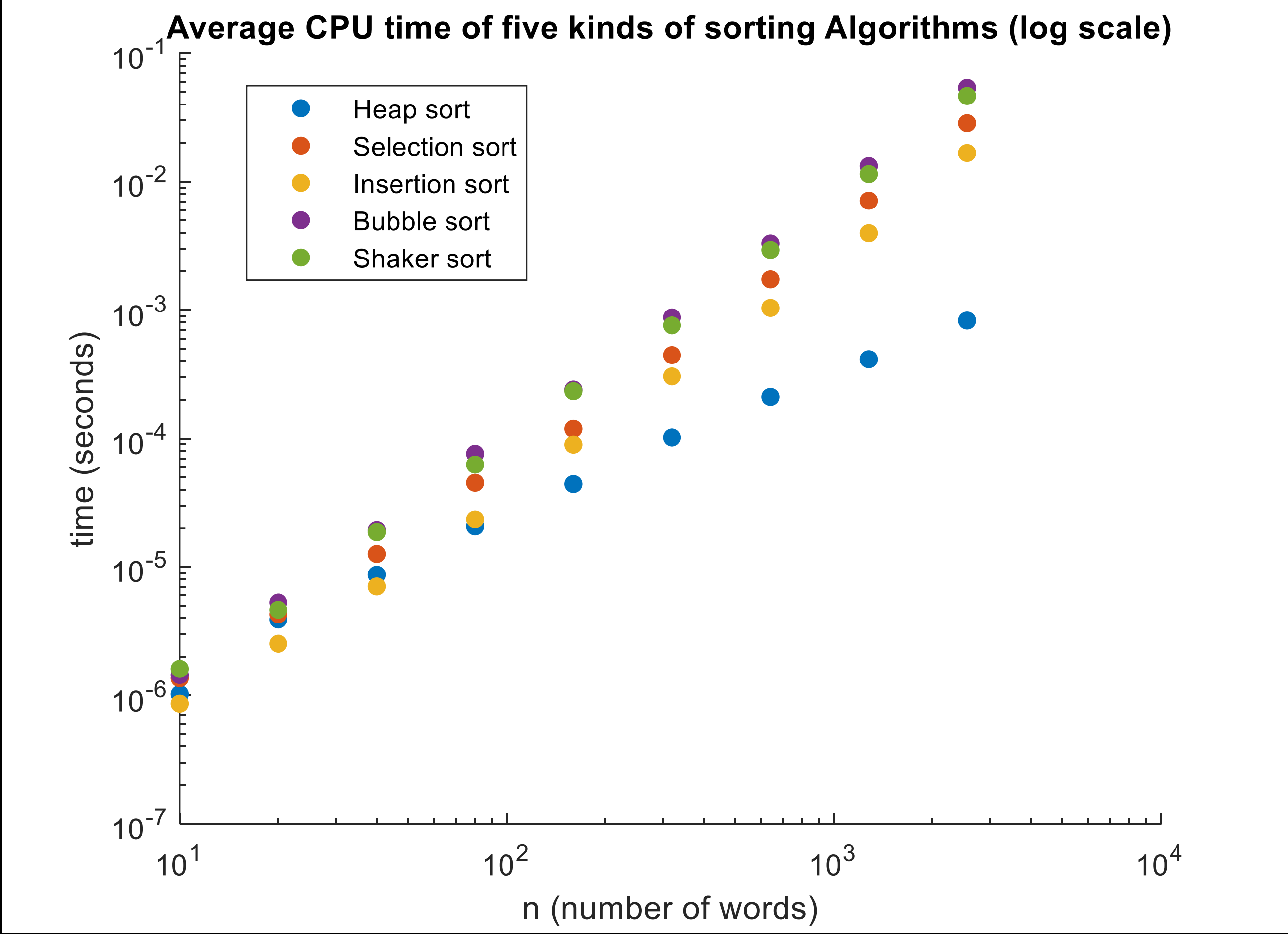

Figure 1 Average CPU time of five kinds of sorting Algorithms.

The heap sort is obviously faster than other four kinds of sorting algorithm. Besides, insertion sort is faster than selection sort because insertion sort can have the chance to execute less steps of the innermost loop, but the times innermost loop executed in selection sort are the same. Selection sort is faster than shaker sort because the times of swaps are less. Shaker sort is a little bit faster than bubble sort because it’s a bidirection version of bubble sort.

II. Best-case CPU Time

n	Best-case CPU time				
	Heap	Selection	Insertion	Bubble	Shaker
10	9.41992E-07	9.09805E-07	2.26021E-07	9.56059E-07	8.75950E-07
20	2.94495E-06	3.06034E-06	4.73976E-07	3.06368E-06	2.81811E-06
40	7.49516E-06	1.09081E-05	8.90255E-07	1.09959E-05	1.10302E-05
80	1.77519E-05	3.78699E-05	1.82819E-06	4.24261E-05	3.90882E-05
160	4.12240E-05	1.05550E-04	3.61633E-06	1.58112E-04	1.14970E-04
320	8.36639E-05	4.16982E-04	7.28226E-06	4.38852E-04	4.39584E-04
640	1.73004E-04	1.65208E-03	1.49980E-05	1.73695E-03	1.73937E-03
1280	3.40627E-04	6.71116E-03	2.95477E-05	8.56068E-03	6.96334E-03
2560	7.24273E-04	2.70053E-02	5.84679E-05	2.84760E-02	2.80732E-02

Table 4 Best-case CPU time of five kinds of sorting algorithm

The best-case CPU time is less than the average CPU time which is shown in **Table3**. Besides, we can observe that the growing speed of CPU time : Selection sort = Bubble sort = Shaker sort > Heap sort > Shaker sort. It consists with the theoretical results shown in **Table 2**.

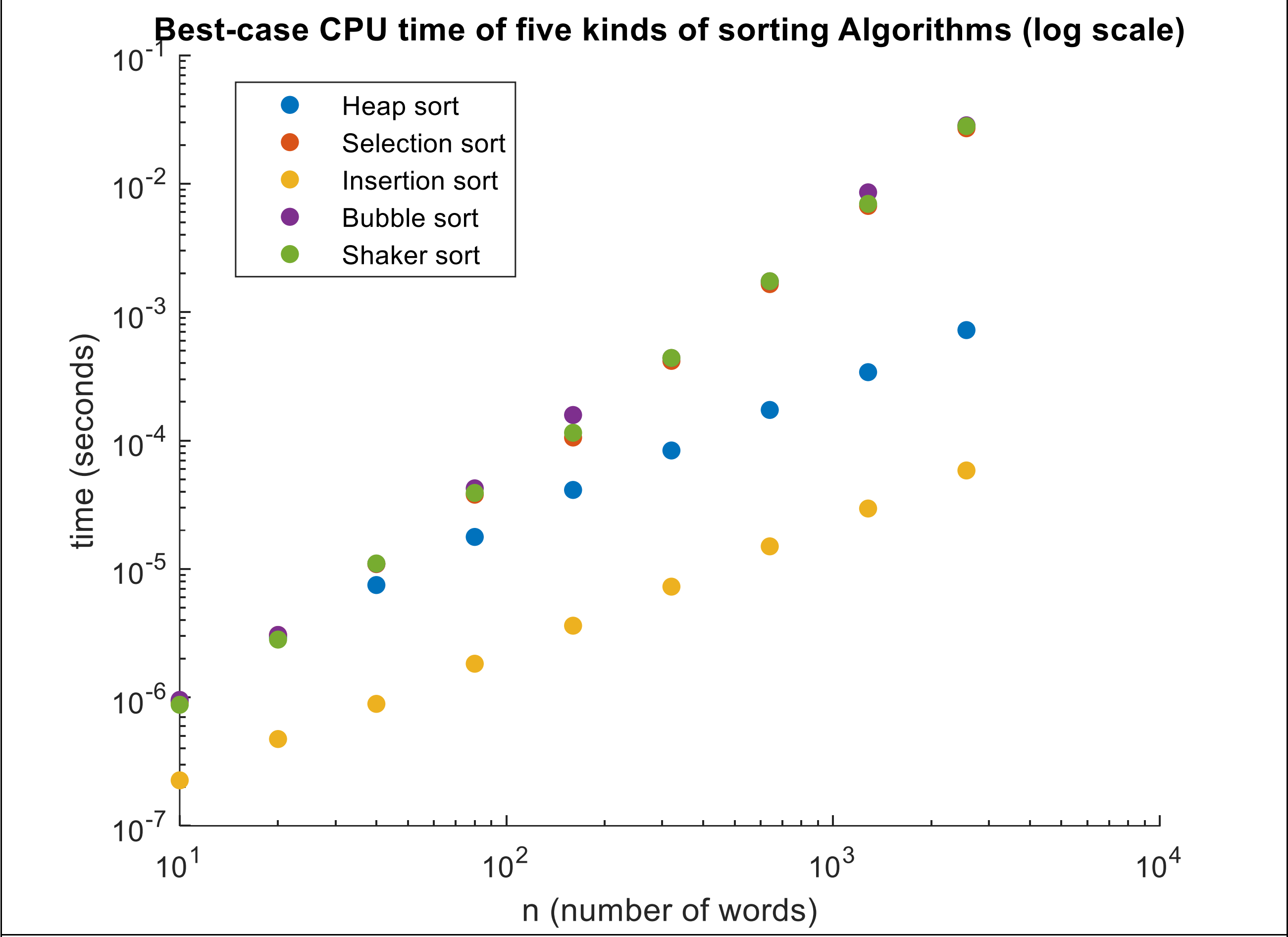

Figure 2 Best-case CPU time of five kinds of searching Algorithms.

Again, Insertion sort is faster than heap sort. Heap sort is faster than the other three kinds of sorts.

III. Worst-case CPU Time

The worst-case CPU time has the strangest result in this report. The following table is the result of worst-case CPU time.

n	Worst-case CPU time				
	Heap	**Selection**	**Insertion**	**Bubble**	**Shaker**
10	1.34802E-06	1.02997E-06	9.92060E-07	1.13702E-06	1.14989E-06
20	3.64399E-06	3.59797E-06	3.56388E-06	4.21500E-06	4.24600E-06
40	8.73184E-06	1.15509E-05	9.47690E-06	1.58741E-05	1.59309E-05
80	2.00539E-05	4.12650E-05	4.27310E-05	6.51910E-05	5.86510E-05
160	4.40061E-05	1.36612E-04	1.30823E-04	1.91167E-04	1.90654E-04
320	9.34219E-05	4.51057E-04	4.85354E-04	7.40518E-04	6.81426E-04
640	1.76535E-04	1.70822E-03	1.93108E-03	2.66179E-03	2.75423E-03
1280	3.58964E-04	6.93617E-03	7.80669E-03	1.10950E-02	1.08107E-02
2560	7.63375E-04	2.74582E-02	3.11762E-02	4.30561E-02	4.30910E-02

Table 5 Worst-case CPU time of five kinds of sorting algorithm

The strangest part is that except for Insertion sort, the other four kinds of sort have the faster Worst-case CPU time than the average CPU time. However, their asymptotic time complexities still consist with **Table 3**. We can observe this result more clearly from **Figure 3**.

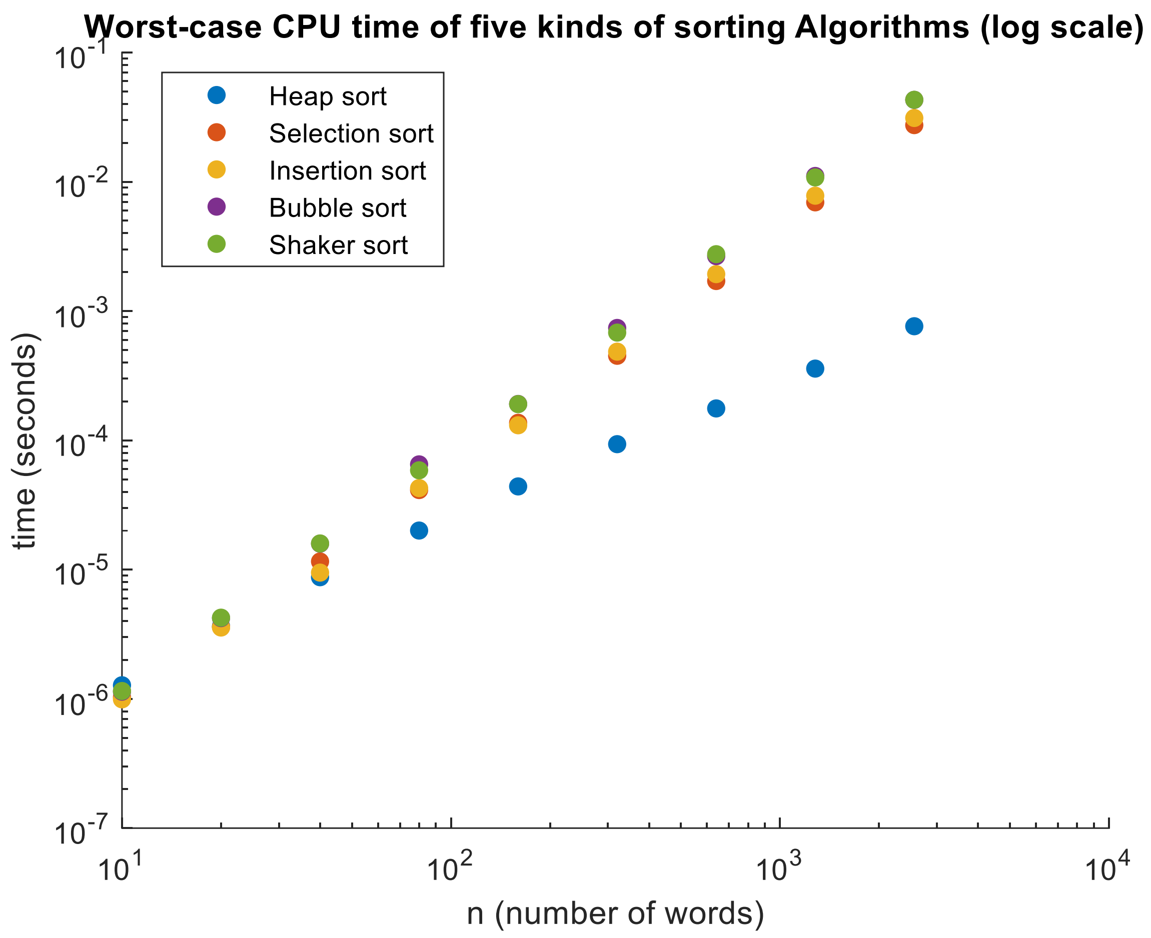

Figure 3 Worst-case CPU time of five kinds of searching Algorithms.

There are two key points in Figure 3. First, their asymptotic time complexities consist with **Table 3**. Second, the worst-case CPU time of Selection sort, Insertion sort, Bubble sort, and Shaker sort are the same which is different from the average CPU time. We can simply think that this is because in the worst case they must execute all steps rather than in the average case they may skip some steps.

Last, I will use an example to illustrate the strange part in worst-case CPU time. This example is to arrange 5 character: ‘1’, ‘2’, ‘3’, ‘4’, and ‘5’. I will try every permutation of them as input database and see which costs the most (least) steps and the longest (shortest) CPU time.

The psedocode of the sample is shown below:

```
// Find the CPU time and steps of each permutation of data
// Input: array list, int n, int k
// Output: no output
Algorithm Perm(list, k, n)
{
    if k = n then {
        Print list;
        count = 0;    // count is the step
        t = GetTime();    // initialize time counter
        for i := 1 to R do { // Repeat R times
            copyArray from list to A;
            do one of the five sorting algorithms;
        }
        t = (GetTime() – t) / R;
        count = count / R;
        print CPU time and count;
    }
    else {
        for i := k to n do {
            swap list[i] and list[k];    // do
            Perm(list, k + 1, n);
            swap list[i] and list[k];    // undo
        }
    }
}
```

Algorithm 8 Testing the CPU time and steps of each permutation

For selection sort, the maximum steps is 52 and the minimum steps is 46. There are many permutations that have steps equal to 52, such as (5, 4, 3, 2, 1), (5, 4, 3, 1, 2), (5, 4, 3, 1, 2). (5, 4, 1, 3, 2), etc. There is only one permutation which has the steps equal to 46. It is (1, 2, 3, 4, 5) as expected. However, the minimum CPU time occurs at the permutation (1, 2, 5, 3, 4) which has the steps equal to 48. The maximum CPU time occurs at the permutation (2, 1, 4, 5, 3) which has the steps equal to 49. However, this is only an example. Everytime I execute, the results CPU time will be different and the maximum and minimum CPU time will occur on different permutation.

It seems that using the steps method will fail in this case. We can't exactly come to the conclusion that if steps is the largest, then the CPU time is largest. This may owing to the compilation technique or other factors that isn't conclude in the steps method. Actually, the step method isn't that precise because it sees each statement as the same step. However, each statement will have the different CPU time. The step method is good for asymptotic analysis rather than too precise analysis.

This condition is also happening in other sorting algorithms except for the best case of insertion sort . The best case of insertion sort is pretty ideal. I think the reason is that the difference between the average and the best case of insertion sort are really obvious, it will affects the execution times of the innermost loop. Thus, we can see the obvious different between them.