

# EE3980 Algorithms

## Homework 2. Heap Sort report

By 105061212 王家駿

### 1. Introduction

In this homework, we use the structure of homework 1 which includes four quadratic sorts, and add the **heap sort** function to the program. Then we compare the efficiency of heap sort with other quadratic sorts, with best-case, worst-case, and average-case analysis included. At the end, we also derive the time and space complexities of the sorting algorithm, and analyze the differences between them.

### 2. Implementation

The program structure is based on homework 1, and modified at several places:

- Use variable size strings by allocating memories based on the string length.
- Use pointers to character while swapping.
- Add functions to implement heap sort.
- Modified the sorting algorithms from 0-based to 1-based.

Since the implementation of the four quadratic algorithms were described in homework 1, we just derive the time/space complexities of them in best-cases, worst-cases, and average cases.

## 2.1. Selection Sort

```
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. }
```

The iteration at line 3 and 5 must go through to the end no matter how the elements of the array was arranged at the beginning. Thus, for the best-case, worst-case, and average-case, the program has to run through the outer loop and the inner loop, so the time complexities are all  $O(n^2)$  for the three cases mentioned above.

For the space complexity, expect the array being sorted, we only need a temporary  $t$  to during the sorting. So the space complexity must be  $O(n)$  (the array length).

Best-case time complexity:  $O(n^2)$

Worst-case time complexity:  $O(n^2)$

Average-case time complexity:  $O(n^2)$

Space complexity:  $O(n)$

## 2.2. Insertion Sort

```
1. Algorithm InsertionSort(A, n)
2. {
3.     for j:=2 to n do{           // A[1:j-1] already sorted
4.         item:=A[j];           // store value of A[j]
5.         i:=j-1;                // initialize i to 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. }
```

The iteration at line 3 must go through to the end ,while the one at line 6 would stop at finding out an  $i$  such that  $A[i] \leq A[j]$ . At the best-case, the iteration at line 6 stops at  $i = j-1$ , which indicates that the loop ends immediately. Thus, the time complexity of the best-case is  $O(n)$ . In the last two cases, the iteration at line 6 might stop at somewhere in the average-case and at the end of the iteration ( $i = 1$ ). Thus, the time complexities of the worst-case and the average-case are both  $O(n^2)$

For the space complexity, expect the array being sorted, we only need a temporary item to during the sorting. So the space complexity must be  $O(1)$  (the array length).

Best-case time complexity:  $O(n)$

Worst-case time complexity:  $O(n^2)$

Average-case time complexity:  $O(n^2)$

Space complexity:  $O(n)$

### 2.3. Bubble Sort

```
1. Algorithm BubbleSort(A, n)
2. {
3.     for i:=1 to n-1 do{                               // find the smallest 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. }
```

The iterations of the loops at line 3 and 4 go to the end, no matter how the elements of the array was arranged at the beginning. Thus, for the best-case, worst-case, and average-case, the program has to run through the outer loop and the inner loop, so the time complexities are all  $O(n^2)$  for the three cases mentioned above.

The algorithm has to run through both the inner and the outer loop even when the array is well-sorted in the beginning, whose time complexity is still  $O(n^2)$ . To minimize the executing time of the best-case, we can add an if-else statement to check

if the array is well-arranged after each outer loop iteration. For the best-case, the array is well-sorted in the beginning, and after the check the loop goes to the end soon.

Since the outer loop only executes for one time, the time complexity can be reduced to  $O(n)$ .

For the space complexity, expect the array being sorted, we only need a temporary  $t$  to during the sorting (in-place sorting). So the space complexity must be  $O(1)$  (the array length).

Best-case time complexity:  $O(n)$  /  $O(n)$  with additional statement

Worst-case time complexity:  $O(n^2)$

Average-case time complexity:  $O(n^2)$

Space complexity:  $O(1)$

## 2.4. Shaker Sort

```
1. Algorithm ShakerSort(A, n)
2. {
3.     l:=1; r:=n;
4.     while l<=r do{                                // exchange from r down to l
5.         for j:=r to l+1 step -1 do{                // swap A[j] and A[j-1]
6.             if(A[j] < 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:=1 to r-1 do{                             // 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. }

```

For the outer loop, the iteration goes to the end when  $l > r$ . Since  $l$  and  $r$  have step 1 and -1 separately at each iteration step, the loop ends when  $l$  and  $r$  meet at the middle of the array, which contributes the time complexity of  $O(n)$ . Then for the two inner loops, the iterations also have time complexities of  $O(n)$  because they have to go through the array until  $l+1$  or  $r-1$ . Thus, the time complexities are all  $O(n^2)$  for the best-case, worst-case, and average-case.

For the space complexity, expect the array being sorted, we only need a temporary  $t$  to during the sorting (in-place sorting). So the space complexity must be  $O(1)$  (the array length).

## 2.5. Heapify

```

1. Algorithm Heapify(A, root, n)
2. {
3.     t:=root; j:=root*2;
4.     while j<=n do{
5.         if(j < n && A[j] < A[j+1]){           // if rchild > lchild
6.             j:=j+1;                           // j is rchild
7.         }
8.         if(t > A[j]){                           // if root > children
9.             break;                             // done
10.        }
11.        else{

```

```

12.         A[j/2]:=A[j];           // place child to parent
13.         j:=j*2;                 // j is child, keep finding
14.     }
15. }
16. A[j/2]:=t;                     // place the root node
17. }
18.

```

This function replaces the root node to rearrange the heap to the form of maxheap.

We found the place where the root node should be inserted from the top, and keep tracing down the children. When tracing to the node  $j$ , if the value of the root node is larger than the children, we place the root node at node  $j$  to form a maxheap. Otherwise, we keep finding by assign  $j$  to the child node which is the larger until finding a right place for the root node.

The loop stops when  $j > n$ , and the  $j$  doubles itself after each iteration step. So the loop would execute at most  $\log_2 n$  times. Thus, the time complexity of heapify once is  $O(\log n)$ .

## 2.6. Heap Sort

```

1. Algorithm HeapSort(A, n)
2. {
3.     for i:=n/2 to 1 step -1 do{
4.         Heapify(A, i, n);       // heapify all the subtrees
                                   // to be a max heap
5.     }
6.
7.     for i:=n to 2 step -1 do{   // repeat n-1 times
8.         t:=A[i];               // swap the first and the last

```

```

9.      A[i]:=A[1];
10.     A[1]:=t;
11.     Heapify(A, 1, i-1);      // make A[1:i-1] be a max heap
12. }
13. }

```

This algorithm rearranges the array in alphabetical order by using the heap sort. At first, we heapify the subtrees with the nodes which have at least one child to make the heap become a maxheap. At each iteration step, we swap the root node, whose value is the largest, with the last node of the array. Then we heapify the tree  $A[1:i-1]$ , due to the last  $i$  nodes were already well-arranged, so the largest element would be placed at the top of the heap, and could be swapped for the next iteration. After the iteration goes to the end, the whole array is well-sorted.

For the two loops of this algorithm, the iterations go almost through the array. Thus, the time complexities are  $O(n * \log n)$ , which had timed the complexities of heapify, for the best-case, worst-case, and average-case.

For the space complexity, expect the array being sorted, we only need a temporary  $t$  to during the sorting (in-place sorting), and the heapify function is also a in-place function. So the space complexity must be  $O(n)$  (the array length).

Best-case time complexity:  $O(n * \log n)$

Worst-case time complexity:  $O(n * \log n)$



Average-case time complexity:  $O(n * \log n)$

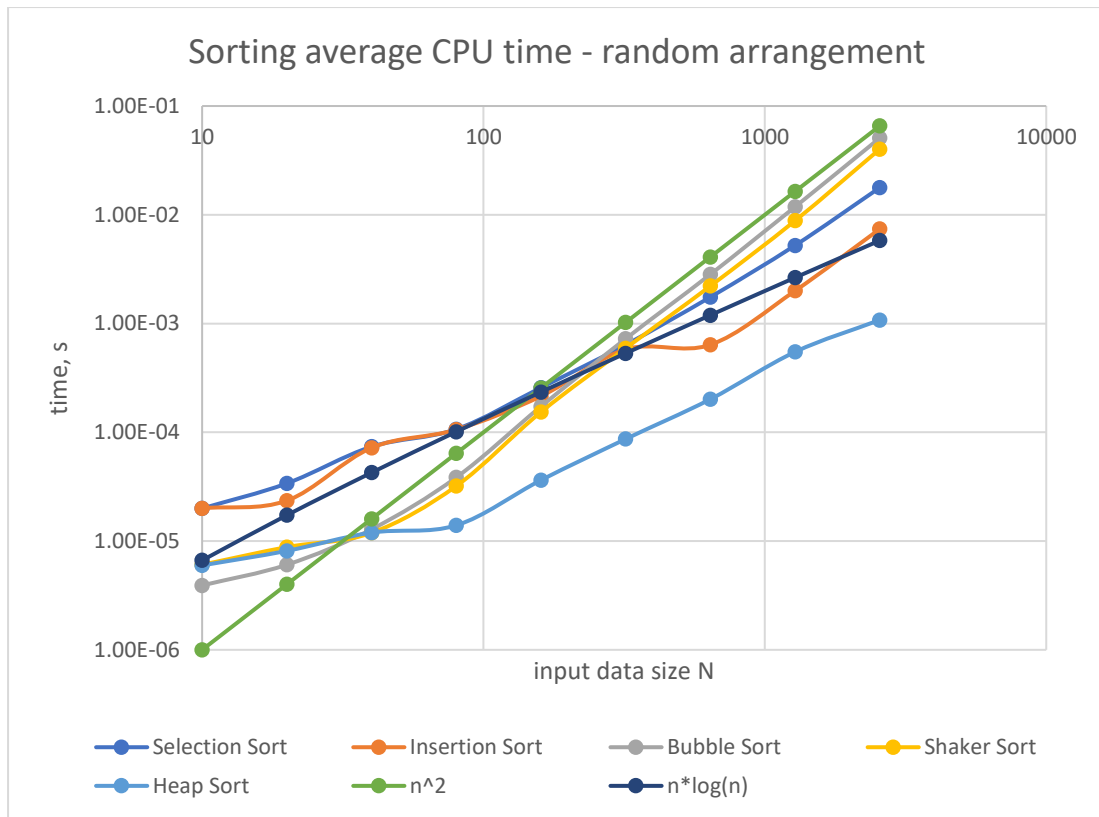
Space complexity:  $O(n)$

### 3. Executing results

For  $R = 500$ , run the testing data from s1.dat to s9.dat with different input data size, and record the average CPU time used.

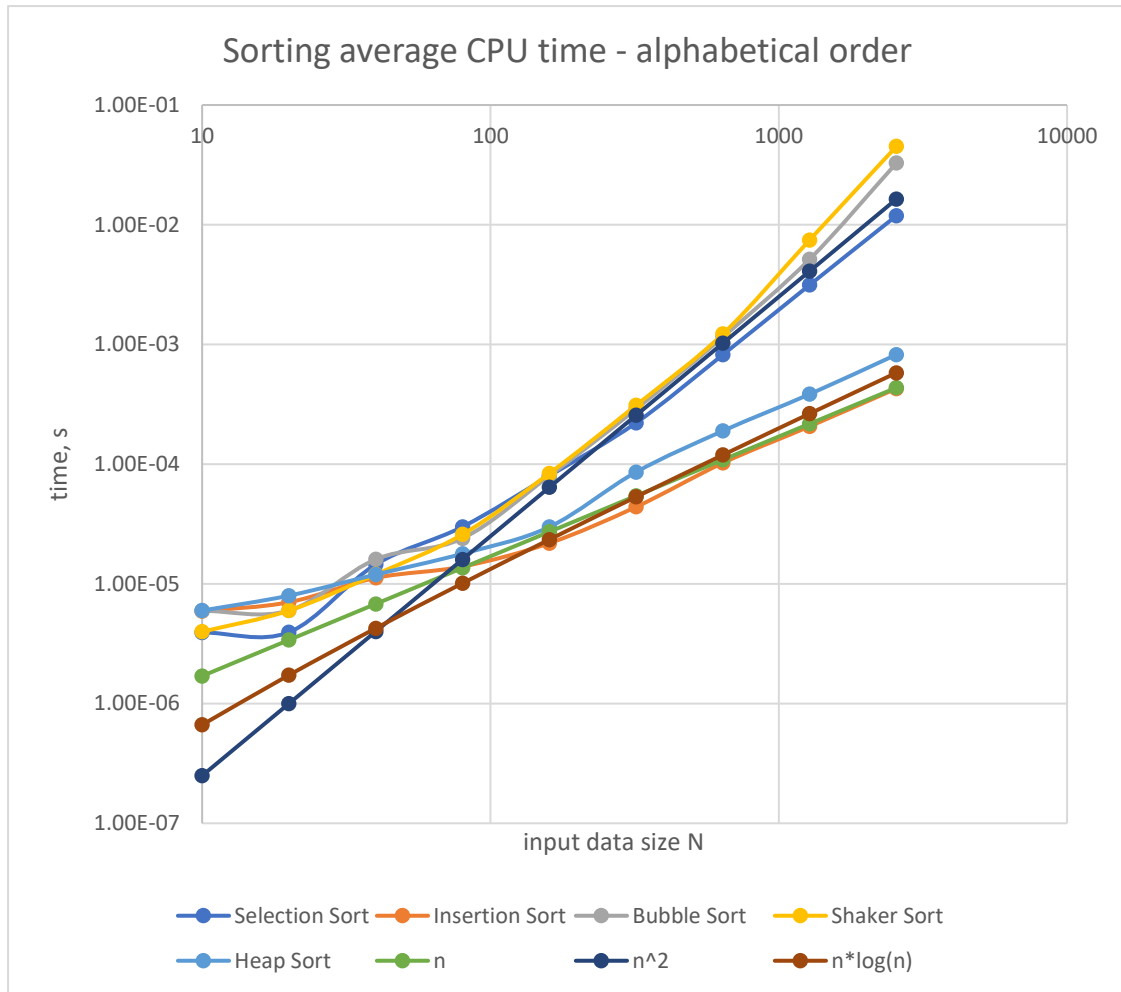
For the random arrangement input:

<b>Data size (N)</b>	<b>Selection Sort</b>	<b>Insertion Sort</b>	<b>Bubble Sort</b>	<b>Shaker Sort</b>	<b>Heap Sort</b>
<b>10</b>	19.95 $\mu$ s	20.04 $\mu$ s	3.912 $\mu$ s	6.060 $\mu$ s	5.976 $\mu$ s
<b>20</b>	33.91 $\mu$ s	23.54 $\mu$ s	6.062 $\mu$ s	8.804 $\mu$ s	8.130 $\mu$ s
<b>40</b>	73.73 $\mu$ s	72.08 $\mu$ s	12.83 $\mu$ s	11.91 $\mu$ s	11.97 $\mu$ s
<b>80</b>	106.7 $\mu$ s	105.7 $\mu$ s	38.45 $\mu$ s	32.04 $\mu$ s	13.95 $\mu$ s
<b>160</b>	257.4 $\mu$ s	214.7 $\mu$ s	172.8 $\mu$ s	153.6 $\mu$ s	36.32 $\mu$ s
<b>320</b>	637.8 $\mu$ s	579.2 $\mu$ s	724.3 $\mu$ s	592.6 $\mu$ s	86.77 $\mu$ s
<b>640</b>	1.751ms	637.6 $\mu$ s	2.831ms	2.212ms	201.5 $\mu$ s
<b>1280</b>	5.225ms	1.999ms	11.83ms	8.817ms	548.5 $\mu$ s
<b>2560</b>	17.76ms	7.398ms	50.77ms	40.13ms	1.078ms



For the alphabetical order input:

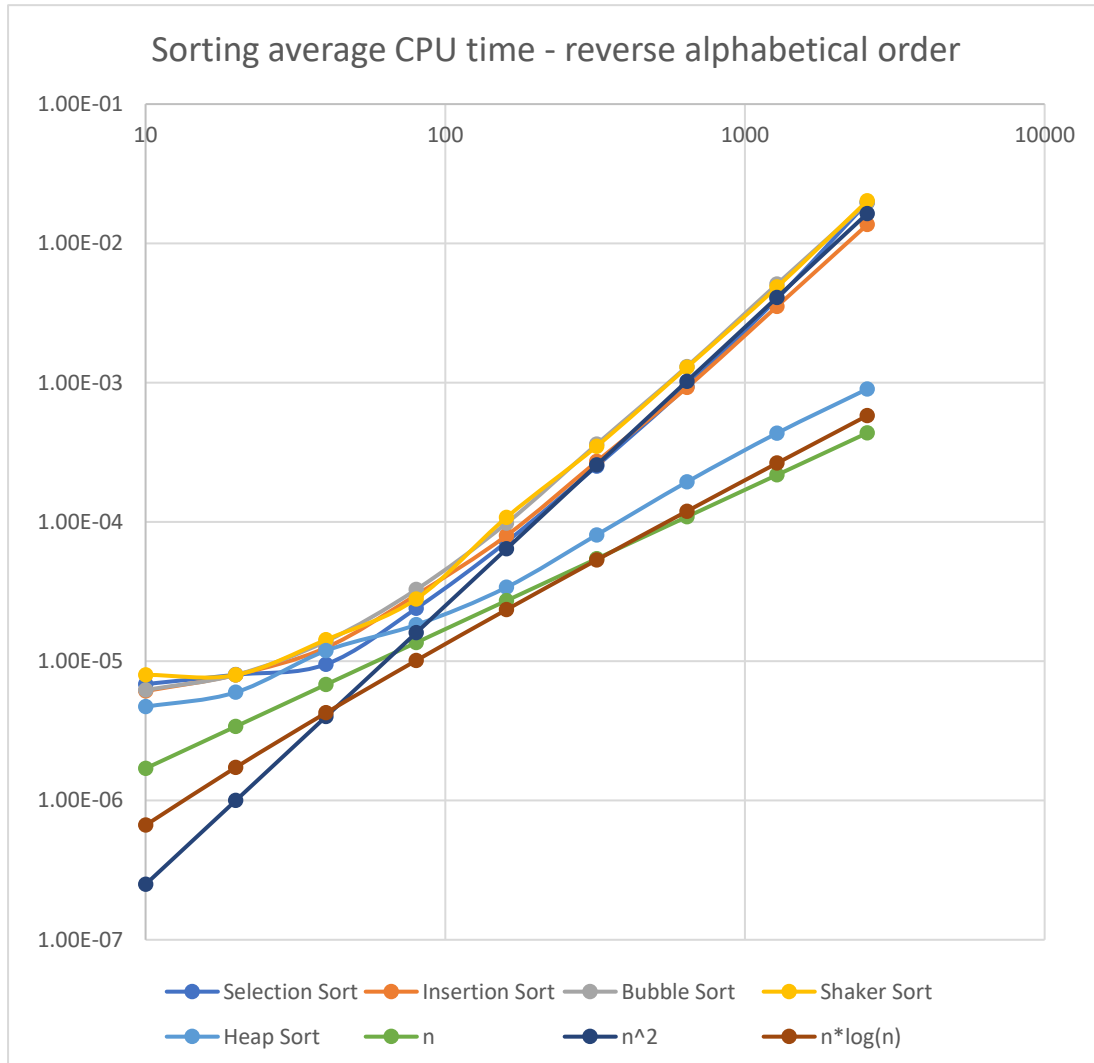
Data size (N)	Selection Sort	Insertion Sort	Bubble Sort	Shaker Sort	Heap Sort
<b>10</b>	3.930 $\mu$ s	5.908 $\mu$ s	5.956 $\mu$ s	3.984 $\mu$ s	5.986 $\mu$ s
<b>20</b>	3.948 $\mu$ s	7.028 $\mu$ s	5.982 $\mu$ s	5.984 $\mu$ s	7.978 $\mu$ s
<b>40</b>	14.53 $\mu$ s	11.22 $\mu$ s	16.04 $\mu$ s	12.00 $\mu$ s	11.97 $\mu$ s
<b>80</b>	29.98 $\mu$ s	13.96 $\mu$ s	24.00 $\mu$ s	25.91 $\mu$ s	17.87 $\mu$ s
<b>160</b>	79.79 $\mu$ s	21.87 $\mu$ s	79.79 $\mu$ s	83.78 $\mu$ s	29.87 $\mu$ s
<b>320</b>	221.5 $\mu$ s	43.95 $\mu$ s	288.4 $\mu$ s	310.2 $\mu$ s	85.85 $\mu$ s
<b>640</b>	820.2 $\mu$ s	102.4 $\mu$ s	1.154ms	1.222ms	189.5 $\mu$ s
<b>1280</b>	3.144ms	206.9 $\mu$ s	5.150ms	7.465ms	384.2 $\mu$ s
<b>2560</b>	11.89ms	428.6 $\mu$ s	32.74ms	45.15ms	823.4 $\mu$ s



For the reverse alphabetical order input:

Data size (N)	Selection Sort	Insertion Sort	Bubble Sort	Shaker Sort	Heap Sort
<b>10</b>	6.854 $\mu$ s	6.108 $\mu$ s	6.220 $\mu$ s	7.982 $\mu$ s	4.712 $\mu$ s
<b>20</b>	7.998 $\mu$ s	7.980 $\mu$ s	7.980 $\mu$ s	7.962 $\mu$ s	5.982 $\mu$ s
<b>40</b>	9.510 $\mu$ s	12.51 $\mu$ s	13.89 $\mu$ s	14.27 $\mu$ s	11.90 $\mu$ s
<b>80</b>	23.93 $\mu$ s	29.94 $\mu$ s	32.69 $\mu$ s	28.00 $\mu$ s	18.25 $\mu$ s
<b>160</b>	71.87 $\mu$ s	79.79 $\mu$ s	97.73 $\mu$ s	107.7 $\mu$ s	33.91 $\mu$ s
<b>320</b>	251.6 $\mu$ s	271.4 $\mu$ s	361.1 $\mu$ s	349.1 $\mu$ s	80.72 $\mu$ s
<b>640</b>	939.6 $\mu$ s	922.1 $\mu$ s	1.305ms	1.287ms	193.5 $\mu$ s

<b>1280</b>	4.001ms	3.531ms	5.105ms	4.857ms	434.1 $\mu$ s
<b>2560</b>	19.66ms	13.64ms	45.79ms	43.37ms	900.5 $\mu$ s



#### 4. Result analysis and conclusion

For the selection sort, we could observe that all of the three input orders lead to the time complexities of  $O(n^2)$  from the graph. With a large number of data input, the average CPU runtime doesn't differ too much (from 11.89ms to 19.66ms). Thus,

there exists a difference, but not too much, on CPU time consumed between the best-case and the worst-case.

For the insertion sort, we could observe that the time complexities are  $O(n^2)$  for the average case and the worst-case, while it can reach the time complexity of  $O(n)$  for the best case (the strings were arranged in alphabetical order at the first). Thus, there is a large scale of decreasing at the time consumed in the best-case.

For the bubble sort, we could observe that all of the three input orders lead to the time complexities of  $O(n^2)$  from the graph. With a large number of data input, the algorithm would be slightly fast at the best-case (alphabetical order), but it is still slower than almost all the other algorithms.

For the shaker sort, we could observe that all of the three input orders lead to the time complexities of  $O(n^2)$  from the graph, and it is also time-consuming as the bubble sort. However, with a large number of data input, the average CPU runtime doesn't differ too much (from 40.13ms to 45.15ms). The reason might be that the shaker sort finds the elements both from the beginning to the end and from the end to the beginning, so the order where the string arranged initially doesn't matter too much for the average CPU time.

For the heap sort, we could observe that all of the three input orders lead to the time complexities of  $O(n * \log(n))$  from the graph. We could also find that the average CPU time measured doesn't differ too much.

When the heap sort operates, we place the last node of the unsorted tree to the root, and keep finding down where it should be placed during heapify. It is hard to minimize the time consumed in this step, because the last node we take is usually in the smaller half of the unsorted part according to the structure of max heap. Thus, the iteration almost has to go through the end of the loop to place the root with small value. In the end, it's difficult to figure out the exact arrangement for the best-case and the worst-case of heap sort. That's why the CPU time consumed doesn't differ too much at the three arrangement orders mentioned above.

The measured results are similar as we predicted before. For the average case and the worst-case, we can use the heap sort whose time complexity is  $O(n * \log(n))$  to minimize the time consumed. And for the best-case, where the strings were already well-arranged, we can use the insertion sort to have the time complexity of  $O(n)$ .