EE3980 Algorithms

Hw03 Heap Sort

106061146 陳兆廷

Introduction:

Paragraph should be double spaced!

In this homework, I'll be analyzing, implementing and observing 5 sorting algorithms: Selection sort, Insertion sort, bubble sort, shaker sort and Heap sort. The goal of the algorithms is to sort an array of vocabularies into alphabetically ordered array. The input of them will be an array of strings, and the output of them will be sorted array of strings.

During the analyzing process, I'll be using table-counting method to calculate the time complexities of 5 algorithms. Furthermore, I'll find the best-case, worst-case and average-case conditions in 5 algorithms accordingly. Before implementing on C code, I'll try to predict the result based on my analysis. Last but not least, I'll calculate their space complexity for the **extra spaces used by the algorithm**.

The implementation of 5 algorithms on C code mainly focus on the average time, best-case and worst-case conditions. 9 testing data are given by Professor Chang, and the length of them are 10 times power of 2s, from 1 to 9.

Lastly, the observation of them will be focusing on the time complexity of the results from implementation. Moreover, I'll compare 5 algorithms with themselves and make some rankings. Finally, I'll check the experimented results with my analysis.

Analysis:

1. Selection Sort:

a. Abstract:

Selection sort is a rather direct approach of searching. In a single iteration, it goes through all the items (n) in the array and remembers and put the smallest element in front of the array. After it repeats the iteration for n times, the whole array is sorted.

b. Algorithm:

```
// 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 {
        j := i;
        for k := i + 1 to n do {
            if (A[k] < A[j]) then j = k
        }
        tmp = list[i]; list[i] = list[j]; list[j] = tmp;
    }
}
```

c. Proof of correctness:

We can find that there’s a truth: at the start of the i-th iteration, the array A[0, … , i - 1] is sorted in the right order, and those of them are smaller than any other elements in A.

During the i-th iteration, the element A[i] will be sorted in the array. Variable j remembers the smallest item in A[i, … , n – 1], and swap it with A[i], and end this iteration. In the next iteration, A[i + 1] will be sorted and the truth, A[0, … , i] is sorted in the right order and smaller than any other elements in A, is still applied. Therefore, the truth applies to all iterations from 1 to n.

The loop terminates when the (n – 1)-th iteration is performed. By that time, A[0, … , n - 1] is sorted. Hence proved.

d. Time complexity:

	s/e	freq	total
`Algorithm SelectionSort(A, n)`	0	0	0
`{`	0	0	0
`    for i := 1 to n do {`	n	1	n
`        j := i;`	1	n - 1	n - 1
`        for k := i + 1 to n do {`	1	c(n-1)	c(n-1)
`            if (A[k] < A[j]) then j = k`	1	c(n-1)	c(n-1)
`        }`	0	0	0
`        tmp = list[i]; list[i] = list[j]; list[j] = tmp;`	3	n – 1	3n-3
`    }`	0	0	0
`}`	0	0	0
p.s. x = n / 2 in worst case	2cn – 2c + 5n - 4		

Worst-case:

When the array is completely in the wrong order, c would be n / 2 since it goes through 1 to n, and the steps would be n^2 + 4n - 4, as the table shows. And the time complexity would be **O(n^2)**.

Best-case:

When the array is completely sorted at the beginning, the steps would be n^2, since the swapping still execute. And the time complexity would be **O(n^2)**.

Average-case:

When the array is randomly ordered, the steps would be 2cn – 2c + 5n - 4, where c is an integer between 1 to n. I choose n / 2 / 2 in this case, since it's the average. Therefore, the steps would be 1/2n^2 + 9/2n - 4, and the time complexity would be **O(n^2)**.

e. Space Complexity:

The algorithm uses 3 integers: i, j, k, and 1 string, tmp for all cases. The space complexity would be **O(1)**.

2. Insertion Sort:

a. Abstract:

Insertion sort is a sorting algorithm that gradually inserts the item in the previously sorted array in the right order.

b. Algorithm:

```
// 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 s.t. 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].
    }
}
```

c. Proof of correctness:

We can find that there's a truth: at the start of the i-th iteration, the array A[0, … , i - 1] is sorted in the right order, and those of them are smaller than any other elements in A.

During the i-th iteration, the element A[i] will be sorted in the array.

Through A[0] to A[i – 1], the algorithm moves whatever is larger than A[i] 1 position right, and inserts A[i] in its right spot. In the next iteration, A[i + 1] will be sorted and the truth, A[0, … , i] is sorted in the right order and smaller than any other elements in A, is still applied. Therefore, the truth applies to all iterations from 1 to n.

The loop terminates when the (n – 1)-th iteration is performed. By that time, A[0, … , n - 1] is sorted. Hence proved.

d. Time complexity:

	s/e	freq	total
`Algorithm InsertionSort(A, n)`	0	0	0
`{`	0	0	0
`  for j := 2 to n do {`	n-1	1	n-1
`    item := A[j ] ;`	1	n-2	n-2
`    i := j – 1 ; .`	1	n-2	n-2
`    while ((i ≥ 1) and (item < A[i])) do {`	2c	n-2	2cn-4c
`      A[i + 1] := A[i] ;`	1	c(n-2)	cn-2n
`      i := i – 1 ;`	1	c(n-2)	cn-2n
`    }`	0	c(n-2)	0
`    A[i + 1] = item ;`	1	n-2	n-2
`  }`	0	0	0
`}`	0	0	0
	4cn-4c-7		

Worst-case:

When the array is completely in the wrong order, c would be n / 2 since it goes through 1 to n, and the steps would be 2n^2 - 9, as the table shows. And the time complexity would be **O(n^2)**.

Best-case:

Can be more clearly described.

When the array is completely sorted at the beginning, the steps would be n - 1, since the still execute on first layer. And the time complexity would be **O(n)**.

Average-case:

When the array is randomly ordered, the steps would be 4cn – 4c -7, where c is an integer between 1 to n. I choose n / 2 / 2 in this case, since it's the average. Therefore, the steps would be n^2 - 8, and the time complexity would be **O(n^2)**.

e. Space Complexity:

The algorithm uses 2 integers: i, j , and 1 string, item for all cases (no tmp for best-case). The space complexity would be **O(1)**.

3. Bubble Sort:

a. Abstract:

Bubble sort gradually swaps every unordered pairs it goes through in the right order in approximately n * n iterations.

b. Algorithm:

```
// 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].
        tmp = A[j ] ; A[j ] = A[j - 1] ; A[j - 1] = tmp;
      }
    }
  }
}
```

c. Proof of correctness:

We can find that there's a truth: at the start of the i-th iteration, the array A[0, … , i - 1] is sorted in the right order, and those of them are smaller than any other elements in A.

During the i-th iteration, the element A[i] will be sorted in the array. Through A[i - 1] to A[0], the algorithm swaps whatever is larger than A[i] with A[i]. In the end of the iteration, A[i] will be in the right spot. In the next iteration, A[i + 1] will be sorted and the truth, A[0, … , i] is sorted in the right order and smaller than any other elements in A, is still applied. Therefore, the truth applies to all iterations from 1 to n.

The loop terminates when the (n – 1)-th iteration is performed. By that time, A[0, … , n - 1] is sorted. Hence proved.

d. Time complexity:

	s/e	freq	total
`Algorithm BubbleSort(A, n)`	0	0	0
`{`	0	0	0
`  for i := 1 to n - 1 do {`	n-1	1	n-1

`for j := n to i + 1 step -1 do {`	c	n-1	c(n-1)
`if (A[j ] < A[j - 1]) {`	1	c(n-1)	c(n-1)
`tmp = A[j];`	1	c(n-1)	c(n-1)
`A[j] = A[j - 1];`	1	c(n-1)	c(n-1)
`A[j - 1] = tmp;`	1	c(n-1)	c(n-1)
`}`	0	0	0
`}`	0	0	0
`}`	0	0	0
`}`	0	0	0
	5cn + n - 5c - 1		

Worst-case:

When the array is completely in the wrong order, c would be n / 2 since it goes through 1 to n, and the steps would be 5/2n^2 +n 3/2, as the table shows. And the time complexity would be **O(n^2)**.

Best-case:

When the array is completely sorted at the beginning, the steps would be n - 1, since the comparing still execute on the first layer. And the time complexity would be **O(n)**.

Average-case:

When the array is randomly ordered, the steps would be 5cn + n - 5c - 1, where c is an integer between 1 to n. I choose n / 2 / 2 in this case, since it's the average. Therefore, the steps would be 5/4n^2 – n – 5/4c, and the time complexity would be **O(n^2)**.

e. Space Complexity:

The algorithm uses 2 integers: i, j , and 1 string, tmp for all cases. The space complexity would be **O(1)**.

4. Shaker Sort:

a. Abstract:

Shaker sort is a sorting algorithm derives from bubble sort. The major difference between them is that shaker sort's swapping goes from both ends respectively, and terminates at the middle, while bubble sort's swapping only goes from one end and terminates at another.

b. Algorithm:

```
// 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 ;
  }
}
```

c. Proof of correctness:

We can find that there's a truth: at the start of the i-th iteration, arrays A[0, … , i - 1] and A[n – i ,… ,n – 1] are sorted in the right order, and those of them are smaller or larger respectively than any other elements in A.

During the i-th iteration, the element A[i] and A[n – i – 1] will be sorted in the array. Through A[0] to A[i – 1], the algorithm moves whatever is larger than A[i] 1 position right, and inserts A[i] in its right spot. Through A[n – i] to A[n – 1], the algorithm moves whatever is smaller than A[n – i - 1] 1 position left, and inserts A[n – i - 1] in its right spot. In the next iteration, A[i + 1], A[n - i - 2], will be sorted and the truth, A[0, … , i] and A[n – i - 1, … , n - 1] are sorted in the right order and smaller or larger than any other elements in A, is still applied. Therefore, the truth applies to all iterations from 1 to n.

The loop terminates when the (n / 2)-th iteration is performed. By that time, A[0, … , n - 1] is sorted. Hence proved.

d. Time complexity:

	s/e	freq	total
`Algorithm ShakerSort(A, n)`	0	0	0
`{`	0	0	0
`  ℓ := 1 ; r := n ;`	2	1	2
`  while ℓ ≤ r do {`	n/2	1	n/2

`    for j := r to ℓ + 1 step -1 do {`	c	n/2	cn/2
`      if (A[j ] < A[j - 1]) {`	1	cn/2	cn/2
`        t = A[j ] ; A[j ] = A[j - 1] ; A[j - 1] = t;`	3	cn/2	3cn/2
`      }`	0	n/2	0
`    }`	0	n/2	0
`    ℓ := ℓ + 1 ;`	1	n/2	n/2
`    for j := ℓ to r - 1 do {`	c	n/2	cn/2
`      if (A[j ] > A[j + 1]) {`	1	cn/2	cn/2
`        t = A[j ] ; A[j ] = A[j + 1] ; A[j + 1] = t;`	3	cn/2	3cn/2
`      }`	0	n/2	0
`    }`	0	n/2	0
`    r := r - 1 ;`	1	n/2	n/2
`  }`	0	0	0
`}`	0	0	0
	5cn + 3/2n + 2		

Worst-case:

When the array is completely in the wrong order, c would be n / 4 since it goes through 1 to n, and the steps would be 5/4n^2 – 3/2n + 2, as the table shows. And the time complexity would be **O(n^2)**.

Best-case:

When the array is completely sorted at the beginning, the steps would be n, since the comparing still execute on the first layer. And the time complexity would be **O(n)**.

Average-case:

When the array is randomly ordered, the steps would be 5cn – 3/2c + 2, where c is an integer between 1 to n. I choose n / 4 / 2 in this case, since it's the average. Therefore, the steps would be 5/8n^2 – 3/16, and the time complexity would be **O(n^2)**.

e. Space Complexity:

The algorithm uses 2 integers: j, r, l , and 1 string, t for all cases. The space complexity would be **O(1)**.

5. Heap Sort:

a. Abstract:

Heap sort is a sorting method derives from Heap tree. With heap-representation of the array, heap sort can easily sort input array by *Heapify*, which is an algorithm that maintain heap-property on specific node.

The process of Heap sort comes in 2 parts. First, it makes sure that the array is a max heap. Then, it takes the largest item, which is the root of the heap tree, to the end of the array. After that, it performs *Heapify* on those except the taken elements. Gradually, the items from large to small will be sorted at the end of the array, and finally complete the sorting.

What is a max heap?

b. Algorithm:

```
// 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.
Algorithm Heapify(A, i, n)
{
    j := 2×i ; // A[j ] is the lchild.
    item := A[i ] ;
    done := false ;
    while ((j <= n) and ( not done )) do { // A[j + 1] is the rchild.
        if ((j < n) and (A[j ] < A[j + 1])) then
            j := j + 1 ; // A[j ] is the larger child.
        if (item > A[j ]) then // If larger than children, done.
            done := true ;
        else { // Otherwise, continue.
            A[⌊j/2⌋] := A[j ] ;
            j := 2×j ;
        }
    }
    A[⌊j/2⌋] := item ;
}
```

```
// Sort A[1 : n] into nondecreasing order.
// Input: Array A with n elements
// Output: A sorted in nondecreasing order.
Algorithm HeapSort(A, n)
{
    for i := ⌊n/2⌋ to 1 step -1 do // Initialize A[1 : n] to be a max heap.
        Heapify(A, i, n) ;
    for i := n to 2 step -1 do { // Repeat n − 1 times
```

```
        t := A[i ] ; A[i ] := A[1] ; A[1] := t ; // Move maximum to the end.
        Heapify(A, 1, i - 1) ; // Then make A[1 : i - 1] a max heap.
    }
}
```

c. Proof of correctness:

I divide the proof into 2 parts. The correctness of *Heapify* and correctness of *HeapSort*.

In the *Heapify* process, it starts from trees that has leaves. It swaps child and root when the order is not right. After it finishes max-heaping the deepest trees, which is approximately between A[n/2] to A[n/4], it goes through A[n/4] to A[n/8], which is second-deepest trees. Finally, *Heapify* runs at the root, which was A[1] and finds the largest item in the array. This way, no matter where the largest item is in the array, by checking each tree, it can find every nodes in the array and find the maximum.

In the *HeapSort* process, it finds the largest value first, then it puts the item at the end of array and repeat the process on array except the item.

We can find that there’s a truth: at the start of the i-th iteration, array A[n - i, … , n - 1] is sorted in the right order, and those of them are larger than any other elements in A.

During the i-th iteration, the element A[n – i – 1] will be sorted in the array. Through A[0] to A[i – 1], the algorithm finds the maximum value by *Heapify* and put it at A[n – i – 1]. In the next iteration, A[n - i - 2] will be sorted and the truth, A[n – i - 1, … , n - 1] is sorted in the right order and larger than any other elements in A, is still applied. Therefore, the truth applies to all iterations from 1 to n.

The loop terminates when the (n – 1)-th iteration is performed. By that time, A[0, … , n - 1] is sorted. Hence proved.

d. Time complexity :

	s/e	freq	total
`Algorithm Heapify(A, i, n)`	0	0	0
`{`	0	0	0
`  j := 2×i ;`	1	1	1
`  item := A[i ] ;`	1	1	1
`  done := false ;`	1	1	1
`  while ((j <= n) and ( not done )) do {`	log(n)	1	log(n)

7. `if ((j < n) and (A[j ] < A[j + 1])) then`	1	log(n)	log(n)
8. `j := j + 1 ;`	1	log(n)	log(n)
9. `if (item > A[j ]) then`	1	log(n)	log(n)
10. `done := true ;`	1	log(n)	log(n)
11. `else {`	1	log(n)	log(n)
12. `A[⌊j/2⌋] := A[j ] ;`	1	log(n)	log(n)
13. `j := 2×j ;`	1	log(n)	log(n)
14. `}`	0	0	0
15. `}`	0	0	0
16. `A[⌊j/2⌋] := item ;`	1	1	1
17. `}`	0	0	0
	c * log(n)		
	s/e	freq	total
1. `Algorithm HeapSort(A, n)`	0	0	0
2. `{`	0	0	0
3. `for i := ⌊n/2⌋ to 1 step -1 do`	log(n)	1	log(n)
4. `Heapify(A, i, n) ;`	log(n)	log(n)	n
5. `for i := n to 2 step -1 do {`	n	1	n
6. `t := A[i ] ; A[i ] := A[1] ; A[1] := t ;`	3	n-1	3n-3
7. `Heapify(A, 1, i - 1) ;`	log(n)	n-1	(n-1)log(n)
8. `}`	0	0	0
9. `}`	0	0	0
	(n-1) log(n) + c * log(n)+5n-3		

Worst-case:

When the array is completely in the wrong order, c would be larger than those in other case, and the steps would be the largest. But here, I'll conclude the steps of it to (n-1) log(n). And the time complexity would be **O(nlog(n))**. Can be more clear!

Best-case:

When the array is completely sorted at the beginning, the steps would be still be c * (n - 1)log(n), since the max-heaping still execute. And the time complexity would be **O(nlog(n))**.

Average-case:

When the array is randomly ordered, the steps would be (n-1) log(n) + c * log(n)+5n-3, where c is an integer depends on the times that it executed in *Heapify*. It is too hard to measure. Therefore, the steps would be (n-1) log(n), and the time complexity would be **O(nlog(n))**.

e. Space Complexity:

The algorithm uses 2 integers: i, j in heapify, i in HeapSort, and 2 strings, tmp for all cases. The space complexity would be **O(1)**.

6. Comparison:

	Selection	Insertion	Bubble	Shaker	Heap
Best	n^2	n - 1	n-1	n	(n-1) log(n)
Worst	n^2+4n-4	2n^2 - 9	5/2n^2+n 3/2	5/4n^2 – 3/2n + 2	(n-1) log(n)
Average	1/2n^2+9/ 2n-4	n^2 - 8	5/4n^2 – n – 5/4c	5/8n^2– 3/16	(n-1) log(n)

	Selection	Insertion	Bubble	Shaker	Heap
Best	O(n^2)	O(n)	O(n)	O(n)	O(nlog(n))
Worst	O(n^2)	O(n^2)	O(n^2)	O(n^2)	O(nlog(n))
Average	O(n^2)	O(n^2)	O(n^2)	O(n^2)	O(nlog(n))

Average Speed (fast>slow): Heap > Selection > Shaker > Insertion > Bubble.
Worst-case Speed (fast>slow): Heap > Selection > Insertion > Shaker > Bubble.
Best-case Speed (fast>slow): Insertion ~= Bubble ~= Shaker > Heap > Selection.

Implementation:

1. Average-case scenario:

Since the most average case of testing the algorithm is to sort a randomly ordered array. I simply sort the original data, which is randomly ordered, and implement on the algorithm, repeat the procedure for 500 times. The mean of the recorded CPU time between those steps are the average CPU runtime of the algorithm.

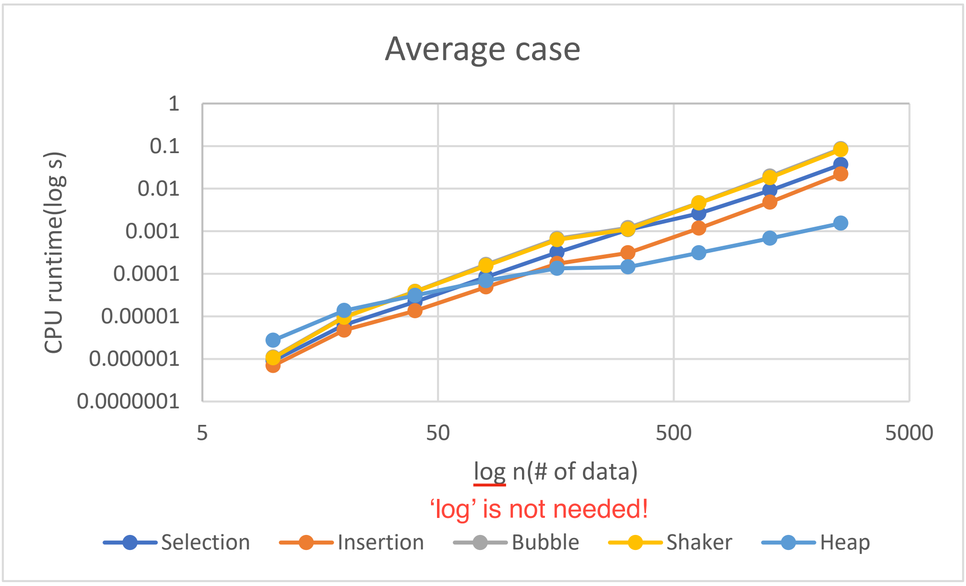
Workflow:

```
t = GetTime();                          // initialize time counter
for i := 0 to 500 do {
    ResetArray(data, list);            // reset array
    Sort(list, n);         // sort list
}
```

```
t = (GetTime() - t) / 500;                  // calculate CPU time / iteration
```

Results: Time unit?

N(num)	Selection	Insertion	Bubble	Shaker	Heap
10	9.05991E-07	6.9809E-07	1.09625E-06	1.05763E-06	2.73991E-06
20	6.31237E-06	4.73595E-06	9.82809E-06	9.47189E-06	1.366E-05
40	2.19498E-05	1.34721E-05	3.84836E-05	3.80139E-05	3.09277E-05
80	8.44002E-05	4.87981E-05	0.000165072	0.000154416	6.86383E-05
160	0.000316864	0.000173066	0.000668294	0.000627528	0.000133876
320	0.001084506	0.000308762	0.001204406	0.001110638	0.000144216
640	0.00261756	0.00117092	0.004635394	0.00458009	0.00030979
1280	0.009051426	0.004790186	0.01961713	0.01803204	0.000680402
2560	0.03666886	0.02205878	0.0865522	0.08182554	0.001532816



2. Worst-case scenario:

For each case, the worst-case scenario is when the array is completely reverse ordered wrong-ordered, but except Heap sort. For Heap sort, since it finds Max-heap first, the worst-case of it should be alphabetically ordered. Therefore, I sort the data into reverse-alphabetically order and alphabetically order respectively, and implement on each sorting algorithms.

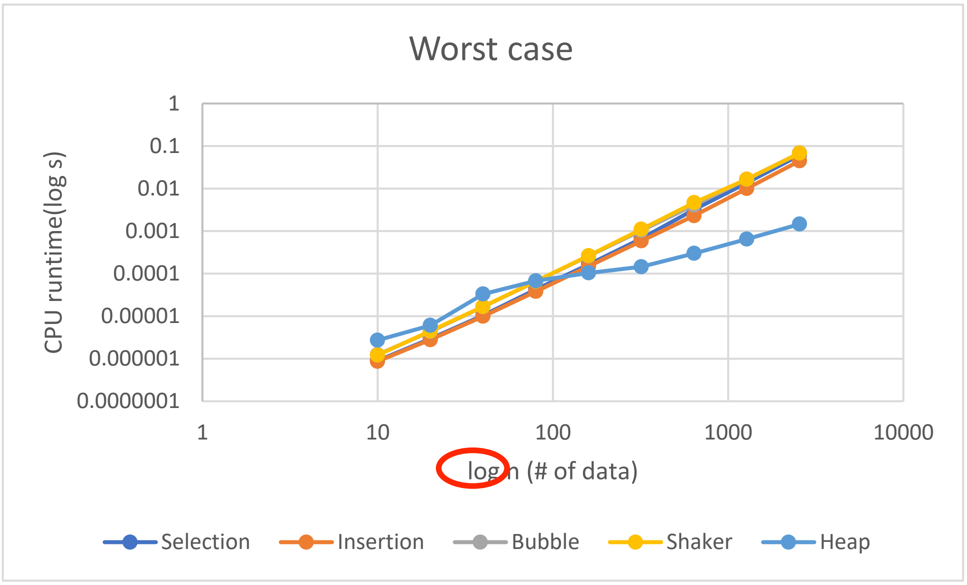
Workflow:

```
t = GetTime();                              // initialize time counter
ReverseSort(data, n) or Sort(data, n)       // sort data in advanced
for i := 0 to 500 do {
     ResetArray(data, list);            // reset array
```

```
        Sort(list, n);          // sort list
}
t = (GetTime() - t) / R;                // calculate CPU time / iteration
```

Results:

N(num)	Selection	Insertion	Bubble	Shaker	Heap
10	8.97884E-07	8.67844E-07	1.24598E-06	1.21403E-06	2.71988E-06
20	2.93779E-06	2.79427E-06	4.3478E-06	4.44794E-06	6.06966E-06
40	1.04218E-05	9.94396E-06	1.66359E-05	1.67542E-05	3.31621E-05
80	4.26297E-05	3.81022E-05	6.54101E-05	6.60701E-05	6.75364E-05
160	0.000164662	0.000148616	0.000259016	0.000263604	0.000103802
320	0.000668184	0.000586808	0.001030974	0.001107828	0.000145038
640	0.003207452	0.002300566	0.00408297	0.004652992	0.000298828
1280	0.01373781	0.01003129	0.01659725	0.01655017	0.000647842
2560	0.06098949	0.04472282	0.0681978	0.06767607	0.00145855



3. Best-case scenario:

For each case, the best-case scenario is when the array is completely ordered, but except Heap sort. For Heap sort, since it finds Max-heap first, the best-case of it should be reversed-alphabetically ordered. Therefore, I sort the data into reverse-alphabetically order and alphabetically order respectively, and implement on each sorting algorithms.

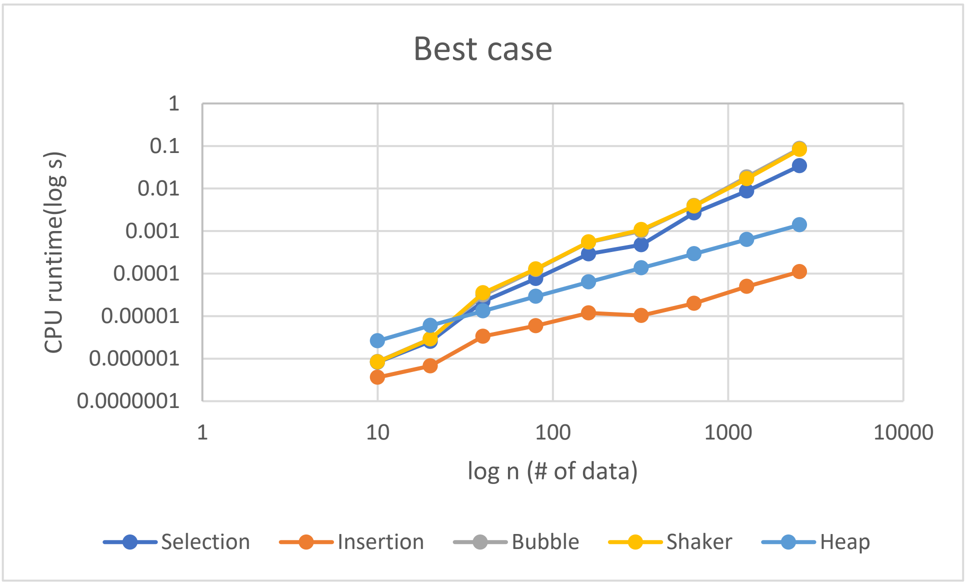
Workflow:

```
t = GetTime();                          // initialize time counter
ReverseSort(data, n) or Sort(data, n)     // sort data in advanced
```

```
for i := 0 to 500 do {
      ResetArray(data, list);            // reset array
      Sort(list, n);          // sort list
}
t = (GetTime() - t) / R;                // calculate CPU time / iteration
```

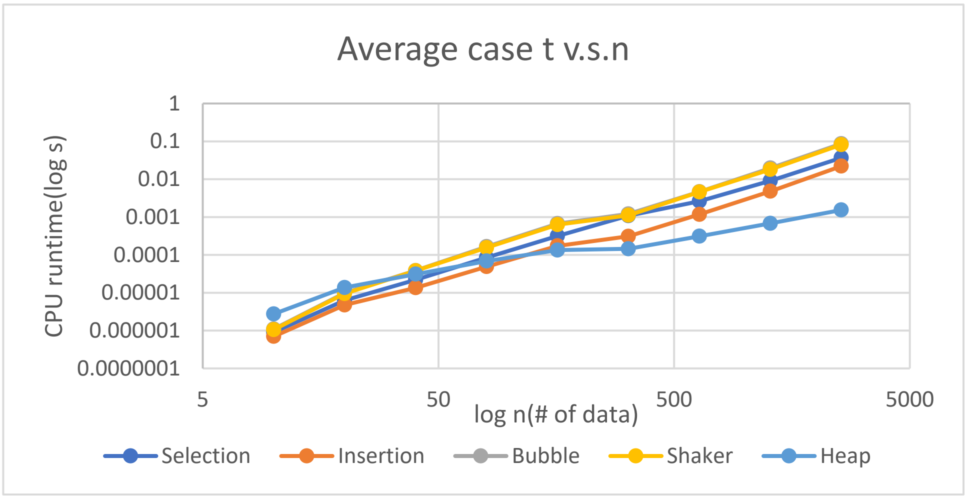
Results:

N(num)	Selection	Insertion	Bubble	Shaker	Heap
10	8.18253E-07	3.61919E-07	8.47817E-07	8.32081E-07	2.61402E-06
20	2.5301E-06	6.74248E-07	2.91014E-06	2.90394E-06	5.99384E-06
40	2.2212E-05	3.34597E-06	3.11379E-05	3.51663E-05	1.3206E-05
80	7.58638E-05	5.92184E-06	0.000124022	0.000128328	2.9078E-05
160	0.00028944	1.18179E-05	0.000543902	0.00055204	6.3376E-05
320	0.000474464	1.03164E-05	0.00100153	0.001077462	0.000135564
640	0.002657486	1.99537E-05	0.003952606	0.003869574	0.000290886
1280	0.00873215	4.97422E-05	0.01868315	0.01696503	0.000632004
2560	0.03405327	0.00011074	0.08690226	0.08247511	0.001394532



Observation:

1. Average-case comparisons:



According to the graph, **4 of the algorithms except Heap sort is O(n^2), while Heap sort's line is a little different than others.**

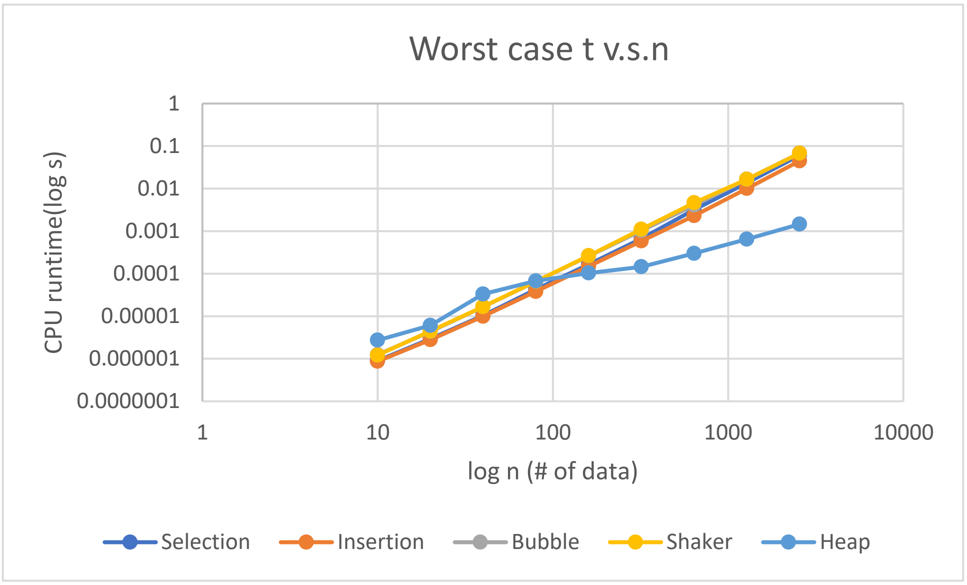
The analysis on Selection, Insertion, Bubble, Shaker sort is in hw01. I'll simply conclude the result: They have some differences, but they are roughly the same since the time complexities are O(n^2). And the speed from fast to slow is **Insertion > Selection > Shaker > Bubble.**

Heap sort, however, appears to be slower than all of them when n is small, and faster than all of them when n is big. The reason for the former statement is because heapify costs more procedures when finding max strings and heapify at each step, while others simply runs through each iterations. However, the difference between O(nlog(n)) and O(n^2) gradually increases when n becomes larger. That is why **Heap Sort is faster than all of them when n is large**.

There is some distortion between data, such as the 5th & 6th N's runtime on Heap sort is not logical. The reason behind this is maybe because the Workstation at NTHUEE is not stable and causes some faster/slower effects when implementing.

Overall, **Heap > Insertion > Selection > Shaker > Bubble.**

2. Worse-case comparisons:



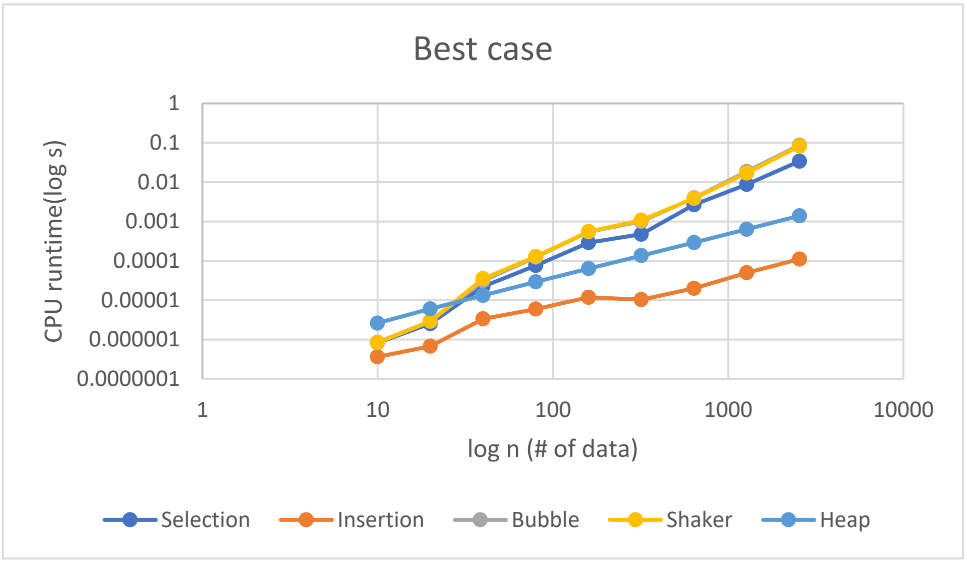
According to the graph, **4 of the algorithms except Heap sort is O(n^2), while Heap sort's line is a little different than others.**

The speed comparison except Heap sort is **Insertion > Selection > Shaker > Bubble,** from fast to slow**.** Since all of them are O(n^2) in time complexity, they are approximately the same. However, it did not meet my prediction. The reason why the steps of Selection sort is less than Insertion sort is because there were some overcounts in if statements. But overall, the result meets our expectation.

Heap sort, however, appears to be slower than all of them when n is small, and faster than all of them when n is big. The reason for it is same with the aforementioned analysis. That is why **Heap Sort is faster than all of them when n is large**.

Overall, **Heap > Insertion > Selection > Shaker > Bubble.**

3. Best-case comparisons:



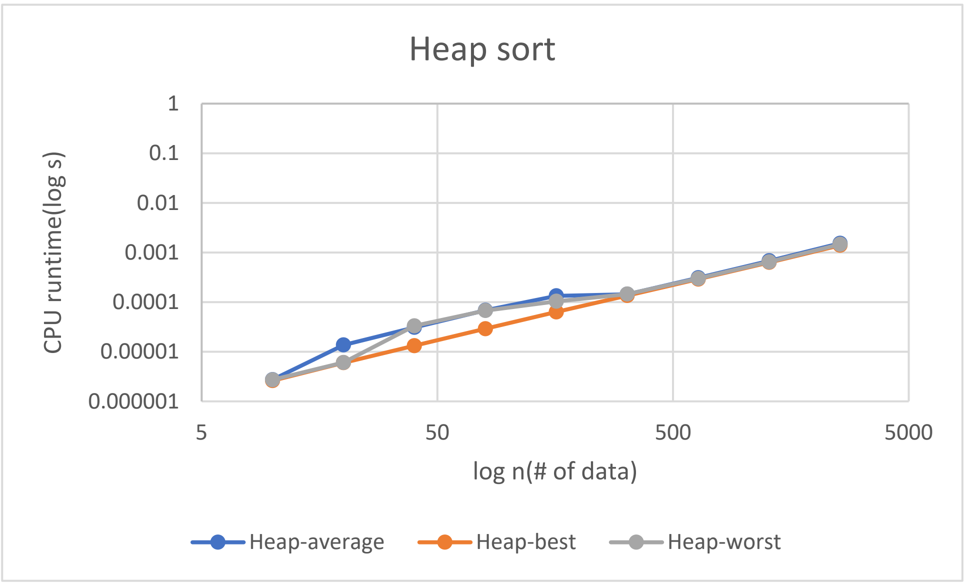
According to the graph, **Insertion Sort is the fastest, and Heap Sort follows, while the other three have no big difference.**

The speed of the best case, however, did not meet my prediction. The reason why behind this is unclear, since the only algorithm that does the swapping in best-case scenario is Selection Sort. There is only a reason that comes to my mind, is because **Insertion Sort is the only algorithm that does not do adding in the second layer iterations**. It only does the comparing. And another curious observation is that, the O(n) characteristic is not obvious. The only reason that I can think of is when EE workstation is executing the code, there's some effect cause by the number of input area. But overall, if we only look at the time complexity, the result meets our expectation.

If we keep out the Insertion sort, Heap sort also remain the properties that was discussed in the previous observation. **Heap Sort is faster than all of them when n is large**.

Overall, **Insertion > Heap > Selection > Shaker > Bubble.**

4. Heap Sort comparison:



Overall, the average, best, and worst case scenarios meets our expectation. **They are not so different regarding time complexity, O(nlog(n))**. If we look closely, the Worst case is slower than average case, and much more slower than best case.

5. Experiment problems:

There is some problem regarding Bubble sort and Shaker sort's worst/best case scenario. The best and worst case of them happen when input is sorted/completely wrong-ordered, obviously. However, the result still appears skeptical since the best case is longer than worst case. I've redone the implementation several times and got the same result. The reason behind this is still unclear and I can only think of the uncertainty of EE workstation.

Conclusion:

1. Time complexities of the 5 algorithms:

	Selection	Insertion	Bubble	Shaker	Heap
Best	O(n^2)	O(n)	O(n)	O(n)	O(nlog(n))
Worst	O(n^2)	O(n^2)	O(n^2)	O(n^2)	O(nlog(n))
Average	O(n^2)	O(n^2)	O(n^2)	O(n^2)	O(nlog(n))

2. Actual runtime comparison (Average):
   Heap > Insertion > Selection > Shaker > Bubble (> means runs faster)
3. Actual runtime comparison (Worst):
   Heap > Insertion > Selection > Shaker > Bubble (> means runs faster)
4. Actual runtime comparison (Best):
   Insertion > Heap > Selection > Shaker > Bubble (> means runs faster)
5. **Heap Sort is faster than all of them when n is large**.
6. **Heap sort's each cases are not so different regarding time complexity, O(nlog(n)).**
7. **Insertion Sort is the only algorithm that does not do adding in the second layer iterations**, thus having the fastest best-case runtime.

hw03.c

```c
// EE3980 HW03 Heap Sorts
// 106061146, Jhao-Ting, Chen
// 2020/03/27

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

int N;                                          // input size
char **data;                                    // input data
char **A;                                       // array to be sorted
int R = 500;                                    // number of repetitions

void readInput(void);                           // read all inputs
void printArray(char **A);                      // print the content of A
void copyArray(char **data, char **A);          // copy data to array A
double GetTime(void);                           // get local time in seconds
void SelectionSort(char **list, int n);         // in-place selection sort
void InvSelectionSort(char **list, int n);      // in-place selection sort
void InsertionSort(char **list, int n);         // in-place insertion sort
void BubbleSort(char **list, int n);            // in-place bubble sort
void ShakerSort(char **list, int n);            // in-place shaker sort
void Heapify(char **list, int i, int n);
void HeapSort(char **list, int n);

int main(void)
{
    int i;                                  // loop index
    double t;                               // for CPU time tracking

    readInput();                            // read input data

//  InvSelectionSort(data, N);              // execute inversed selection sort
//  printArray(data);
    t = GetTime();                          // initialize time counter
    for (i = 0; i < R; i++) {
        copyArray(data, A);                 // initialize array for sorting
//      SelectionSort(A, N);                // execute selection sort
//      InsertionSort(A, N);                // execute insertion sort
//      BubbleSort(A, N);                   // execute bubble sort
//      ShakerSort(A, N);                   // execute shaker sort
        HeapSort(A, N);                     // execute Heap sort
        if (i == 0) printArray(A);          // print sorted results
    }
    t = (GetTime() - t) / R;                // calculate CPU time / iteration
//  printf("Selection sort:\n  N = %d\n  CPU time = %e\n", N, t);   // result
//  printf("Insertion sort:\n  N = %d\n  CPU time = %e\n", N, t);   // result
//  printf("Bubble sort:\n  N = %d\n  CPU time = %e\n", N, t);  // result
//  printf("Shaker sort:\n  N = %d\n  CPU time = %e\n", N, t);  // result
    printf("Heap sort:\n  N = %d\n  CPU time = %e\n", N, t);    // result

    return 0;
}

void readInput(void)                            // read all inputs
{
    int i, j, len;                              // for looping and dynamic store
    char str[50];                               // for dynamic store

    scanf("%d", &N);                            // read N for # of words
    data = (char **)calloc(N, sizeof(char *));  // initialize data size (N)
    A = (char **)calloc(N, sizeof(char *));     // initialize A size (N)

    for (i = 0; i < N; i++) {
        scanf("%s", str);                       // store each word in data
        len = strlen(str);
        data[i] = (char *)calloc(len, sizeof(char));
        A[i] = (char *)calloc(len, sizeof(char));
        for (j = 0; j < len; j++)  {
            data[i][j] = str[j];
            A[i][j] = str[j];
        }
    }
}

void printArray(char **A)       // print the content of array A
{
    int i;                                  // for looping
    for (i = 0; i < N; i++) {
        printf("%s\n", A[i]);               // print each string
    }

}

void copyArray(char **data, char **A)    // copy data to array A
{
    int i;                                  // for looping

    for (i = 0; i < N; i++) {
        A[i] = data[i];                     // copy from data to A
    }
}

double GetTime(void)                        // demonstration code from 1.1.3
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    return tv.tv_sec + 1e-6 * tv.tv_usec;
}

void SelectionSort(char **list, int n) // in-place selection sort
{
    int i, j, k;                        // for looping
    char *tmp = (char *)calloc(50, sizeof(char));   // initialize tmp size
    for (i = 0; i < N; i++) {               // for every A[i]
        j = i;                              // Initialize j to i
        for (k = i + 1; k < N; k++) {   // search for smallest in A[i+1 : n]
            if (strcmp(list[k], list[j]) < 0) { // found, store in j
                j = k;
            }
        }
        tmp = list[i];                      // swap A[i] & A[j]
        list[i] = list[j];
        list[j] = tmp;
    }
}

void InsertionSort(char **list, int n) // in-place insertion sort
{
    int i, j;
    char *tmp = (char *)calloc(50, sizeof(char));   // initialize tmp size
    for (j = 1; j < N; j++) {               // Assume A[0: j-1] already sorted
        tmp = list[j];                      // Move A[j] to its proper place
        i = j - 1;                          // Init i to be j-1
        while ((i >= 0) && (strcmp(tmp, list[i]) < 0)) {//Find i for A[i]<=A[j]
        while ((i >= 0) && (strcmp(tmp, list[i]) < 0)) { // Find i for A[i]<=A[j
]
            list[i + 1] = list[i];          // Move A[i] up by one position
            i = i - 1;
        }
        A[i + 1] = tmp;                     // Move A[j] to A[i+1]
    }
}

void BubbleSort(char **list, int n) // in-place bubble sort
{
    int i, j;
    char *tmp = (char *)calloc(50, sizeof(char));   // Initialize tmp size
    for (i = 0; i < N - 1; i++) {           // Find the smallest item for A[i]
        for (j = N - 1; j > i; j--) {
            if (strcmp(list[j], list[j - 1]) < 0) { // swap A[j] & A[j-1]
                tmp = list[j];
                list[j] = list[j - 1];
                list[j - 1] = tmp;
            }
        }
    }
}

void ShakerSort(char **list, int n)  // in-place shaker sort
{
    int l = 0, r = N-1, j;
    char *tmp = (char *)calloc(50, sizeof(char));   // Initialize tmp size
    while (l <= r) {
        for (j = r; j >= l+1; j--) {        // Element exchange from r down to l
        for (j = r; j >= l + 1; j--) {        // Element exchange from r down to
 l
            if (strcmp(list[j], list[j - 1]) < 0) { // swap A[j] & A[j-1]
                tmp = list[j];
                list[j] = list[j - 1];
                list[j - 1] = tmp;
            }
        }
        l++;
        for (j = l; j <= r-1; j++) {        // Element exchange from l to r
        for (j = l; j <= r - 1; j++) {        // Element exchange from l to r
            if (strcmp(list[j], list[j + 1]) > 0) { // swap A[j] and A[j+1]
                tmp = list[j];
                list[j] = list[j + 1];
                list[j + 1] = tmp;
            }
        }
        r--;
    }
}

void Heapify(char **list, int i, int n)
{
    int j, done;
    char *tmp = (char *)calloc(50, sizeof(char));   // Initialize tmp size

    j = 2 * i;              // list[j] is the lchild
    tmp = list[i - 1];
    done = 0;

    while (((j - 1) < n) && (!done)) {      // list[j] is the rchild
        if (((j - 1) < n - 1) && (strcmp(list[j - 1], list[j]) < 0)) {
            j = j + 1;                      // list[j - 1] is the larger child
        }
        if (strcmp(tmp, list[j - 1]) > 0) { // If larger than children, done
            done = 1;
        } else {                            // Otherwise, continue
            list[(j / 2) - 1] = list[j - 1];
            j = 2 * j;
        }
    }
```

```c
    list[(j / 2) - 1] = tmp;
}

void HeapSort(char **list, int n)
{
    int i;                                          // for looping
    char *tmp = (char *)calloc(50, sizeof(char));   // Initialize tmp size

    for (i = (n / 2); i >= 1; i--) {    // Initialize **list to be a max heap
        Heapify(list, i, n);
    }
    for (i = n; i >= 2; i--) {      // Repeat n-1 times
        tmp = list[i - 1];          // Move maximum to the end
        list[i - 1] = list[0];
        list[0] = tmp;
        Heapify(list, 1, i - 1);    // Then make list[1: i-1] a max heap
    }
}

void InvSelectionSort(char **list, int n) // in-place selection sort
{
    int i, j, k;                        // for looping
    char *tmp = (char *)calloc(50, sizeof(char));   // initialize tmp size
    for (i = 0; i < N; i++) {               // for every A[i]
        j = i;                              // Initialize j to i
        for (k = i + 1; k < N; k++) {   // search for biggest in A[i+1 : n]
            if (strcmp(list[k], list[j]) > 0){  // found, store in j
            if (strcmp(list[k], list[j]) > 0) {  // found, store in j
                j = k;
            }
        }
        tmp = list[i];                      // swap A[i] & A[j]
        list[i] = list[j];
        list[j] = tmp;
    }
}
```

[Program Format] can be improved.
[Max] heap should be described before heapSort.
[Worst-case] CPU time measurement for RDsearch has extra operations and hence not accurate.
[Bubble] sort and Shaker sort both need n^2 string comparisons and thus the time complexity is $O(n^2)$ in all cases.
[Space] complexity O(1)?
[Resulting] tables and figures can be improved.
[Report] needs to have double line space.
[Report] is getting better. Please continue to improve upon it.

Score: 79