# EE3980 Algorithms

# HW2 Heap Sort

**104061212 馮立俞**

**2017/3/18**

## I. Introduction

Continuing HW01, in which we analyzed the efficiency of **Insertion Sort**, **Selection Sort**, and **Bubble Sort**. In this assignment, another sorting algorithm, **Heap Sort** is implemented. Additionally, we'll explore how efficiency change for each algorithm when they encounter best case or worst case situations.

## II. Approach & Analysis

### Heap Sort

```
1.  Algorithm HeapSort(A, n)  // Sort A[1 : n] into nondecreasing order.
2.  {
3.          for i: = ⌊n / 2⌋ to 1 step- 1 do // Init A[1 : n] to be a max heap.
4.               Heapify(A, i, n);
5.          for i: = n to 2 step- 1 do {              // Move maximum to the end.
6.                  t: = A[i];A[i] = A[1];A[1] = t; // Then make A[1 : i-1] a max
7.                                                 //heap.
8.                  Heapify(A, 1, i- 1);
9.          }
10. }
```

```
1.  Algorithm Heapify(A, i, n) // To maintain max heap property for the tree with
                               // root A[i ]. The size of A is n.
2.  {
3.          j: = 2× i;item: = A[i];done: = false; // A[2 × i ] is the lchild.
4.          while ((j _ n) and(not done))
5.              do { // A[2 × i + 1] is the rchild.
6.                  if ((j < n) and(A[j] < A[j + 1])) then j: = j + 1;
7.                  if (item > A[j]) then done: = true;
8.                        // If larger than children, done.
9.                  else { // Otherwise, continue.
10.                     A[⌊j / 2⌋]: = A[j];
11.                     j: = 2× j;
12.                 }
13.             }
14.         A[⌊j / 2⌋]: = item;
15. }
```

To analyze **Heap Sort**'s efficiency, we need to look at **Heapify**.

The **Heapify** function has $O(\log n)$ in worst case, and $\Omega(1)$ in best case.

The worst case occurs when the root should make its way to the leaf level, i.e. from level 1 to level $\log_2 n$ .

The best case, on the contrary, occurs when the root is already the maximum of the given tree. Only initialization and one comparison is needed in such case.

Therefore, in the process of **Heap Sort**, the best case occurs when the to-be-heapified array is already a max heap, for which its roots barely need to move. ( $\Theta(n)$ )And the worst case occurs when the input is a min-heap. Almost all members would move up along the heap in the first loop of **HeapSort**

Function.( $\Theta(n \log n)$ )

**EDIT: Though an already heapified array can save us some effort during initialization (line 3, 4 in **HeapSort**), there's no difference in efficiency for the rest of the algorithm.

Recap: **Insertion Sort**

```
1.  void InsertionSort(char ** list, int n) {
2.      int i, j;
3.      char * temp;
4.      for (j = 1; j < n; j++) {
5.          temp = list[j];
6.          i = j - 1;
7.          while ((i >= 0) && (strcmp(temp, list[i]) < 0)) {
8.              list[i + 1] = list[i];
9.              i--;
10.         }
11.         list[i + 1] = temp;
12.     }
13. }
```

We can obviously see that from the while loop that the best case for **Insertion Sort** happens when the initial sequence is (non-decreasingly in this case) ordered. The complexity can be reduced to $\Theta(n)$ in best case.

**Selection Sort** and **Bubble Sort** performs $\frac{n(n-1)}{2}$ times of operation regardless of the input. There's no best or worst case for them. Though an ordered sequence could save some swapping for **Bubble Sort,** and save a few variable assignment for **Selection Sort**. The coefficient of the time complexity of two algorithms are reduced,

yet are still $\Theta(n^2)$.

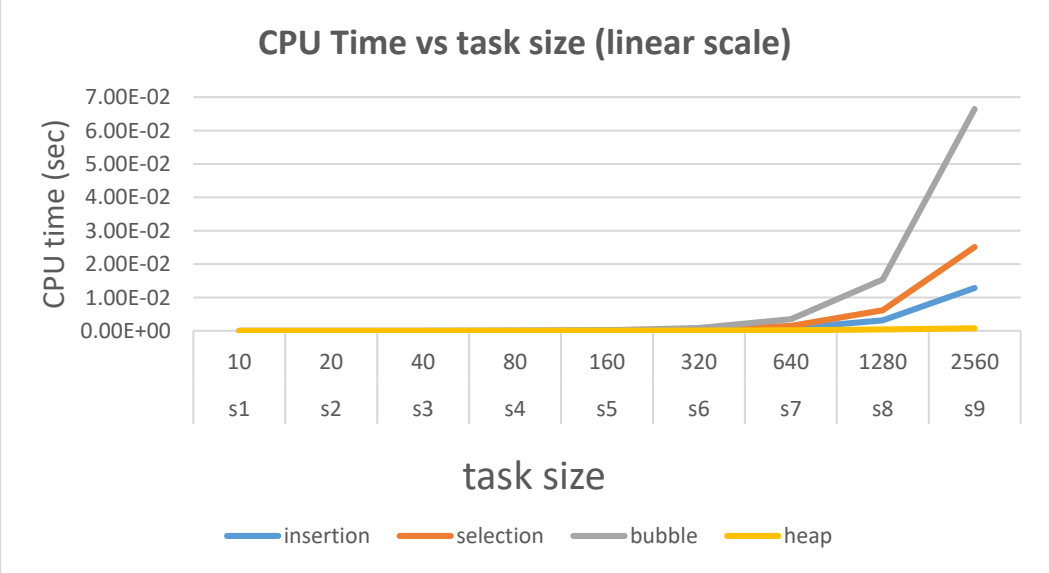
For all algorithms above ,nonetheless , their space complexity are all $\Theta(n)$.

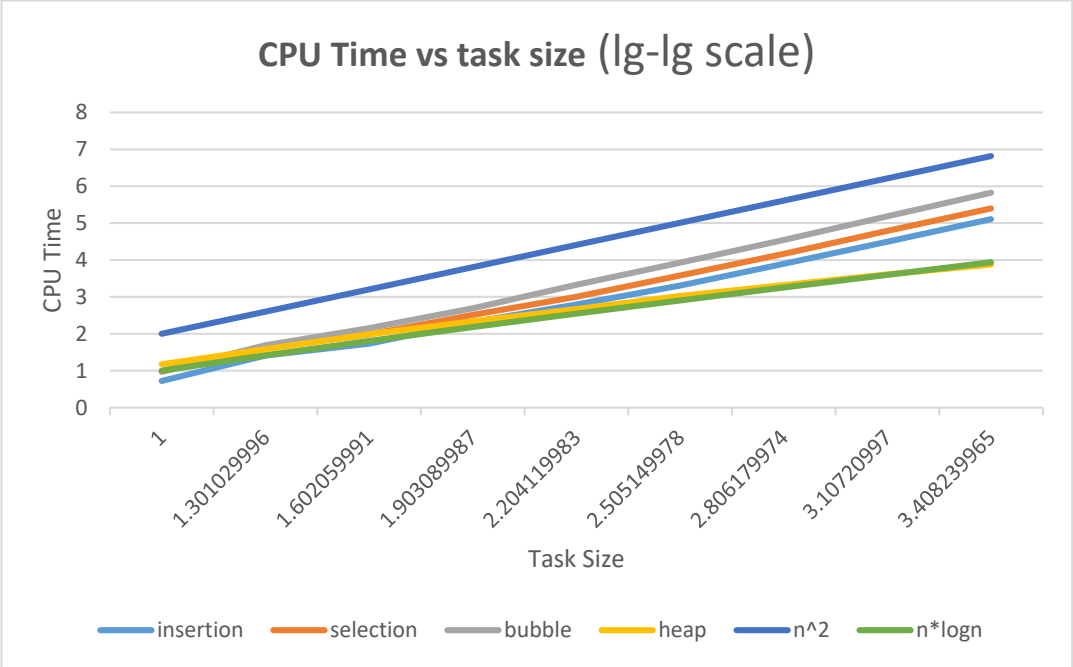
## III. Results & Analysis

The Below table shows the execution time of different sorting algorithm w.r.t. different size of tasks.

| Task Size | insertion | selection | bubble | heap |
|---|---|---|---|---|
| 10 | 5.28E-07 | 9.38E-07 | 9.78E-07 | 1.48E-06 |
| 20 | 2.57E-06 | 4.06E-06 | 4.87E-06 | 3.84E-06 |
| 40 | 5.37E-06 | 9.60E-06 | 1.43E-05 | 9.70E-06 |
| 80 | 2.07E-05 | 3.27E-05 | 4.90E-05 | 2.18E-05 |
| 160 | 6.23E-05 | 1.01E-04 | 2.14E-04 | 4.67E-05 |
| 320 | 2.04E-04 | 3.79E-04 | 8.51E-04 | 1.06E-04 |
| 640 | 7.90E-04 | 1.47E-03 | 3.51E-03 | 2.10E-04 |
| 1280 | 3.15E-03 | 6.18E-03 | 1.54E-02 | 4.10E-04 |
| 2560 | 1.28E-02 | 2.51E-02 | 6.65E-02 | 7.59E-04 |

To visualize the above table, we can plot as following

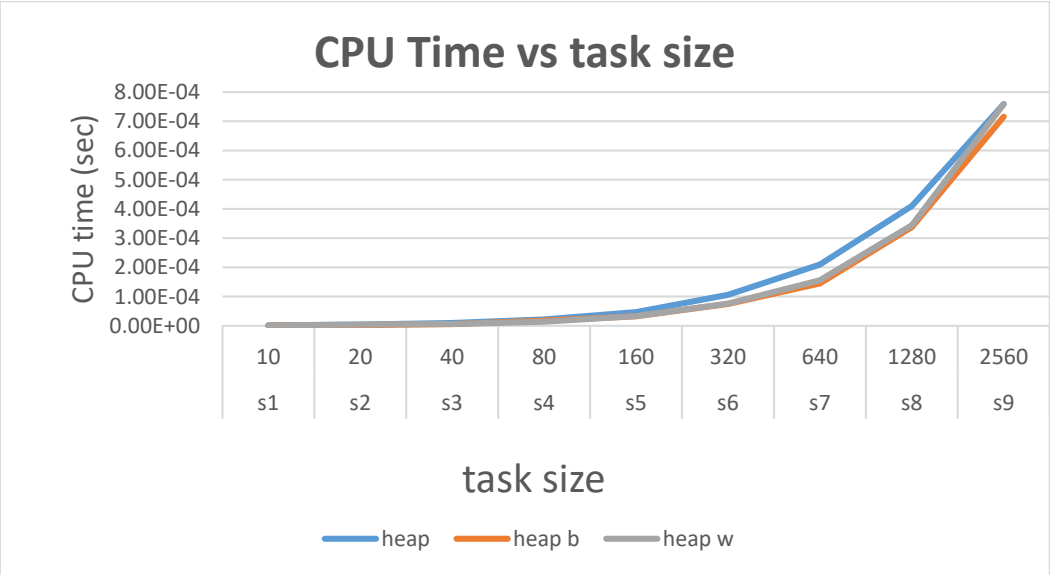





We can see from the above graph that Heap Sort, unlike the other three, grows with `nlogn`.

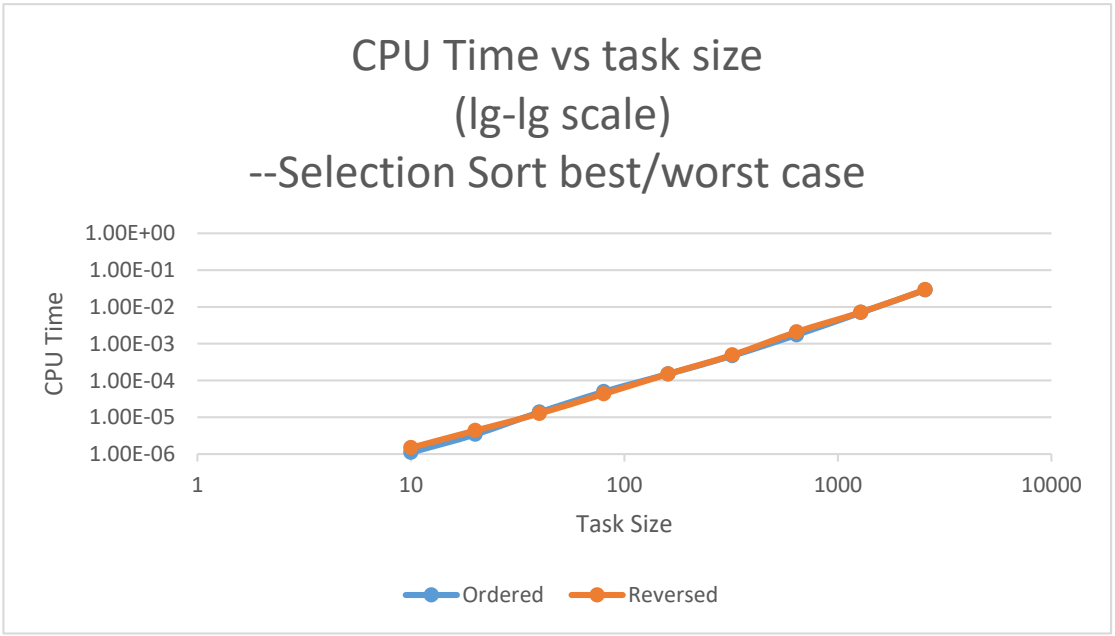
Next, we'll show the performance difference in best or worst case for each algorithm.

**Heap Sort**

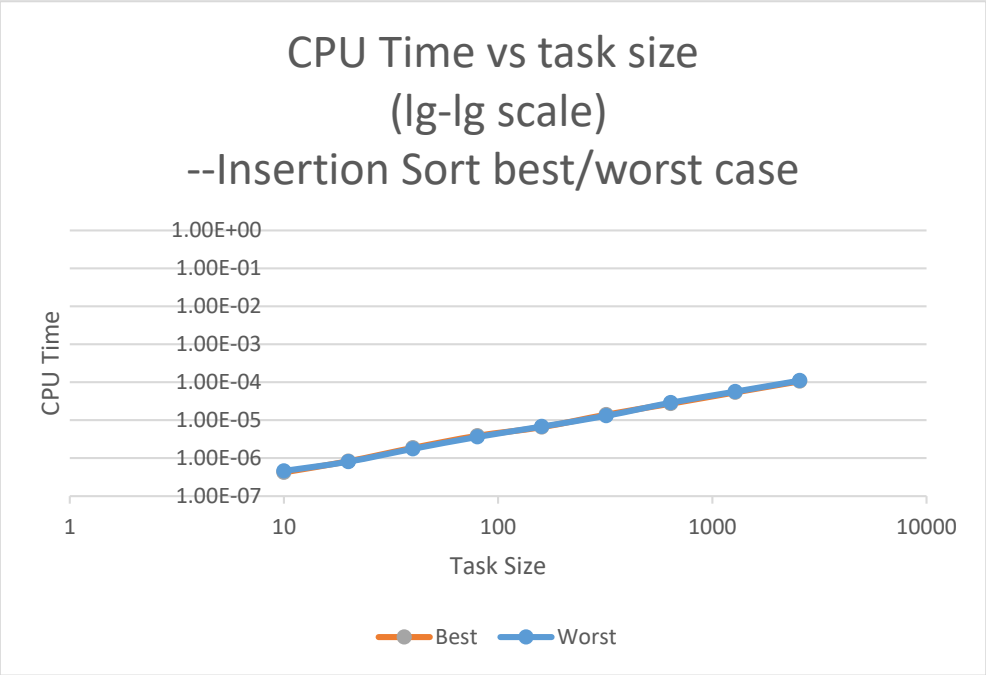



Best/ Worst case difference wasn't observed clearly though. The difference was not noticeable for the remaining three algorithms, either.

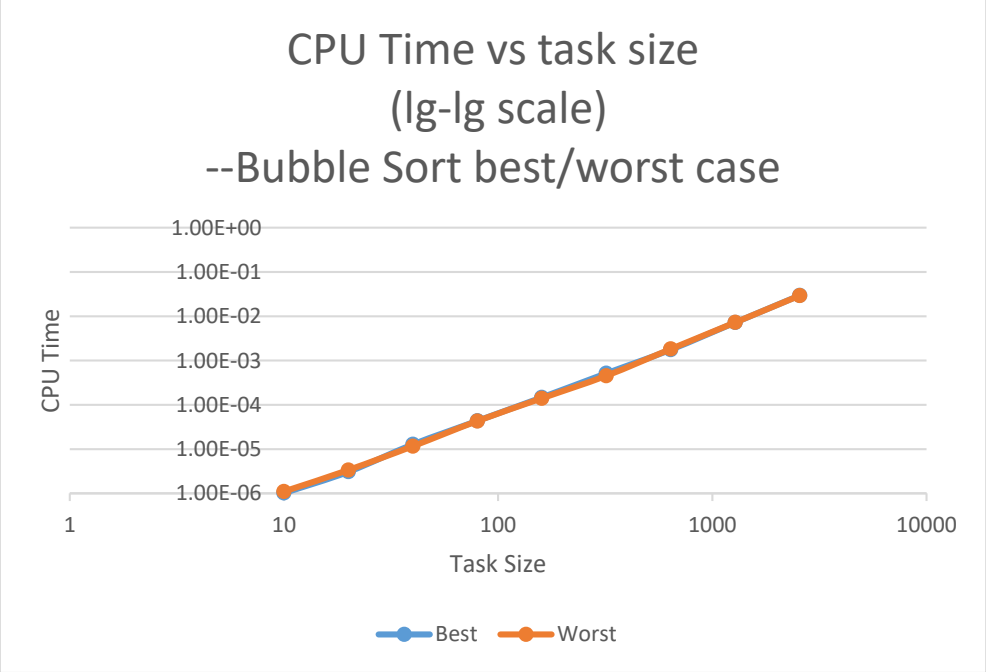
**Selection Sort**



**Insertion Sort**




**Bubble Sort**



The above plots are based on the following tables, for reference.

Selection Sort

| | Ordered | Reversed |
|---|---|---|
| 10 | 1.10E-06 | 1.48E-06 |
| 20 | 3.41E-06 | 4.34E-06 |
| 40 | 1.37E-05 | 1.26E-05 |
| 80 | 4.94E-05 | 4.30E-05 |
| 160 | 1.50E-04 | 1.50E-04 |
| 320 | 4.71E-04 | 4.93E-04 |
| 640 | 1.73E-03 | 2.09E-03 |
| 1280 | 7.00E-03 | 7.18E-03 |
| 2560 | 2.87E-02 | 2.94E-02 |

Insertion Sort

| | Best | Worst |
|---|---|---|
| 10 | 4.20E-07 | 4.58E-07 |
| 20 | 8.24E-07 | 8.04E-07 |
| 40 | 1.86E-06 | 1.75E-06 |
| 80 | 3.87E-06 | 3.60E-06 |
| 160 | 6.54E-06 | 6.77E-06 |
| 320 | 1.40E-05 | 1.30E-05 |
| 640 | 2.72E-05 | 2.88E-05 |
| 1280 | 5.40E-05 | 5.64E-05 |
| 2560 | 1.06E-04 | 1.10E-04 |

Bubble Sort

| | Best | Worst |
|---|---|---|
| 10 | 1.02E-06 | 1.10E-06 |
| 20 | 3.08E-06 | 3.36E-06 |
| 40 | 1.28E-05 | 1.16E-05 |
| 80 | 4.35E-05 | 4.30E-05 |
| 160 | 1.47E-04 | 1.40E-04 |
| 320 | 5.14E-04 | 4.52E-04 |
| 640 | 1.76E-03 | 1.82E-03 |
| 1280 | 7.22E-03 | 7.26E-03 |
| 2560 | 2.95E-02 | 2.93E-02 |