EE3980 Algorithms

Hw03 Heap Sort

106061146 陳兆廷

**Introduction:**

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 analyzation.

**Analysis:**

1. **Selection Sort:**
2. 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.

1. Algorithm:
2. // Sort the array A[1 : n ] into nondecreasing order.
3. // Input: A[1 : n], int n
4. // Output: A, A[i] <= A[j] if i < j.
5. Algorithm SelectionSort(A, n)
6. {
7. **for** i := 1 to n **do** {
8. j := i;
9. **for** k := i + 1 to n **do** {
10. **if** (A[k] < A[j]) then j = k
11. }
12. tmp = list[i]; list[i] = list[j]; list[j] = tmp;
13. }
14. }
15. 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.

1. Time complexity:

|  |  |  |  |
| --- | --- | --- | --- |
|  | s/e | freq | total |
| 1. Algorithm SelectionSort(A, n) 2. { 3. **for** i := 1 to n **do** { 4. j := i; 5. **for** k := i + 1 to n **do** { 6. **if** (A[k] < A[j]) then j = k 7. } 8. tmp = list[i]; list[i] = list[j]; list[j] = tmp; 9. } 10. } | 0  0  n  1  1  1  0  3  0  0 | 0  0  1  n - 1  c(n-1)  c(n-1)  0  n – 1  0  0 | 0  0  n  n - 1  c(n-1)  c(n-1)  0  3n-3  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, since the swapping still execute. And the time complexity would be **O(n)**.

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)**.

1. Space Complexity:

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

1. **Insertion Sort:**
2. Abstract:

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

1. Algorithm:
2. // Sort A[1 : n] into nondecreasing order.
3. // Input: array A, int n
4. // Output: array A sorted.
5. Algorithm InsertionSort(A, n)
6. {
7. **for** j := 2 to n **do** { // Assume A[1 : j − 1] already sorted.
8. item := A[j ] ; // Move A[j ] to its proper place.
9. i := j − 1 ; // Init i to be j − 1.
10. **while** ((i ≥ 1) and (item < A[i])) **do** { // Find i s.t. A[i] ≤ A[j].
11. A[i + 1] := A[i] ; // Move A[i] up by one position.
12. i := i − 1 ;
13. }
14. A[i + 1] = item ; // Move A[j ] to A[i + 1].
15. }
16. }
17. 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.

1. Time complexity:

|  |  |  |  |
| --- | --- | --- | --- |
|  | s/e | freq | total |
| 1. Algorithm InsertionSort(A, n) 2. { 3. **for** j := 2 to n **do** { 4. item := A[j ] ; 5. i := j − 1 ; . 6. **while** ((i ≥ 1) and (item < A[i])) **do** { 7. A[i + 1] := A[i] ; 8. i := i − 1 ; 9. } 10. A[i + 1] = item ; 11. } 12. } | 0  0  n-1  1  1  2c  1  1  0  1  0  0 | 0  0  1  n-2  n-2  n-2  c(n-2)  c(n-2)  c(n-2)  n-2  0  0 | 0  0  n-1  n-2  n-2  2cn-4c  cn-2n  cn-2n  0  n-2  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:

When the array is completely sorted at the beginning, the steps would be n - 1, since the swapping still execute. 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)**.

1. 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).

1. **Bubble Sort:**
2. Abstract:

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

1. Algorithm:
2. // Sort A[1 : n] into nondecreasing order.
3. // Input: array A, int n
4. // Output: array A sorted.
5. Algorithm BubbleSort(A, n)
6. {
7. **for** i := 1 to n − 1 **do** { // Find the smallest item for A[i].
8. **for** j := n to i + 1 step −1 **do** {
9. **if** (A[j ] < A[j − 1]) { // Swap A[j ] and A[j − 1].
10. tmp = A[j ] ; A[j ] = A[j − 1] ; A[j − 1] = tmp;
11. }
12. }
13. }
14. }
15. 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.

1. Time complexity:

|  |  |  |  |
| --- | --- | --- | --- |
|  | s/e | freq | total |
| 1. Algorithm BubbleSort(A, n) 2. { 3. **for** i := 1 to n − 1 **do** { 4. **for** j := n to i + 1 step −1 **do** { 5. **if** (A[j ] < A[j − 1]) { 6. tmp = A[j]; 7. A[j] = A[j − 1]; 8. A[j − 1] = tmp; 9. } 10. } 11. } 12. } | 0  0  n-1  c  1  1  1  1  0  0  0  0 | 0  0  1  n-1  c(n-1)  c(n-1)  c(n-1)  c(n-1)  0  0  0  0 | 0  0  n-1  c(n-1)  c(n-1)  c(n-1)  c(n-1)  c(n-1)  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. 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)**.

1. Space Complexity:

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

1. **Shaker Sort:**
2. 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.

1. Algorithm:
2. // Sort A[1 : n] into nondecreasing order.
3. // Input: array A, int n
4. // Output: array A sorted.
5. Algorithm ShakerSort(A, n)
6. {
7. ℓ := 1 ; r := n ;
8. **while** ℓ ≤ r **do** {
9. **for** j := r to ℓ + 1 step −1 **do** { // Element exchange from r down to ℓ
10. **if** (A[j ] < A[j − 1]) { // Swap A[j ] and A[j − 1].
11. t = A[j ] ; A[j ] = A[j − 1] ; A[j − 1] = t;
12. }
13. }
14. ℓ := ℓ + 1 ;
15. **for** j := ℓ to r − 1 **do** { // Element exchange from ℓ to r
16. **if** (A[j ] > A[j + 1]) { // Swap A[j ] and A[j + 1].
17. t = A[j ] ; A[j ] = A[j + 1] ; A[j + 1] = t;
18. }
19. }
20. r := r − 1 ;
21. }
22. }
23. 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 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 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.

1. Time complexity:

|  |  |  |  |
| --- | --- | --- | --- |
|  | s/e | freq | total |
| 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** { 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 := ℓ to r − 1 **do** { 12. **if** (A[j ] > A[j + 1]) { 13. t = A[j ] ; A[j ] = A[j + 1] ; A[j + 1] = t; 14. } 15. } 16. r := r − 1 ; 17. } 18. } | 0  0  2  n/2  c  1  3  0  0  1  c  1  3  0  0  1  0  0 | 0  0  1  1  n/2  cn/2  cn/2  n/2  n/2  n/2  n/2  cn/2  cn/2  n/2  n/2  n/2  0  0 | 0  0  2  n/2  cn/2  cn/2  3cn/2  0  0  n/2  cn/2  cn/2  3cn/2  0  0  n/2  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 swapping still execute. 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)**.

1. Space Complexity:

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

1. **Heap Sort:**
2. Abstract:

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

1. Algorithm:
2. // Sort A[1 : n] into nondecreasing order.
3. // Input: array A, int n
4. // Output: array A sorted.
5. Algorithm InsertionSort(A, n)
6. {
7. **for** j := 2 to n **do** { // Assume A[1 : j − 1] already sorted.
8. item := A[j ] ; // Move A[j ] to its proper place.
9. i := j − 1 ; // Init i to be j − 1.
10. **while** ((i ≥ 1) and (item < A[i])) **do** { // Find i s.t. A[i] ≤ A[j].
11. A[i + 1] := A[i] ; // Move A[i] up by one position.
12. i := i − 1 ;
13. }
14. A[i + 1] = item ; // Move A[j ] to A[i + 1].
15. }
16. }
17. 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.

1. Time complexity:

|  |  |  |  |
| --- | --- | --- | --- |
|  | s/e | freq | total |
| 1. Algorithm InsertionSort(A, n) 2. { 3. **for** j := 2 to n **do** { 4. item := A[j ] ; 5. i := j − 1 ; . 6. **while** ((i ≥ 1) and (item < A[i])) **do** { 7. A[i + 1] := A[i] ; 8. i := i − 1 ; 9. } 10. A[i + 1] = item ; 11. } 12. } | 0  0  n-1  1  1  2c  1  1  0  1  0  0 | 0  0  1  n-2  n-2  n-2  c(n-2)  c(n-2)  c(n-2)  n-2  0  0 | 0  0  n-1  n-2  n-2  2cn-4c  cn-2n  cn-2n  0  n-2  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:

When the array is completely sorted at the beginning, the steps would be n - 1, since the swapping still execute. 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)**.

1. Space Complexity:

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

1. **Comparison:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Linear | | Bidirectional | | Random-directional | |
| Best | 1 | O(1) | 1 | O(1) | 1 | O(1) |
| Worst | 2n+2 | O(n) | 3/2n+2 | O(n) | 2n+3 | O(n) |
| Average | n+2 | O(n) | 3/4n+2 | O(n) | n+3 | O(n) |

**Average Speed (fast>slow): BDSearch > LinearSearch > RDSearch.**

**Worst-case Speed (fast>slow): BDSearch > LinearSearch ~= RDSearch.**

**Implementation:**

1. **Average-case scenario:**

Since the most average case of testing the algorithm is randomly pick an item and search, I pick an item in given data randomly, implement on the algorithm, and repeat the procedure 500 times. The mean of the recorded CPU time between those steps are the average CPU runtime of the algorithm.

Workflow:

1. t = GetTime();                          // initialize time counter
2. **for** i := 0 to 500 **do** {
3. random\_num = RandomNumber(N);     // generate a random number
4. Search(random\_num);         // search item accordingly
5. }
6. t = (GetTime() - t) / 500;                // calculate CPU time / iteration

Results:

|  |  |  |  |
| --- | --- | --- | --- |
| N(num) | Linear(s) | Bidirectional(s) | Random-directional(s) |
| 10 | 2.00272E-07 | 1.83582E-07 | 2.34127E-07 |
| 20 | 2.57969E-07 | 2.52247E-07 | 4.95911E-07 |
| 40 | 3.84331E-07 | 3.63827E-07 | 4.42028E-07 |
| 80 | 6.46114E-07 | 6.0606E-07 | 7.10011E-07 |
| 160 | 1.30606E-06 | 1.08576E-06 | 1.28174E-06 |
| 320 | 2.27022E-06 | 2.04182E-06 | 2.33221E-06 |
| 640 | 4.50611E-06 | 3.78609E-06 | 4.53568E-06 |
| 1280 | 8.42619E-06 | 7.39813E-06 | 8.87442E-06 |
| 2560 | 1.682E-05 | 1.57599E-05 | 1.78042E-05 |

1. **Worst-case scenario:**

For Linear Search, the worst case will happen when the item for searching is at the end of the array. Therefore I choose to **search data[N - 1]** as the worst-case condition.

For Bidirectional Search, the worst case will happen when the item for searching is at the middle of the array. Therefore, I choose to **search data[N/2]** as the worst-case condition.

For Bidirectional Search, the worst case will happen when the item for searching is at the front or end of the array, depending on the random number algorithm generates. But, there’s no way for me to set the item for searching according to the random number generated by algorithm without altering the algorithm itself. Therefore, I choose to **search data[N/2],** as the worst-case condition.

Since the worst-case scenario must compare with average case, I generate a random number in this workflow, too. This way, they are compared with each other in the same condition.

Workflow:

1. t = GetTime();                          // initialize time counter
2. **for** i := 0 to 500 **do** {
3. random\_num = RandomNumber(N);          // generate a random number
4. Search(n);     // search for item (list[N])
5. }
6. t = (GetTime() - t) / R;                // calculate CPU time / iteration

Results:

|  |  |  |  |
| --- | --- | --- | --- |
| N(num) | Linear(s) | Bidirectional(s) | Random-directional(s) |
| 10 | 2.39849E-07 | 2.26021E-07 | 1.99795E-07 |
| 20 | 3.85761E-07 | 3.48091E-07 | 2.89917E-07 |
| 40 | 6.38008E-07 | 5.8794E-07 | 4.26292E-07 |
| 80 | 1.16205E-06 | 1.0643E-06 | 6.97613E-07 |
| 160 | 2.2378E-06 | 1.98603E-06 | 1.266E-06 |
| 320 | 4.34637E-06 | 3.96013E-06 | 2.36225E-06 |
| 640 | 8.57401E-06 | 7.61604E-06 | 4.54807E-06 |
| 1280 | 1.69978E-05 | 1.50738E-05 | 8.9159E-06 |
| 2560 | 3.3946E-05 | 3.00579E-05 | 1.76859E-05 |

**Observation:**

1. Average-case v.s. Worst-case comparisons:

|  |  |  |
| --- | --- | --- |
| Linear | Bidirectional | Random-directional |
|  |  |  |

p.s. The x-axis is log(# of data), y-axis is CPU runtime. The scale of three charts are the same, y: from 0 to 0.00004, x: 1 to 10000.

For Linear Search and Bidirectional Search, **the Worst-case scenario has a longer CPU runtime.** This indicates that the experiments between average case and worst case is valid. However, the result of Random-directional Search did not seem valid since the worst-case scenario should spend longer time than average-case. The reason of this is because the worst-case of this is to pick the target in the middle, and this seem like an average case, too. Therefore, there’s no big changes between average and worst case of RDSearch.

I did one more experience, I gave RDSearch list[0] or list[n-1] to search, depending on the direction it generated. This way, I can **simulate the true worst-case**, which would happen if it’s an unlucky day. The spent time of this **worst-case scenario is much longer than the original result**.

1. Average-case comparisons:

According to the graph in **Implementation**, **all 3 algorithms’ time complexities are clearly O(n), since they are linear.**

Furthermore, according to this graph, we can conclude the speeds of the 3 algorithms, is **BDSearch > Linear Search > RDSearch (> means faster than)**. This result exactly meets my prediction. However, based on the results and the analysis, there’s only a slightly difference between the speeds of the three algorithms.

The reason why Bidirectional search is the fastest, I think, is because: **having half of the iterations but contains double amount of statements (BDSearch) could run faster than full iterations with single statement (Linear Search)**. Bidirectional search runs through only n/2 iterations while Linear search runs though n iterations.

The reason why Random-directional search is the slowest, I think, is because it contains a random number generator and some if/else in it. Therefore, it is the slowest in the average case.

1. Worse-case comparisons:

According to the graph in **Implementation**, the **worst-case scenario fits our calculation of time complexity, which are all O(n)**.

The **worst case of Random-directional Search is much faster**. This is because we cannot make sure that which direction it will go, so we can only assign a target that is in the middle of the array for the algorithm to find. It is no different than the average-case.

I’ve done another worst-case scenario for Random-directional search, which is choosing the worst case depending on the random-generated direction. The result fits the observation in Average-case part and my calculation.

**Conclusion:**

1. Time complexities of the 3 algorithms:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linear | Bidirectional | Random-directional |
| Best | O(1) | O(1) | O(1) |
| Worst | O(n) | O(n) | O(n) |
| Average | O(n) | O(n) | O(n) |

And they are verified by implementation on gcc on EE Workstation.

1. Actual runtime comparison (Average):

**BDSearch > Linear Search > RDSearch (> means runs faster)**

1. Actual runtime comparison (Worst):

**RDSearch > BDSearch > Linear Search (> means runs faster)**

1. Half iteration but contains double statement (BDSearch) could run faster than whole iteration with single statement (Linear Search).
2. The worst-case of Random-directional Search is not accurate.