**EE 3980 Algorithms**

Homework 3. Heap Sort

105061110 周柏宇

2020/3/25

1. **Introduction**

In this homework,

1. **Analysis & Implementation**
   1. **Selection Sort**
2. // Sort A[1 : n] into nondecreasing order.
3. // Input: array A with n elements
4. // Output: array A sorted
5. Algorithm SelectionSort(A, n)
6. {
7. **for** i := 1 to n **do** {
8. j := i; // initialize j to be i
9. // find the smallest in A[i + 1 : n]
10. **for** k := i + 1 to n **do**
11. //if found, record the index
12. **if** (A[k] < A[j]) then j := k;
13. // swap A[i] and A[j]
14. t := A[i]; A[i] := A[j]; A[j] := t;
15. }
16. }

In selection sort, we can see that the program has to go through the same number of comparisons and swaps no matter how the input data distributes. Therefore, the worst-case and best-case performance are the same, which requires swaps.

Best-case time complexity:

Average time complexity:

Worst-case time complexity:

Space complexity:

* 1. **Insertion Sort**

1. // Sort A[1 : n] into nondecreasing order.
2. // Input: array A with n elements
3. // Output: array A sorted
4. Algorithm InsertionSort(A, n)
5. {
6. // assume A[1 : j - 1] already sorted
7. **for** j := 2 to n **do** {
8. item := A[j]; // move A[j] to its proper place
9. i := j - 1; // initialize i to be j - 1
10. // find i such that A[i] <= A[j]
11. **while** ((i >= 1) and (item < A[i])) **do** {
12. // move A[i] up by one position
13. A[i + 1] := A[i];
14. i := i - 1;
15. }
16. A[i + 1] := item; // move A[j] to A[i + 1]
17. }
18. }

In insertion sort, we notice that the while loop at line 11 goes on when the index is not out of bound and . Hence, the while loop will execute at most times when given , i.e. is lesser than all the elements to its left. If this condition holds for , the worst-case input will be in nonincreasing order. Therefore, if we count the number of times line 13 being executed, it will be

times; while the best-case

Best-case time complexity:

Average time complexity:

Worst-case time complexity:

Space complexity:

* 1. **Bubble Sort**

1. // Sort A[1 : n] into nondecreasing order.
2. // Input: array A with n elements
3. // Output: array A sorted
4. Algorithm BubbleSort(A, n)
5. {
6. // find the smallest item for A[i]
7. **for** i := 1 to n - 1 **do** {
8. **for** j := n to i + 1 step -1 **do** {
9. **if** (A[j] < A[j - 1]) {
10. // swap A[j] and A[j - 1]
11. t := A[j];
12. A[j] := A[j - 1];
13. A[j - 1] := t;
14. }
15. }
16. }
17. }

Best-case time complexity:

Average time complexity:

Worst-case time complexity:

Space complexity:

* 1. **Shaker Sort**

1. // Sort A[1 : n] into nondecreasing order.
2. // Input: array A with n elements
3. // Output: array A sorted
4. Algorithm ShakerSort(A, n)
5. {
6. l := 1; r := n;
7. **while** l <= r **do** {
8. // element exchange from r to l
9. **for** j := r to l + 1 step -1 **do** {
10. **if** (A[j] < A[j - 1]) {
11. // swap A[j] and A[j - 1]
12. t := A[j];
13. A[j] := A[j - 1];
14. A[j - 1] := t;
15. }
16. }
17. l := l + 1;
18. // element exchange from l to r
19. **for** j := l to r - 1 **do** {
20. **if** (A[j] > A[j + 1]) {
21. // swap A[j] and A[j + 1]
22. t := A[j];
23. A[j] := A[j + 1];
24. A[j + 1] := t;
25. }
26. }
27. r := r - 1;
28. }
29. }

Best-case time complexity:

Average time complexity:

Worst-case time complexity:

Space complexity:

* 1. **Heap Sort**

1. // To enforce max heap property for n-element heap A
2. // with root i.
3. // Input: size n max heap array A with root at i
4. // Output: updated A
5. Algorithm Heapify(A, i, n)
6. {
7. j := i \* 2; // init A[j] to be the lchild of A[i]
8. item := A[i]; // copy the original A[i]
9. done := **false**;
10. **while** ((j <= n) and (not done)) **do** {
11. **if** ((j < n) and (A[j] < A[j + 1])) then
12. j := j + 1; // make A[j] the larger child
13. // if A[j] is larger than the original A[i]
14. **if** (item > A[j]) then
15. done = **true**; // end the loop
16. **else** {
17. // overwrite A[j]'s parent with A[j]
18. A[⌊j / 2⌋] := A[j];
19. j := j \* 2; // move down to lchild
20. }
21. }
22. // move the original A[i] to proper place
23. A[⌊j / 2⌋] = item;
24. }

27. // Sort A[1 : n] into nondecreasing order.
28. // Input: array A with n elements
29. // Output: array A sorted
30. Algorithm HeapSort(A, n)
31. {
32. // initialize A[1 : n] to be a max heap
33. **for** i := ⌊n / 2⌋ to 1 step -1 **do**
34. Heapify(A, i, n);
36. **for** i := n to 2 step -1 **do** {
37. // move maximum to the end
38. t := A[1]; A[i] := A[1]; A[i] := t;
39. // make A[1 : i - 1] a max heap
40. Heapify(A, 1, i - 1);
41. }
42. }

Best-case time complexity:

Average time complexity:

Worst-case time complexity:

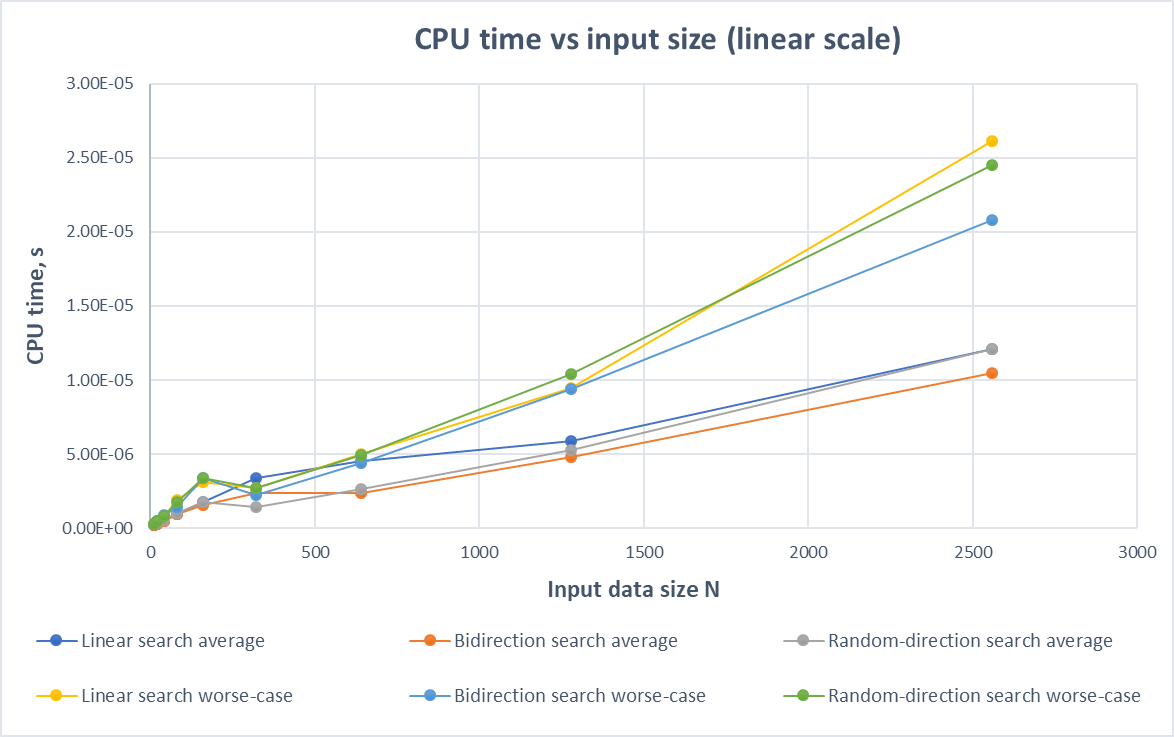
Space complexity:

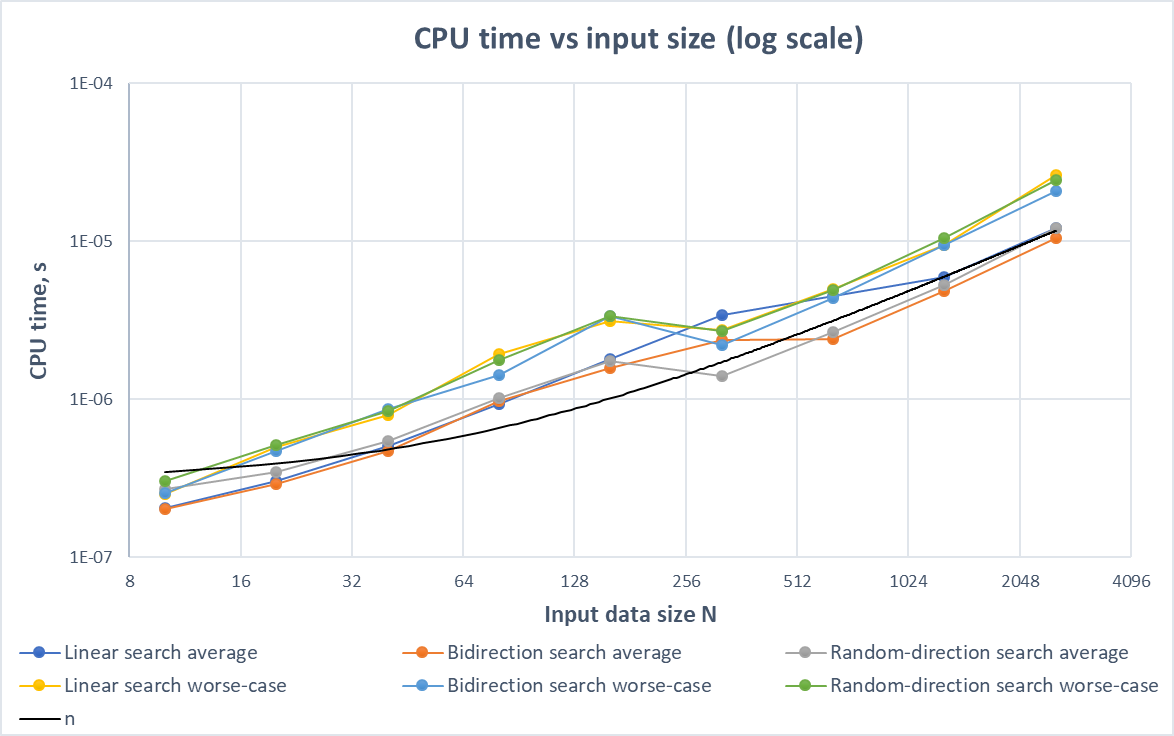
1. **Result and Observation**

To measure the average performance, we search all possible words in the list and average the CPU time over the number of words as average performance. As for the worse-case performance, we choose to search the word that requires most comparisons for each search algorithm. To exhibit and compare the growth of CPU time, we run three search algorithms with different numbers of input for several times. Specifically, the average performance is averaged over 500 trials and the worse-case performance is averaged over 5000 trials.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | Average | | | Worse-case | | |
| *Linear search* | *Bidirection search* | *Random-direction search* | *Linear search* | *Bidirection search* | *Random-direction search* |
| 10 | 0.206375 | 0.200796 | 0.270987 | 0.250006 | 0.25382 | 0.304413 |
| 20 | 0.305009 | 0.292397 | 0.346279 | 0.501442 | 0.469017 | 0.512171 |
| 40 | 0.506604 | 0.473344 | 0.548053 | 0.80018 | 0.861788 | 0.837994 |
| 80 | 0.930452 | 0.97487 | 1.01422 | 1.93582 | 1.41358 | 1.78337 |
| 160 | 1.78876 | 1.56901 | 1.75746 | 3.11975 | 3.37939 | 3.3812 |
| 320 | 3.40464 | 2.36826 | 1.39842 | 2.74181 | 2.21658 | 2.68579 |
| 640 | 4.51523 | 2.40808 | 2.66012 | 4.99859 | 4.39959 | 4.93641 |
| 1280 | 5.89816 | 4.81463 | 5.28454 | 9.4604 | 9.413 | 10.4376 |
| 2560 | 12.0937 | 10.4677 | 12.0814 | 26.1076 | 20.804 | 24.4856 |

Table 1. CPU time [μs] vs input data size *N*





According to our analysis, we expect the performance to be the same in terms of the average and worst case since the only difference between them is the order of searching, and it roughly holds. The bidirection search seems to be a little bit faster due to our implementation which requires less operation at loop comparison. We also expect the worse-case performance to be around two times slower than that of the average case. In Table 1, for data set with a larger number of words, we certainly can see the relationship.

Since the average and worse-case time complexity are for all three search algorithms, in the linear scale scatter plot, we can observe the linear trend for all the lines if we ignore the fluctuation at small input sizes. In the log scale scatter plot, we can compare the slope with the linear line to show that they are indeed linear, and the difference between average and worse-case lines is only the bias.