**EE 3980 Algorithms**

Homework 3. Heap Sort

105061110 周柏宇

2020/3/25

1. **Introduction**

In this homework, we implemented heap sort and compared its performance with selection sort, insertion sort bubble sort and shaker sort by sorting lists containing different number of randomly-ordered English words. We not only measured their average performance, but also rearranged the input data according to our analysis to test their best-case and worst-case execution time. In the end, we plot out the result and observe the growth of CPU time to verify our derivation of their time complexity.

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 is

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 occurs when

which requires the input to be in nondecreasing order and the while loop will only execute one time. Therefore, the best-case time complexity is . On average, we expect the time complexity of the while loop to be bounded by rather than , which makes average time complexity of insertion sort .

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. }

In bubble sort, what the inner loop does is moving the smallest element in to index *i* by swapping. Therefore, if the input data is in nonincreasing order, it will require most swaps, thus making the worst-case performance to be . On the contrary, if the input data is already in nondecreasing order, then no swap is needed. However, in our implementation, we still need of comparisons.

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. }

In shaker sort, the first for loop moves the smallest element in to *l* and the second for loop moves the greatest element in to *r*. Therefore, to make the execution time longest, the input data should order as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| greatest | 2nd greatest | … | 2nd smallest | smallest |

which is in nonincreasing order; the execution time will be the shortest when in input data is in nondecreasing order, since it requires least swap. Nevertheless, just as bubble sort, the number of comparisons is deterministic, which makes both best-case and worst-case time complexity .

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

To make the implementation of heap sort easier, we separate the operation *Heapify* from the main algorithm. What *Heapify* does is moving the root element to proper place by swapping with its larger child. In *HeapSort*’s first for loop, we execute *Heapify* from the lowest non-leave node all the way to the root. In this way, we can make sure all the subtrees follow the max heap property, thus making array A a max heap. As for the second loop of *HeapSort*, we swap the root element, which is the maximum, with the last element of the array and *Heapify* the array with size decrease by one to make the next maximum be at the root again. Eventually, the element in A will sorted in nondecreasing order.

In *Heapify*, the while loop ends when *j* is greater than *n* and *j* doubles every time when root element is smaller than . Therefore, *Heapify* has worst-case time complexity in some scenarios, e.g. when the root element is minimum of the subtree. *Heapify* has best-case time complexity if the root element is greater than its children.

In *HeapSort*, if the input data is in min heap order, worst case will occur in every execution of *Heapify* in first for loop. Thus, it will result in time complexity

However, if the input data is already in max heap, then the first loop will only contribute since every *Heapify* only takes .

For the second for loop in *HeapSort*, the worst-case time complexity is bounded by . Therefore, the worst-case complexity of heap sort is . However, it is hard to analyze the best-case and average time complexity explicitly. As a result, I can only upper-bounded them by .

If we want to test the best-case and worst-case performance, using input with nonincreasing and nondecreasing order respectively may be justified. The graph below is the number of while loop executed versus the number of elements to be sorted. For an array with *n* distinct elements, there are *n*! permutations. I enumerate all permutations and execute the heap sort to find the maximal/minimal number of while loop executed. We can treat the input with increasing order to be the lower bound of worst case and input with increasing order to be upper bound of best case.

Best-case time complexity:

Average time complexity:

Worst-case time complexity:

Space complexity:

1. **Result and Observation**

To measure the best-case, average and worst-case performance, we execute each sorting algorithm with their corresponding rearranged input according to our previous analysis. Also, each data point in the table is the average result of 500 executions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Selection sort | Insertion sort | Bubble sort | Shaker sort | Heap sort |
| Best-case | Nondecreasing | | | | Nonincreasing |
| Average | Original data (randomly-ordered) | | | | |
| Worst-case | Nonincreasing | | | | Nondecreasing |

Table 1. Types of input to measure the performance for sorting algorithms

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | Average | | | | |
| *Selection sort* | *Insertion sort* | *Bubble sort* | *Shaker sort* | *Heap sort* |
| 10 | 2.28357 | 1.21212 | 2.92778 | 2.85769 | 2.83003 |
| 20 | 6.68621 | 4.98009 | 10.54 | 10.8159 | 8.19206 |
| 40 | 22.172 | 14.8222 | 41.3857 | 42.2621 | 17.6578 |
| 80 | 82.8619 | 25.6281 | 169.11 | 166.694 | 39.4602 |
| 160 | 305.348 | 78.0401 | 684.662 | 673.282 | 91.7859 |
| 320 | 1154.24 | 357.7 | 1798.58 | 2205.04 | 213.186 |
| 640 | 2660.69 | 1275.13 | 5642.05 | 5257.94 | 481.65 |
| 1280 | 8227.41 | 5581.26 | 20420.3 | 19429.4 | 1072.71 |
| 2560 | 34145.1 | 22112.4 | 87123.1 | 78771.4 | 1709.87 |

Table 2. Average CPU time [μs] vs input data size *N*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | Worst Case | | | | Lower Bound of Worst Case |
| *Selection sort* | *Insertion sort* | *Bubble sort* | *Shaker sort* | *Heap sort* |
| 10 | 2.32601 | 0.87595 | 1.28031 | 1.39809 | 2.85387 |
| 20 | 7.26175 | 2.97165 | 4.68588 | 4.76599 | 6.84404 |
| 40 | 11.2519 | 12.1999 | 18.024 | 18.5838 | 16.4618 |
| 80 | 42.1681 | 39.0759 | 70.5919 | 74.8219 | 39.1397 |
| 160 | 159.4 | 150.846 | 279.596 | 281.164 | 88.2602 |
| 320 | 636.564 | 591.236 | 1131.56 | 1123.68 | 200.886 |
| 640 | 2579.73 | 2338.45 | 4399.62 | 4487.31 | 452.89 |
| 1280 | 10645.9 | 9373.83 | 17959.5 | 17540 | 990.148 |
| 2560 | 49081.6 | 38519.5 | 72413.9 | 70900.9 | 1615.98 |

Table 2. Worst-case CPU time [μs] vs input data size *N*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | Best Case | | | | Upper Bound of Best Case |
| *Selection sort* | *Insertion sort* | *Bubble sort* | *Shaker sort* | *Heap sort* |
| 10 | 2.03371 | 0.874043 | 0.748158 | 0.92411 | 0.976086 |
| 20 | 2.52008 | 0.674248 | 3.93009 | 2.93636 | 2.68412 |
| 40 | 8.38614 | 1.44815 | 11.8718 | 12.6138 | 8.3518 |
| 80 | 32.9399 | 3.17574 | 50.5142 | 53.412 | 36.1724 |
| 160 | 113.562 | 5.99384 | 221.974 | 248.068 | 85.0878 |
| 320 | 435.574 | 11.7278 | 988.59 | 1768.83 | 187.258 |
| 640 | 1720.92 | 24.6601 | 3762.97 | 3764.88 | 419.666 |
| 1280 | 7004.51 | 49.8462 | 16679.9 | 16191 | 950.098 |
| 2560 | 28182.4 | 108.496 | 72331.7 | 69787 | 947.914 |

Table 3. Best-case CPU time [μs] vs input data size *N*