CSE 12 Program 4

File: PR4.txt

Name: Chih-Hsuan Kao

PID: A92092372 / Login: cs12sgh

Date: May 17, 2018

/\*\* PART 2: Running Time of Various Sort Algorithms \*\*/

/\*\* I. UNSORTED CASE \*\*/

1. Bubble
   1. Testing parameter:

private static final int NUMWORDS = 700;

numwords \*= 2; (such that \*2 after each interval)

I set the initial value to be 700 (which is relatively small compared to others) such that the runtime isn’t too long for the expected output. I set numwords \*= 2 such that I can see the output when N = 2.

* 1. The Output of the actual test

Document: random-strings.txt

sortAlg: 0

=======================================

1: 700 words in 19 milliseconds

2: 1400 words in 28 milliseconds

3: 2800 words in 93 milliseconds

4: 5600 words in 402 milliseconds

5: 11200 words in 1853 milliseconds

* 1. From the output above, we can approximately see that BubbleSort on this unsorted input is O(n^2). From interval 2 to 3, as words double (N = 2), 28 \* 2^2 = 112 which is close to 93 (actual observation). From interval 3 to 4, as words double (N = 2), 93 \* 2^2 = 372 which is close to 402 (actual observation).

1. Insertion
   1. Testing parameter:

private static final int NUMWORDS = 10000;

numwords \*= 2; (such that \*2 after each interval)

I set the initial value to be 10000 such that the runtime isn’t too long (or too short) for me to wait for the output. I set numwords \*= 2 such that I can easily see the output when N = 2.

* 1. The Output of the actual test

Document: random-strings.txt

sortAlg: 1

=======================================

1: 10000 words in 14 milliseconds

2: 20000 words in 26 milliseconds

3: 40000 words in 84 milliseconds

4: 80000 words in 324 milliseconds

5: 160000 words in 1400 milliseconds

From the output above, we can approximately see that InsertionSort on this unsorted input is O(n^2). From interval 3 to 4, as words double (N = 2), 84 \* 2^2 = 336 which is close to 324 (actual observation). From interval 4 to 5, as words double (N = 2), 324 \* 2^2 = 1296 which is close to 1400 (actual observation).

1. Merge
   1. Testing parameter:

private static final int NUMWORDS = 10000;

numwords \*= 2; (such that \*2 after each interval)

I set the initial value to be 10000 such that the runtime isn’t too long (or too short) for me to wait for the output. I set numwords \*= 2 such that I can easily see the output when N = 2.

* 1. The Output of the actual test

Document: random-strings.txt

sortAlg: 2

=======================================

1: 10000 words in 18 milliseconds

2: 20000 words in 12 milliseconds

3: 40000 words in 27 milliseconds

4: 80000 words in 64 milliseconds

5: 160000 words in 153 milliseconds

Note that the log I use here has base of 2.

From the output above, we can approximately see that the MergeSort algorithm that I implemented has O(n logn) complexity on this unsorted input as I expect. From interval 3 to 4, as words double (N = 2), 27 \* 2\*log(2) = 54 which is close to 64 (actual observation). From interval 4 to 5, as words double (N = 2), 64 \* 2\*log(2) = 128 which is close to 153 (actual observation).

1. Quick
   1. Testing parameter:

private static final int NUMWORDS = 5000;

numwords \*= 2; (such that \*2 after each interval)

I set the initial value to be 5000 such that the runtime isn’t too long (or too short) for me to wait for the output, and that initial value of 5000 gives me a clear enough pattern to observe. I set numwords \*= 2 such that I can easily see the output when N = 2.

* 1. The Output of the actual test

Document: random-strings.txt

sortAlg: 3

=======================================

1: 5000 words in 10 milliseconds

2: 10000 words in 5 milliseconds

3: 20000 words in 10 milliseconds

4: 40000 words in 25 milliseconds

5: 80000 words in 50 milliseconds

Note that the log I use here has base of 2.

From the output above, we can approximately see that the QuickSort algorithm that has O(n logn) complexity on this unsorted input as I expect. From interval 3 to 4, as words double (N = 2), 10 \* 2\*log(2) = 20 which is close to 25 (actual observation). From interval 4 to 5, as words double (N = 2), 25 \* 2\*log(2) = 50 which is exactly 50 (actual observation).

/\*\* II. SORTED CASE \*\*/

1. Bubble
   1. Testing parameter:

private static final int NUMWORDS = 25000;

numwords \*= 2; (such that \*2 after each interval)

I set the initial value to be 25000 (which seems very large) such that the pattern is observable. If too small, the whole things will finish in 0 millisecond. I set numwords \*= 2 such that I can see the output when N = 2.

* 1. The Output of the actual test

Document: random-strings-sorted.txt

sortAlg: 0

=======================================

1: 25000 words in 2 milliseconds

2: 50000 words in 3 milliseconds

3: 100000 words in 5 milliseconds

4: 199999 words in 10 milliseconds

* 1. From the output above, we can approximately see that BubbleSort on this completely sorted input is O(n), less than and differs from the unsorted case (O(n^2)). We have Best Case here. This is because the only thing the algorithm does is traverse through each element of the list. From interval 2 to 3, as words double (N = 2), 3 \* 2 = 6 which is close to 5 (actual observation). From interval 3 to 4, as words double (N = 2), 5 \* 2 = 10 which is exactly the actual observation.

1. Insertion
2. Testing parameter:

private static final int NUMWORDS = 20000;

numwords \*= 2; (such that \*2 after each interval)

I set the initial value to be 20000 such that the runtime isn’t too long (or too short) for me to wait for the output. If bigger, would exceed my word limit; if smaller, no pattern could possibly be observed. I set numwords \*= 2 such that I can easily see the output when N = 2.

1. The Output of the actual test

Document: random-strings-sorted.txt

sortAlg: 1

=======================================

1: 20000 words in 6 milliseconds

2: 40000 words in 2 milliseconds

3: 80000 words in 5 milliseconds

4: 160000 words in 11 milliseconds

5: 199999 words in 14 milliseconds

From the output above, we can approximately see that InsertionSort on this sorted input is O(n), which differs from the unsorted case, which has O(n^2) time complexity. We have Best Case here. From interval 2 to 3, as words double (N = 2), 2 \* 2 = 4 which is close to 5 (actual observation). From interval 3 to 4, as words double (N = 2), 5 \* 2 = 10 which is close to 11 (actual observation).

1. Merge
2. Testing parameter:

private static final int NUMWORDS = 10000;

numwords \*= 2; (such that \*2 after each interval)

I set the initial value to be 10000 such that the runtime isn’t too long (or too short) for me to wait for the output, while the pattern is clear enough for me to observe. I set numwords \*= 2 such that I can easily see the output when N = 2.

1. The Output of the actual test

Document: random-strings-sorted.txt

sortAlg: 2

=======================================

1: 10000 words in 16 milliseconds

2: 20000 words in 9 milliseconds

3: 40000 words in 16 milliseconds

4: 80000 words in 37 milliseconds

5: 160000 words in 82 milliseconds

Note that the log I use here has base of 2.

From the output above, we can approximately see that the MergeSort algorithm that I implemented has O(n logn) complexity on this sorted input as I expect. This is the same as the unsorted case. From interval 3 to 4, as words double (N = 2), 16 \* 2\*log(2) = 32 which is close to 37 (actual observation). From interval 4 to 5, as words double (N = 2), 37 \* 2\*log(2) = 74 which is close to 82 (actual observation).

1. Quick
2. Testing parameter:

private static final int NUMWORDS = 2000;

numwords \*= 2; (such that \*2 after each interval)

I set the initial value to be 2000 (which seems small) such that the runtime isn’t too long for me to wait for the output. I set numwords \*= 2 such that I can easily see the output when N = 2.

1. The Output of the actual test

Document: random-strings-sorted.txt

sortAlg: 3

=======================================

1: 2000 words in 57 milliseconds

2: 4000 words in 99 milliseconds

3: 8000 words in 282 milliseconds

4: 16000 words in 855 milliseconds

5: 32000 words in 3342 milliseconds

1. Note that the log I use here has base of 2.

From the output above, we can approximately see that the QuickSort algorithm has O(n^2) complexity on this sorted input. This is the worst case. If the pivot is the first element (as implemented in the code) then already sorted or inverse sorted data is the worst case.

From interval 3 to 4, as words double (N = 2), 282 \* 2^2 = 1128 which is close to 855 (actual observation). From interval 4 to 5, as words double (N = 2), 855 \* 2^2 = 3420 which is exactly 3342 (actual observation).

/\*\* III. What do you notice about the behaviour of the various algorithms in the pre-sorted case? \*\*/

//Analysis on Bubble Sort

As I just analyzed, the bubble sort algorithm has O(n^2) time complexity on unsorted case, but has O(n) time complexity on presorted case. As another support of my assertion, the following two sets have exactly same conditions, on two different files--

Document: random-strings.txt

sortAlg: 0

=======================================

1: 700 words in 19 milliseconds

2: 1400 words in 28 milliseconds

3: 2800 words in 93 milliseconds

4: 5600 words in 402 milliseconds

5: 11200 words in 1853 milliseconds

Document: random-strings-sorted.txt

sortAlg: 0

=======================================

1: 700 words in 0 milliseconds

2: 1400 words in 0 milliseconds

3: 2800 words in 0 milliseconds

4: 5600 words in 1 milliseconds

5: 11200 words in 1 milliseconds

BubbleSort works much more efficiently on presorted case. This is because in Best case, what the algorithm does is to traverse through each element once without actually doing anything.

//Analysis on Insertion Sort

As I just analyzed, the insertion sort algorithm has O(n^2) time complexity on unsorted case, but has O(n) time complexity on presorted case. As another support of my assertion, the following two sets have exactly same conditions, on two different files--

Document: random-strings.txt

sortAlg: 1

=======================================

1: 10000 words in 15 milliseconds

2: 20000 words in 28 milliseconds

3: 40000 words in 92 milliseconds

4: 80000 words in 354 milliseconds

5: 160000 words in 1534 milliseconds

Document: random-strings-sorted.txt

sortAlg: 1

=======================================

1: 10000 words in 5 milliseconds

2: 20000 words in 1 milliseconds

3: 40000 words in 3 milliseconds

4: 80000 words in 5 milliseconds

5: 160000 words in 14 milliseconds

InsertionSort works much more efficiently on presorted case. This is because in Best case, For each outer loop execution, if the element is already in sorted position, only a single comparison is made. Each element not in sorted position requires at most N comparisons.

//Analysis on Merge Sort

As I just analyzed, the merge sort algorithm has O(n\*log(n)) time complexity on BOTH sorted and presorted case. As another support of my assertion, the following two sets have exactly same conditions, on two different files--

Document: random-strings.txt

sortAlg: 2

=======================================

1: 10000 words in 18 milliseconds

2: 20000 words in 12 milliseconds

3: 40000 words in 27 milliseconds

4: 80000 words in 64 milliseconds

5: 160000 words in 153 milliseconds

Document: random-strings-sorted.txt

sortAlg: 2

=======================================

1: 10000 words in 16 milliseconds

2: 20000 words in 9 milliseconds

3: 40000 words in 23 milliseconds

4: 80000 words in 48 milliseconds

5: 160000 words in 106 milliseconds

InsertionSort works similarly in both cases. This is because in both cases, Merge sort divides the input in half until a list of 1 element is reached, which requires log N partitioning levels. At each level, the algorithm does about N comparisons selecting and copying elements from the left and right partitions, yielding N \* log N comparisons.

//Analysis on Quick Sort

As I just analyzed, the quick sort algorithm has O(n\*log(n)) time complexity on BEST case and O(n^2) time complexity in WORST case. The best case here is the unsorted one; the worst case here is the presorted one. As another support of my assertion, the following two sets have exactly same conditions, on two different files--

Document: random-strings.txt

sortAlg: 3

=======================================

1: 5000 words in 10 milliseconds

2: 10000 words in 5 milliseconds

3: 20000 words in 10 milliseconds

4: 40000 words in 25 milliseconds

5: 80000 words in 50 milliseconds

Document: random-strings-sorted.txt

sortAlg: 3

=======================================

1: 5000 words in 162 milliseconds

2: 10000 words in 594 milliseconds

3: 20000 words in 2285 milliseconds

4: 40000 words in 8125 milliseconds

5: 80000 words in 32362 milliseconds

InsertionSort works much more efficiently on unsorted case. This is because basically, at each level, the algorithm does at most N comparisons moving the l and h indices. If the pivot yields two equal-sized parts, then there will be log N levels, requiring the N \* log N comparisons. However, partitioning may yield unequal sized part in some cases. If the pivot selected for partitioning is the smallest or largest element, one partition will have just 1 element, and the other partition will have all other elements. This causes the O(n^2) time complexity in the worst case.

/\*\* PART 3: Examining Modified Insertion Sort \*\*/

Describe how modified insertion sort differs from classic insertion sort. Specifically,

* 1. What does the method binsearch actually do?

The binSearch method is called by the sort method. The method takes in a sorted ArrayList and a target, that is used to look for where to insert. Unlike typical Insertion Sort, it doesn’t go from one element to another one-by-one; instead, it set a midpoint as a local variable, and use binary search algorithm to look for the element by going on both side (right and left). The method uses binary search. If the search key is smaller than anything in the list, the algorithm returns -1. If the search key is bigger than anything else, the algorithm the algorithm returns size-1 such that the inserted element will be on the rightmost position. Otherwise, search in the middle and repeat the search on the remaining left and right sublist until found the proper place to insert.

* 2. How is it used in the modified insertion sort?

In the sort method, the binSearch method is called by passing in the input arrayList and the target that detects where the inserted element should be inserted. The method uses binary search. If the search key is smaller than anything in the list, the algorithm returns -1. If the search key is bigger than anything else, the algorithm the algorithm returns size-1 such that the inserted element will be on the rightmost position. Otherwise, search in the middle and repeat the search on the remaining left and right sublist until found until found the proper place to insert.

Then, with the help of this method, we know where to insert. Back in the sort method, on line 21, the insertion key is successfully added with inserted.add(loc+1,target) because loc is what we just got from the binSearch method. The binSearch method helps find the proper place to insert more efficiently by performing binary search.

* 3. What is the space complexity of classic insertion sort?

(in other words, how much  additional (temporary) space is required to have insertion sort work)

The space complexity of classic insertion sort is O(1). This is because we need an additional space to store the variable that indicates where is the right position for the key to be inserted after searching process. With the space holding the variable, we’ll make sure nothing overwrites the value before performing actual insertion.

* 4. What is the space complexity of modified insertion sort?

(how much additional  (temporary) space is used by modified insertion sort)

In this code, the space complexity is O(n). This is because in addition to the extra space that stores the variable that indicates where is the right position for the key to be inserted (as I mentioned in previous question), the code also creates an ArrayList to insert from the List, and the list has size of n. Thus, the space complexity in this case is O(1) + O(n) = O(n).