CS 221 Analysis of Algorithms Homework

Sam Stoodant

*All growth functions must be in simplified t(n) = \_\_\_\_ format with only* ***one*** *constant factor,* ***one*** *n factor, etc. Runtime order must be presented in proper big-O notation. All writing is required to be proofread for professional-quality grammar, spelling, capitalization, punctuation, complete sentences, etc.*

# Algorithm: doSomething()

*Testing questions in this section refer to the DoSomethingTester class given with the assignment. DoSomethingTester initializes an array of integers of size n configured according to input parameters. The required second argument indicates how the input array will be generated: (a)scending, (d)escending, (r)andom, or (m)ultiples/duplicates. If (m)ultiples/duplicates is the choice, a third argument indicating the number of duplicates must be provided as well. Note that the reported result for any numDuplicates less than n is subject to vagaries of random distribution. DoSomethingTester usage:*

*$ java DoSomethingTester [array size n] [a | d | r | m numDuplicates]*

*where a → ascending order, d → descending order, r → random order, and m → multiples/duplicates, requiring a third argument for the number of duplicates up to a maximum of n*

## **Minimum Statements, Constant Factor**

What statements are executed in a call to doSomething() before reaching a return statement when the array size is zero (n == 0)? (Do not count the initialization of method arguments or return statements.) What is t(0) for find(), the minimum cost and the constant factor?

If the array size was 0, the for loop would initialize int i and check if it is less than the array length, which would be false and that would be the end of the method. So, no matter what, at least 2 statements get executed.

Predicted t(0) = 2

### Run DoSomethingTester 0 a

What is your prediction for t(0)? How many statements does the test report? How do the results compare to your expectation? If there is a discrepancy, go back to the code to figure out why that might be. What (if anything) do you need to modify about your analysis to better align with the empirical results?

Predicted t(0) Statements: 2

DoSomethingTester 0 -1 Statements: 2

DoSomethingTester returned 2 as the statement count as predicted, so I think my analysis for minimum statements is correct.

Final t(0) = 2

## **Best Case Scenario**

Assume a large array size n. The doSomething() outer loop clearly depends on n. What about the inner loop? How many inner loop iterations occur for every outer loop iteration? Does anything other than array size affect the number of inner loop iterations? What statements are executed in every iteration of the outer loop? What statements are executed in every iteration of the inner loop? Under what conditions would the condition statement inside the inner loop be false every time, such that the statements within the if block are never executed? What is the best case growth function t(n) under these conditions?

The for loop is going to iterate once for every index in the array, although it doesn’t look like that index is ever used. So whatever happens in the loop is going to be multiplied by the size of the array, which is n for this algorithm.

Variables left and right always get initialized and compared, so that’s 3 statements that always happen in the loop. The while loop condition and the left++ and right-- statements inside make it look like the loop is always walking from the ends toward the middle of the array. I think that means the inner loop is going to happen n/2 times, every time through the outer loop. Nothing about the array contents affects that number of loop iterations.

Inside the while loop, there is a condition statement that will always get checked, left and right both get updated, and the while loop’s condition is going to get checked, so that’s 4 statements that happen for every while loop iteration. The part that changes is whether or not the value at array[left] and array[right] are out of order. If so, three statements get executed to swap those values. If everything in the array is already in order, those three statements would never get executed, so that seems like the best case scenario for this algorithm is when everything is already in order.

Predicted tbest(n) = 2 + n(5 + (n/2)4) = 2n^2 + 5n + 2

### Run DoSomethingTester 100 a and DoSomethingTester 100 m 100

What is your predicted number of statements when n == 100? How does the number of reported statements align with your expectation? If there is a discrepancy, go back to the code to figure out why that might be. What do you need to modify about your analysis to better align with the empirical results?

Predicted tbest(100) Statements: 20502

DoSomethingTester 100 a Statements: 20502

DoSomethingTester 100 m 100 Statements: 20502

So, first of all, I’m really thrilled that my predicted growth function exactly matched what the tester reported for the ascending array. I was surprised, though, to realize the array with all duplicate values has the same result. I didn’t think of that, but it makes sense now.

Final tbest(n) = 2n^2 + 5n + 2

## **Worst Case Scenario**

Assume a large array size n. The doSomething() outer loop clearly depends on n. What about the inner loop? How many inner loop iterations occur for every outer loop iteration? Does anything other than array size affect the number of inner loop iterations? What statements are executed in every iteration of the outer loop? What statements are executed in every iteration of the inner loop? Under what conditions would the condition statement inside the inner loop be true every time, such that the statements within the if block are always executed? What is the worst case growth function t(n) under these conditions?

If the best case scenario was everything already being in order so (array[left] > array[right]) is always false, the worst case is going to make it always true. If the array is in reverse order, I expect that would be the case. The same outer loop statements and inner loop statements get executed, but there are now the three additional statements (looks like a 3-step swap sequence) in every inner loop iteration.

Predicted tworst(n) = 2 + n(5 + (n/2)\*(4+3)) = (7/2)n^2 + 5n + 2

### Run DoSomethingTester 100 d

What is your predicted number of statements when n == 100? How does the number of reported statements for the actual worst case compare to your expectation? If there is a discrepancy, go back to the code to figure out why that might be. What (if anything) do you need to modify about your analysis to better align with the empirical results?

Predicted tworst(100) Statements: (7/2)(100^2) + 5(100) + 2 = 35502

DoSomethingTester 100 d Statements: 20652

Whoops. Okay, my prediction is way higher than what the tester reported. In fact, the reported statements weren’t much worse than the best case numbers. It looks like the if (array[left] > array[right]) condition for each right and left index can only ever be true once. So the first time through the for loop, the condition is true n/2 times, but then every subsequent time through the for loop is totally wasted effort! Nothing ever changes after the first time!

So, reverse order is still the worst case, but the growth function should really be:

maximum T(n) = 2 + n(5 + (n/2)4) + **(n/2)3** = 2n^2 + (13/2)n + 2

This function matches the tester results with the new predicted worst case statements: 20652

Final tworst(n) = 2n^2 + (13/2)n + 2

## **Expected Average Case Scenario**

Assume a large array size n. Assuming a random array of unique elements, what is the expected average frequency of the innermost condition statement being true? Why? How does this compare to the worst case? What is the expected average growth function t(n) under these conditions?

The only difference between the worst case scenario above is that the if condition would be true only about half as often on that first time through the outer loop. After the first pass through the outer loop, it will never be true again, just like with the worst case.

Predicted tavg(n) = 2 + n(5 + (n/2)4) + **(n/2)\*3/2** = 2n^2 + (23/4)n + 2

### Run DoSomethingTester 100 r

What is your predicted number of statements when n == 100? How does the average number of statements to find all elements align with your expectation? If there is a discrepancy, go back to the code to figure out why that might be. What (if anything) do you need to modify about your analysis to better align with the empirical results?

Predicted tavg(100) Statements: 2(100^2) + (24/4)(100) + 2 = 20602

DoSomethingTester 100 r Statements: 20595

I actually ran the tester several times when it didn’t match the first time. I realized that testing with random arrays is different every time because of randomness, but the results were consistently in between the best case and worst case counts and usually close to my prediction, so I am confident in my analysis.

Final tavg(n) = 2n^2 + (23/4)n + 2

## **Order**

What is the runtime order (big-O) of find()?

O(n^2).