CS 221 Analysis of Algorithms Homework

[your name]

*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: find()

*Testing questions in this section refer to the FindTester class given with the assignment. FindTester initializes an array of integers in ascending order from 1 to n inclusive. If a target value is provided, FindTester reports the number of statements executed before a result is returned. If no target value is provided, FindTester reports the average number of statements to find all values in the array. FindTester usage:*

$ java FindTester [array size n] <optional target value>

## **Minimum Statements, Constant Factor**

What statements are executed in a call to find() 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?

Predicted t(0) =

### Run FindTester 0 -1

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:

FindTester 0 -1 Statements:

Final t(0) =

## **Best Case Scenario**

Assuming a large array size n and the target element is located at index 0, what statements are executed before the index is returned? What is the best case growth function t(n) under these conditions?

Predicted tbest(n) =

### Run FindTester 100 1

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:

FindTester 100 1 Statements:

Final tbest(n) =

## **Worst Case Scenario**

Assuming a large array size n, what would be necessary such that the method returns -1? How many times does the loop iterate? What statements are executed in each loop iteration? What is the worst case growth function t(n) under these conditions?

Predicted tworst(n) =

### Run FindTester 100 -1

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:

FindTester 100 1 Statements:

Final tworst(n) =

## **Expected Average Case Scenario**

Assuming a randomly ordered array of unique elements and the target element is in the array, where would a target element be located **on average**? What is the expected average number of loop iterations if this is the case? What statements are executed in each complete loop iteration? Are there any loop statements that will **not** be executed when the target is found? What is the expected average growth function t(n) under these conditions?

Predicted tavg(n) =

### Run FindTester 100

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:

FindTester 100 Statements:

Final tavg(n) =

## **Order**

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

O()

# Algorithm: replaceAll()

*Testing questions in this section refer to the ReplaceAllTester class given with the assignment. ReplaceAllTester initializes an array of integers of size n configured according to input parameters. If an oldValue argument is provided, ReplaceAllTester reports the number of statements executed to replace that value in an array where all elements are in ascending order from 1 to n inclusive. If no oldValue is provided, ReplaceAllTester reports the average number of statements to replace a randomly-located element within the array. If the optional third numDuplicates argument is provided, that number of duplicate 1s will be randomly distributed in the array and the expectation is that oldValue will be 1, such that the number of statements to replace all duplicate values will be reported. Note that the reported result for any numDuplicates less than n is subject to vagaries of random distribution. ReplaceAllTester usage:*

$ java *ReplaceAllTester* [array size n] <oldValue> <numDuplicates>

## **Minimum Statements**, Constant Factor

What statements are executed in a call to replaceAll() when the array size is zero (n == 0)? Do not overlook statements executed in find() or the assignment of its return value. So what is t(0) for replaceAll(), the minimum cost and constant factor?

Predicted t(0) =

### Run ReplaceAllTester 0 -1

What is your predicted number of statements when n == 0? How do the test 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:

ReplaceAllTester 0 -1 Statements:

Final t(0) =

## Best Case Scenario

Assuming a large array size n, what would cause the replaceAll() while loop to never iterate? What would be the cost of the first find() call? What statements are executed in replaceAll(), itself? What is the total best case growth function t(n) under these conditions?

Predicted tbest(n) =

### Run ReplaceAllTester 100 -1

What is your predicted number of statements when n == 100? How does the number of reported statements compare 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:

ReplaceAllTester 100 -1 Statements:

Final tbest(n) =

## Worst Case Scenario

Assuming n is large, all values in the array equal oldValue, and newValue does not equal oldValue, how many times will the while loop iterate? What is the cost of the first call to find()? What is the cost of the last call to find()? What is the average cost of a find() call within the while loop? What other statements are executed in every iteration of the while loop? What is the total worst case growth function t(n) under these conditions?

Predicted tworst(n) =

### Run ReplaceAllTester 100 1 100

What is your predicted number of statements when n == 100? How does the number of reported statements for the actual worst case 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 tworst(100) Statements:

ReplaceAllTester 100 1 100 Statements:

Final tworst(n) =

## Expected Case Scenario

Assuming a large, randomly ordered array of ***unique*** elements and oldValue is a value in the array, how many replaceAll() while loop iterations will occur? What is the expected cost of the first call to find()? What is the expected cost of the second call to find()? What is the expected growth function t(n) for replaceAll() under these conditions?

Predicted tavg(n) =

### Run ReplaceAllTester 100

What is your predicted number of statements when n == 100? How does the number of reported statements compare 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:

ReplaceAllTester 100 1 100 Statements:

Final tavg(n) =

## **Order**

What is the runtime order (big-O) of replaceAll() based on the above growth functions?

O()

# Algorithm: sortIt()

*Testing questions in this section refer to the SortItTester class given with the assignment. SortItTester 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. SortItTester usage:*

$ java *SortItTester* [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 sortIt() when the array size is zero (n == 0) or one (n == 1)? So what is t(0) and t(1), the minimum cost and constant factor for sortIt()?

Predicted t(0 or 1) =

### Run SortItTester 0 a and SortItTester 1 a

How does the number of reported statements compare with your expectations? 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 t(0 or 1) Statements:

SortItTester 0 a Statements:

SortItTester 1 a Statements:

Final t(0 or 1) =

## **Best Case Scenario**

Assume a large array size n and elements in the array are already in ascending sorted order. The sortIt() outer loop depends only on n, but the inner loop is sensitive to the ordering of elements in the array and the current index of the outer loop. How many times will the outer loop iterate? How many times will the inner loop iterate? What statements are executed in every iteration of the outer loop? What is the growth function under these conditions?

Predicted tbest(n) =

### Run SortItTester 100 a and 100 m 100

What is your predicted number of statements when n == 100? How does the number of reported statements compare with your expectation? (Were you surprised that there are two “best case scenarios”?) 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? *(Note that the inner loop condition could be legitimately counted as 1, 2, 3, or even 4 statements. SortItTester compromises and counts the inner loop condition as 2 statements.)*

Predicted tbest(100) Statements:

SortItTester 100 a Statements:

SortItTester 100 m 100 Statements:

Final tbest(n) =

## **Worst Case Scenario**

Assume a large array size n and elements in the array are arranged in descending order. The sortIt() outer loop depends only on n, but the inner loop is sensitive to the ordering of elements in the array and the current index of the outer loop. How many inner loop iterations would there be when next == 1? How many inner loop iterations would there be when next == array.length - 1? What is the average number of inner loop iterations per outer loop iteration under these conditions? What statements are executed for each iteration of the inner loop? What is the total worst case t(n) for sortIt() under these conditions?

Predicted tworst(n) =

### Run SortItTester 100 d

What is your predicted number of statements when n == 100? How does the number of reported statements compare 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? *(Note that the inner loop condition could be legitimately counted as 1, 2, 3, or even 4 statements. SortItTester compromises and counts the inner loop condition as 2 statements.)*

Predicted tworst(100) Statements:

SortItTester 100 d Statements:

Final tworst(n) =

## **Expected Average Case Scenario**

Assume a large array size n and the array contains unique elements in random order. How does the expected average number of inner loop iterations per outer loop iteration compare to the worst case? Why? How many inner loop iterations are expected on average? What is the total expected t(n) growth function for sortIt() under these conditions?

Predicted tavg(n) =

### Run SortItTester 100 r

What is your predicted number of statements when n == 100? How does the number of reported statements for a random case align with your expectation? (You may want to run the test several times.) 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? *(Note that the inner loop condition could be legitimately counted as 1, 2, 3, or even 4 statements. SortItTester compromises and counts the inner loop condition as 2 statements.)*

Predicted tavg(100) Statements:

SortItTester 100 r Statements:

Final tavg(n) =

## **Order**

What is the runtime order (big-O) of sortIt() based on the above growth functions?

O()