CS 221 Analysis of Algorithms Homework

Morgan Zabriskie

*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.*

*Empirical results to compare with your predicted results come from the pre-compiled AoATester class given with the assignment. Run AoATester directly from the command line. AoATester configures an array of integers appropriate for the specified method and use case and reports the actual number of executed statements. The first command line argument specifies the method to test. The second argument specifies the use case. The optional third argument specifies the length of the array, which must be a positive integer. For the minimum statements use case, the third argument is ignored, even if a value is given. For other use cases, the length defaults to 100 unless specified otherwise.  
AoATester usage:*

$ java AoATester <find|replaceAll|sortIt> <min|best|worst|expected> [array length]

# Algorithm: find()

## 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?

When the program enters the find() method with an array size of zero, it immediately enters the for loop and (1) initializes int i and (2) checks if i is less than the array length. With an array of zero, i is not less than 0 so the for loop will exit. It moves on to the return statement but we shouldn’t count those, so that is 2 statements.

Predicted t(0) = 2

### Run: AoATester find min

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

AoATester find min Statements: 2

My predicted t(0) statements was 2 and the program also returned 2, which means my analysis of statements with an empty array was correct.

Final t(0) = 2

## 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?

If the target element is in the first location of the array (index 0), then the for loop will be entered and (1) int i will be initialized to zero and (2) the for loop will check that i is less than the array length. Since the array length in this scenario is large, this pass would check and we would enter the for loop. We know in this scenario that the first array location contains the target element, which means we don’t have to iterate through the array and therefore don’t have an n value. So far, tbest(n) = 2.

In the for loop, there is immediately an if statement checking if the element at location i matches our target value. In this scenario, it does match this target value so the method returns i. This if statement does not iterate through the entire array, so it is a constant 1.

In this best case scenario, even if we have a large array size n, we only have to look at the first element in the array before exiting the method. So our tbest(n) would be 2 + 1, which is 3.

Predicted tbest(n) = 3

### Run: AoATester find best 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: 3

AoATester find best 100 Statements: 3

The tester gave the same value that I predicted, which was 3. This makes sense because regardless of array length, we only have to look at the first value before we exit the method. Because of this, tbest is a constant and shouldn’t contain an n term.

Final tbest(n) = 3

## 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?

In the worst case, we have to iterate through the entire array before finding that our target value is not in our set, which would cause the program to return a -1.

In each for loop iteration two statements will always happen: (1) int i will be initialized to zero and (2) the for loop will check that i is less than the array length. We will do each of these for every element in the array, so this would be 2n.

Entering the if statement, we will compare if our current array value equals the target value. It doesn’t, so we exit and continue along in our for loop. The if statement has a value of 1.

In total, our tworst(n) = (2n)(1), which is the same thing as 2n.

Predicted tworst(n) = 2n

### Run: AoATester find worst 100

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: 200

AoATester find worst 100 Statements: 302

My prediction and the answer given by the tester were pretty different. Looking back at the code, I can see where my first error lies. In the for loop, we have three actions: initializing i (which happens only once, during our first loop), checking if i is less than our array length (which happens once initially and then at the end of every loop), and incrementing i (which happens at the end of every loop). In the first two examples, we never made it all the way through the for loop so I hadn’t taken those factors into consideration.

This would mean we have two initial actions regardless of array size, then 3 actions (check the if statement, increment i, check i is less than array length) that happen for every element in the array. This would give us the statement 2 + 3n.

Final tworst(n) = 2 + 3n

## 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?

In the average case, we would iterate through our for loop (n/2) times. We start with our two actions that happen regardless of array size (initializing i and checking i is less than array length), and every time that we go through the entire for loop/if statement we do three actions. This would give us a value of 2 + (3/2)n. When we reach our target value, we would return the value of i and not do our final two actions in the loop (incrementing i and checking that it’s less than array length), so we would have 1 additional action our final time through. This would give us a taverage of 2 + (3/2)n + 1, which is (3/2)n + 3.

Predicted texp(n) = (3/2)n + 3.

### Run: AoATester find expected 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 texp(100) Statements: 153

AoATester find expected 100 Statements: 151.5

My prediction and the answer from the tester are very close. The difference is 1.5, which I suspect is coming from my constant 3. This would mean my constant would need to be 1.5 instead of 3, which implies that it should be cut in half. Also, since this is an average over many run times it seems like the answer will never be exact.

Thinking about it some more, it makes sense to start with our tworst, which is 2 + 3n, and add 1 additional constant to account for the last run through the array, where we would add 1 statement with the if statement conditional check. This would give us 3 + 3n. Then, since this is supposed to be an average over many times using this method, we would take our worst case and divide it by 2. This would give us (3 + 3n) / 2. We can test t(100), which is 151.5, which matches the tester statements.

Final texp(n) = (3 + 3n) / 2

## Order

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

O(n)

# Algorithm: replaceAll()

## 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?

If the array size is zero, we will start by passing that array and target value to the find() method. In that method, with an array of zero, we would have 2 statements (see explanation above).

Find() would return -1, which becomes our index value in the replaceAll() method. We would do our first check in the while statement, which would be false since index of -1 is not greater than -1. Because the while statement is false, we would exit the method and no values would be replaced.

The find() method gives us 2 statements and the body of replaceAll() gives us 1 statement, so total would be 3 when the array is empty.

Predicted t(0) = 3

### Run: AoATester replaceAll min

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: 3

AoATester replaceAll min Statements: 4

I was off by one in my prediction – I see the missing statement is when we return the value from the find() function, that is an additional statement. I was counting that entire line as one, when really that entire line is three statements. So it should be find() = 2 statements, initializing index = 1 statement, and checking the while loop = 1 statement, which in total is 4 statements.

Final t(0) = 4

## 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?

In order for the replaceAll() loop to never iterate, we would need the oldValue to not exist in the array at all. This would mean we would call find() once, see that the value isn’t in the array, index would be -1 which would cause the while loop to fail. We would then exit the method and it wouldn’t loop.

The tbest requires that we iterate through the entire array of n elements, which for find() is tworst from our analysis above (2 + 3n). After we have looped through all our elements in find(), we will initialize index (+1) and check our while loop (+1), which will be false and will end the method. This means our total is 2 + 3n + 1 + 1, or 4 + 3n.

Predicted tbest(n) = 4 + 3n

### Run: AoATester replaceAll best 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 do you need to modify about your analysis to better align with the empirical results?

Predicted tbest(100) Statements: 304

AoATester replaceAll best 100 Statements: 304

My prediction and the tester statement value were the same, so my analysis was correct.

Final tbest(n) = 4 + 3n

## 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?

If we are looking at the worst case scenario, then we will have to replace every value in the array with the new value. When we do each find(), we will loop through the array until we find the next oldValue. At first this will be the best case for find() with the element in the first spot in the array, and at the end it will be the worst case for find() where we don’t have any oldValues left in the array. Assuming this, we can use average case for find() in the while loop.

Our first run through will start with 1 statement for initializing index and the best case for find(), which is 3. We will also have +1 statement to check the while loop. So before the first while loop, we have a constant of 5.

Then once we’re in the while loop, we always have two statements (setting array[index] to newValue and setting index to the find() return statement). So this is a constant 2. We then will run through find(), and we can use the average case. This is (3 + 3n) / 2. Then we will check the while loop again, adding another 1. So the while loop has a value of 3 + ((3 + 3n) / 2), which we will run n times (for each value in the array). So this is n(3 + ((3 + 3n) / 2)), which can be simplified down to 3n + 1.5n +1.5n2, or 1.5n2 + 4.5n.

Adding the initial statements that we’ll always do, which is 5, we get a total statement of 1.5n2 + 4.5n + 5.

Predicted tworst(n) = 1.5n2 + 4.5n + 5

### Run: AoATester replaceAll worst 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: 15,455

AoATester replaceAll worst 100 Statements: 15,754

I was close with my prediction but not quite right. The first factor, 1.5n2 appears to be right because it gets us to the correct order of magnitude at 15,000. I think this means my n factor is not quite right, and my constant also is off by 1. This means that one of the constants I was including in the initial for loop should be counted as part of the recurring while loop. I also wasn’t accounting for the final find() check that would add two statements where we would iterate i and check the for loop condition, then exit when it is false.

Redoing my analysis:

Our first run through will start with 1 statement for initializing index and the best case for find(), which is 3. So before the first while loop, we have a constant of 4.

Then once we’re in the while loop, we always have three statements (setting array[index] to newValue, setting index to the find() return statement, and checking for the while loop). So this is a constant 3. We then will run through find(), and we can use the average case. This is (3 + 3n) / 2. Then we will check the while loop again, adding another 1. So the while loop has a value of 4 + ((3 + 3n) / 2), which we will run n times (for each value in the array). On our final run through find() we will add 2 additional statements, when we iterate i and check the condition which will be false So this is n(6 + ((3 + 3n) / 2)), which can be simplified down to 6n + 1.5n +1.5n2, or 1.5n2 + 7.5n.

Adding the initial statements that we’ll always do, which is 4, we get a total statement of 1.5n2 + 7.5n + 4.

Final tworst(n) = 1.5n2 + 7.5n + 4

## 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?

In the expected case, we are assuming that there is only one oldValue to replace. I would also assume that on average, oldValue would not be in the first location in the array. The expected cost of this call to find() should be the average t(n) for find(), along with the constant 1 statement that we always get for the first part of the replaceAll() function. This would be 1 + ((3 + 3n) / 2).

Then we would enter the while() loop. We have our first constant statements, checking the while loop, setting the array index, and setting the new index value. So 3 statements total. We then have another call to find(). In this case, we would look through the whole array and not find our value, returning -1. This is the worst case for find(), which is 2 + 3n. Then we would check the while loop one more time, see that it is not true, and exit the method. This would add 1 more constant statement. So this would be 4 + (2 + 3n), which is 6 + 3n.

Adding these together, we would get 1 + ((3 + 3n) / 2) + 6 + 3n, and simplified down this is 4.5n + 8.5

Predicted texp(n) = 4.5n + 8.5

### Run: AoATester replaceAll expected 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 texp(100) Statements: 458.5

AoATester replaceAll expected 100 Statements: 458.5

My prediction was correct! So my analysis of what would happen if there was only one value to replace in the array was correct.

Final texp(n) = 4.5n + 8.5

## Order

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

O(n2)

# Algorithm: sortIt()

## 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()?

When the array size is zero, the initial for loop initializes the next variable and checks that is less than the array size. In this case it isn’t true, so the method exits and we have run 2 statements. For array size of 1, this is the same because we initialize the for loop variable to equal 1 instead of 0. So it would still not be less than the array length and we would exit the method having run 2 statements. So t(0 or 1) = 2.

Predicted t(0 or 1) = 2

### Run: AoATester sortIt min

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: 2

AoATester sortIt min Statements: 2

My prediction and the tester statement count are the same, so my analysis of what would happen when the array length is 0 or 1 is correct.

Final t(0 or 1) = 2

## 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?

In the case where all elements are already in ascending order, we would start with our initial 2 statements for the for loop to initialize next and check that the for loop condition is met. In this case with a large array size, it would be met and we would enter the for loop. Total statements is 2.

We then enter the for loop and have 3 statements that will always execute – setting int value, setting int index, and checking that our while loop condition is met. I’m not sure if this while loop should be counted as 1 or 2 statements, since it is checking 2 conditions. I will count it as 1 for now.

In this scenario the while loop will never be true (value will always be greater than the value behind it in the array), so we will never enter the while loop. We then have 1 additional statement to set the value of array[index] to value. In this case it’s just setting the same value again.

At the end of the for loop we will have 2 more statements to iterate next and to check the for loop condition. This is a total of 6 statements that will happen n times because we will check that every number in the array is in the right order. So this is 6n.

In total, the best case scenario is 2 + 6n.

Predicted tbest(n) = 2 + 6n

### Run: AoATester sortIt best 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? *(Note that the inner loop condition could be legitimately counted as 1, 2, 3, or even 4 statements. AoATester compromises and counts the inner loop condition as 2 statements.)*

Predicted tbest(100) Statements: 602

AoATester sortIt best 100 Statements: 695

My predicted answer the tester statement total were a bit off. If I had read a little further down, I would see that the tester is counting the inner while loop condition has 2, and I counted it as 1. Adjusting for that, the new statement would be 7n + 2. Using that equation, the number is still a bit too high and I can’t figure out why. I’m going to set up a breakpoint and run through the debugger because I think something is happening on the last pass through that I’m not guessing correctly.

I see now that we aren’t running through the loop n times, we’re running through it (n -1) times because we’re starting at the second element of the array, not the first. So the equation in the for loop is 7(n – 1), or 7n – 7. The overall equation is 7n + 2 – 7, which is 7n – 5. This equation gives the correct value of 695 using an array size of 100.

Final tbest(n) = 7n - 5

## 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?

In the worst case, we would run through every value in the array to check that it is in the right order (outer loop), but we would also have to run through the inner while loop multiple times to get the element in the correct order. The number of times we run through the while loop would grow with each element, so that at the last value of the array we would be running through the entire n elements of the array.

We start with our initial 2 statements for the for loop to initialize next and check that the for loop condition is met. In this case with a large array size, it would be met and we would enter the for loop. So every time this function is run, we add a constant + 2 to the equation.

We then enter the for loop and have 4 statements that will always execute – setting int value, setting int index, and checking that our 2 while loop conditions are met. At the end of the for loop we have 3 more statements, setting the final index value, incrementing the for loop variable, and checking the for loop is correct. This is a total of 7 statements that happen (n – 1) times (because we start looking at the second element in the array, not the first), so our total statement right now is 7(n -1) + 2, or 7n – 5.

When we enter the while loop, we have 4 statements that execute every time, setting array[index], decrementing index, and checking our two while statement conditions. This happens n number of times (the first time, when index = 1 we will only go through the while loop 1 time, when index = n we will run through the while loop n times). On average this would be n / 2 times. Our overall equation is 4(n / 2), or 2n.

Including this with our for loop, we get (7n – 5)(2n), or 14n2 – 10n.

Predicted tworst(n) = 14n2 – 10n

### Run: AoATester sortIt worst 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? *(Note that the inner loop condition could be legitimately counted as 1, 2, 3, or even 4 statements. AoATester compromises and counts the inner loop condition as 2 statements.)*

Predicted tworst(100) Statements: 139,000

AoATester sortIt worst 100 Statements: 20,495

My prediction and the tester statement total are way off! I’ve added code to check for numstatements and my code is giving the same value as the tester. So I know my assumptions for number of statements is correct, but the way I’m putting them together in the equation is wrong.

I know that the first part of the method will give us +2. Then the for loop gives us 7n – 7, so that is total 7n – 5. Adding in the while loop, I know that the number of statements grows each time we increase our place in the array. Writing this out I see that our total statements in the while loop are 1, 3, 6, 10, 15, etc as we grow the array. I looked up what this pattern is, and it’s called a triangular number sequence and the equation for that is (n(n + 1)) / 2. Our n is actually (n – 1) because we start at the second number in the array in our for loop. So replacing n with (n - 1) in the triangular number sequence equation gives us ((n – 1)((n – 1) + 1)) / 2, and simplified down this is (n2 – n) / 2. There are four statements that execute, so the while loop equation is 4(n2-n) / 2, or 2n2 – 2n.

Adding all these statements together, you get 7n – 5 + 2n2 – 2n, or 2n2 + 5n – 5. Putting 100 in this equation gives us the correct answer of 20,495.

Final tworst(n) = 2n2 + 5n - 5

## 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?

The way this method works, we will always sort through every element of the array to check that everything is in the right order. So we will still use 7n – 5 to account for the for loop.

In this case the while loop is not always going to have to adjust every number, because some might already be in ascending order with their neighbor. So we can assume that this will be the worst case divided by 2, which would be (2n2 – 2n) / 2 or n2 – n.

Adding these all together you get 7n – 5 + n2 – n. Simplified down this is n2 + 6n – 5. At first glance this makes sense because it’s going to give us a number somewhere between the best and worst case, which is what the average should do.

Predicted texp(n) = n2 + 6n - 5

### Run: AoATester sortIt expected 100

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. AoATester compromises and counts the inner loop condition as 2 statements.)*

Predicted texp(100) Statements: 10,595

AoATester sortIt expected 100 Statements: 10,595

My predicted result and tester results are the same! I’m excited because I struggled through the best and worst case scenarios for this method. Figuring out we were running the loops (n -1) times not n times was a big hurdle to getting these right. My analysis for the average case makes sense.

Final texp(n) = n2 + 6n - 5

## Order

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

O(n2)