CS 221 Analysis of Algorithms Homework

Reagan Ovechka

*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()

Constant Factor

Initial Analysis

What statements are executed in a call to find() before reaching a return statement when the array size is zero (n == 0)? (Exclude initialization of method parameters or return statements.)

Statements when size==0: int i =0; (creation of int i), i < array.length; (check condition)

The number of statements when n == 0, the minimum possible number of statements, is the constant factor for the find() growth function. What is t(0) for find()?

Predicted t(0) = 2

Compare and Resolve

Change the font of the next line to black to reveal the actual constant factor:
t(0) = 2
How does your prediction compare to the actual constant factor?

My prediction was correct to the actual constant factor.

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 actual constant factor?

Best Case Scenario

Initial Analysis

The best case scenario for any algorithm describes the contents or organization of an input that results in the fewest possible number of statements being executed for a large input size n. For find(), the best case scenario is when the target element is located at index 0. How many loop iterations will occur if the target element is found at index 0?

One loop iteration will occur, but it will check the conditional statement twice.

What statements are executed before the index is returned? (Exclude initialization of method arguments or return statements.) Are there any statements in find() that will never be reached?

Statements: int i = 0; (initialization of i), i < array.length; (condition check), array[i] == value; (conditional check)

The best case growth function reflects the sum of all statements executed for any large input size n under the best case conditions. What is your predicted best case growth function t(n)?

Predicted $t_{best}(n) = 3$

Compare and Resolve

Change the font of the next line to black to reveal the actual best case growth function:
$t_{best}(n) = 3$
How does your prediction compare to the actual best case growth function?

My prediction was accurate to the best case growth function.

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 actual best case growth function?

Worst Case Scenario

Initial Analysis

The worst case scenario for any algorithm describes the contents or organization of an input that results in the maximum possible number of statements being executed for a large input size n. For find(), the worst case scenario is an input where the method returns -1. Assuming a large array size n, what would be necessary such that the method returns -1?

For the method to return -1, either the array is size 0, or none of the elements in the array match the target value.

How many times does the loop iterate in the worst case scenario for large array size n?

3n

What statements are executed in each loop iteration?

Statements: int i =0; i < array.length;

The worst case growth function reflects all statements executed for any large input size n under the worst case conditions. Multiply the number of loop iterations by the number of statements per iteration and add the constant factor to get the complete growth function. What is your predicted worst case growth function t(n)?

Predicted $t_{worst}(n) = 2 + 3n$

Compare and Resolve

Change the font of the next line to black to reveal the actual worst case growth function:
$t_{worst}(n) = 3n + 2$
How does your prediction compare to the actual worst case growth function?

My prediction was accurate to the actual worst case growth function.

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 actual worst case growth function?

Expected Average Case Scenario

Initial Analysis

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**

n/2

What is the expected average number of loop iterations under these conditions?

n/2*3

What statements are executed in each complete loop iteration? Are there any loop statements that will **not** be executed when the target is found?

Each iteration executes: int i =0; i < array.length; if (array[i] == value); i++;

Not executed if target is found: i++

Multiply statements executed in each loop iteration by the expected number of loop iterations. Add statements that occur before the loop and account for any statements that aren't executed when the method exits. What is the expected average growth function t(n) under these conditions?

Expected number of loop iterations: n/2(3)

Occur before the loop(constant): 2

Predicted $t_{avg}(n) = 2+n/2(3)$

Compare and Resolve

Change the font of the next line to black to reveal the actual average case growth function:
$t_{avg}(n) = floor(3n/2 + 3/2)$
That is probably not exactly what you predicted, but how does your prediction compare to the actual function?

My prediction is similar because they both contain 3n/2, but the constant value is different.

Some of the weirdness, here, has to do with code not conforming to continuous functions. You can't have half a statement. A statement either gets executed or it doesn't. We won't worry about non-continuous functions going forward. If your prediction's constant factor was only off by ½, up or down, you really got close enough. If not, what (if anything) do you need to modify about your analysis to better align with the empirical results?

I need to consider the return that happens in this scenario, the return will happen at some point, making the constant 3 instead of 2.

Order

Identify the largest factor from your growth functions and eliminate the coefficient, if any. What is the runtime order (big-O) of find()?

O(n)

Algorithm: replaceAll()

Constant Factor

Initial Analysis

What statements are executed in a call to replaceAll() when the array size is zero (n == 0)? Do not overlook statements executed in calls to find().

1. Int Index = find(array, oldValue); (Initialization of Index)
2. find(array, oldValue)
    a. We know that when length = 0, find executes 3 statements
3. Index > -1 : (checks conditional)

Conditional was false, so method ends. Total of 5 statements

So what is t(0), the constant factor, for replaceAll()?

Predicted t(0) = 5

Compare and Resolve

Change the font of the next line to black to reveal the actual constant factor for replaceAll():
t(0) = 4
How does your prediction compare with the actual constant factor?

My prediction was 1 more than the actual.

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 actual constant factor?

I believe that my analysis was one number more than the actual because I counted int index and the call to the find method as two separate statements, when its really just one singular statement.

Best Case Scenario

Initial Analysis

The best case scenario for replaceAll() avoids ever entering the while loop in replaceAll(). Assuming a large array size n, what would cause the replaceAll() while loop to never iterate?

The best case scenario is that the find function returns -1 because the old value does not exist in the array

What would be the cost of the first find() call? What find() case does this represent?

The cost is 3n+2, this represents the worst case for find().

What other statements are executed in replaceAll(), itself, under these conditions?

The only other statement that is ran in the replaceAll() method is the conditional check in the while loop: (index > -1). This is only checked once in this scenario.

Add the cost of all statements under best case conditions and combine like terms. What is the total simplified best case growth function t(n) for replaceAll() under these conditions?

3n+2+1

Predicted $t_{best}(n) = 3+3n$

Compare and Resolve

Change the font of the next line to black to reveal the actual best case growth function for replaceAll():
$t_{best}(n) = 3n + 4$
How does your prediction compare with the actual best case growth function?

My prediction was off by one in the constant.

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 actual growth function?

The discrepancy occurred because I was not counting the call to find() as a separate statement from the method running, I was making them the same statement. If I counted the call to the method and the execution of the method as separate, I would get the actual answer.

Worst Case Scenario

Initial Analysis

The worst case scenario for replaceAll() would maximize the number of while loop iterations and calls to find(). 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 has to happen before the while loop can exit?

The worst case scenario is that the array only contains occurrences of old value and they all need to be replaced.

Before the while loop can exit, every value needs to be replaced and find() needs to return -1.

The statements:

- Index > -1: 1
- Array[index] = new value: 1
- Index = find(array, old value): 1
    - Method executed: 3n+2

The loop iterates n(3n+4) times

What is the cost of the first call to find() before reaching the while loop under these conditions?

3n + 4, this is the call and execution of find()

What is the cost of the first call to find() within the while loop?

The cost of the first call to find() in the while loop is also 3n + 4 because the find() method does not change in this time, so I will still iterate the same amount of time.

What is the cost of the last call to find() within the while loop?

The last call to find is also the same, 3n + 4

What is the average cost of a find() call within the while loop?

The elements cost 3n + 4, but to find the average cost, n now equals (n+1)/2, giving us 3(n+1)/2 + 4. The average then turns out to be (3n + 11)/2

What other statements are executed in every iteration of the while loop?

The other statements executed in every iteration is the check of the conditional statement and array[index] = newValue;

Multiply the average cost of all statements within the while loop by the number of while loop iterations. Add this to all statements prior to the while loop and simplify the function, combining like terms. What is the total worst case growth function t(n) under these conditions?

n((3n + 11)/2 + 2) + (3n + 4)

(3n^2 + 15n)/2 + (3n + 4)

*simplify

Predicted $t_{worst}(n) = (3n^2)/2 + (21n)/2 + 4$

Compare and Resolve

Change the font of the next line to black to reveal the actual worst case growth function:
$t_{worst}(n) = (3/2)n^2 + (15/2)n + 4$
How does your prediction compare to the actual worst case growth function for replaceAll()? Did you have the same factors, but coefficients were off? If coefficients were off, how close were they?

My prediction was pretty close!

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 actual worst case growth function?

I think I may have calculated one extra statement in the 3n + 4 when finding the average, which led for a higher value in the worst case calculation.

Expected Case Scenario

Initial Analysis

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?

Only one iteration will occur, because saying that oldValue is *a* value in the array implies there is only one iteration, meaning that the loop will only occur once for the single value being replaced

What is the expected cost of the first call to find(), before the while loop?

3n + 4

What is the cost of the last call to find(), within the while loop?

3n + 4

What is the expected growth function t(n) for replaceAll() under these conditions?

(3n + 4 before the while loop) + (3n + 6 in the while loop)

Predicted $t_{avg}(n) = 6n + 10$

Compare and Resolve

Change the font of the next line to black to reveal the actual expected average case growth function:
$t_{avg}(n) = (9/2)n + 7$
How does your prediction compare to the actual expected average case growth function?

It is not close unfortunately, besides the n.

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 actual growth function?

I think it is not close because I see now that it wanted the average, and I did not replace n with n+1/2

Order

Identify the largest factor from your growth functions and eliminate the coefficient, if any. What is the runtime order (big-O) of replaceAll()?

O(n^2)

Algorithm: sortIt()

Minimum Statements, Constant Factor

Initial Analysis

What statements are executed in a call to sortIt() when the array size is zero (n == 0) or one (n == 1)?

- int next = 1; next < array.length;

What is the t(0 or 1), the constant factor for sortIt()?

Predicted t(0 or 1) = 2

Compare and Resolve

Change the font of the next line to black to reveal the actual constant factor for sortIt():
t(0 or 1) = 2
How does your prediction compare with the actual constant factor?

My prediction matched the actual

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 actual constant factor?

Best Case Scenario

Initial Analysis

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?

7n

How many times will the inner loop iterate under best case scenario conditions?

Zero times because everything is in the correct order, so the while loop condition will never be true.

What statements are executed in every iteration of the outer loop? *(Note that the compound inner loop condition could be legitimately counted as 1, 2, 3, or even 4 statements. I am compromising and counting the inner loop condition as 2 statements.)*

- Next < array.length;
- next++
- int value = array[next];
- Int index = next
- While (index > 0
- Value < array[index - 1])
- Arrat[index] = value

Working inside-out, multiply the number of statements within each loop by the number of loop iterations under best case conditions. Add this to statements prior to the loop. Simplify the final function by combining like terms. What is the best case growth function for sortIt()?

Because next starts at 1, it iterates 1 less times than the length of the array, meaning that we can subtract 7 statements

The statement next = 1 only runs once, meaning we can add 1

7n - 7 + 1

Predicted $t_{best}(n) = 7n - 6$

Compare and Resolve

Change the font of the next line to black to reveal the best case growth function for sortIt():
$t_{best}(n) = 7n - 5$
How does your prediction compare to the actual best case growth function?

My prediction is one number off of the actual best case growth function.

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?

I didnt count the final failed condition check in the for loop, meaning I was off by 1 in the constant.

Worst Case Scenario

Initial Analysis

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 (the first time through the inner loop)?

The while loop will iterate only once because it switched the value from index 1 to index 0, then incremented index - -, index now == 0 and the while loop does not repeat because the first condition is not met.

How many inner loop iterations would there be when next == array.length - 1 (the last time through the inner loop)?

The inner loop will iterate n - 1 times in this case because the last element needs to move all the way to index 0. The element has to move n-1 indexes to the left, and to do that the loop needs to iterate n-1 times to get it to index 0.

What is the average number of inner loop iterations per outer loop iteration under these conditions?

In this scenario, for next = k, the inner loop iterates k times. The total iterations then is the sum over k where k = 1…. n-1, this equals n(n-1)/2. To find the average number of inner loop iterations per outer loop iteration, we divide the total number of inner loop iterations by total outer loop iterations: (n(n-1))/(n-1)

What statements are executed for each iteration of the inner loop? *(Note that the compound inner loop condition could be legitimately counted as 1, 2, 3, or even 4 statements. I am compromising and counting the inner loop condition as 2 statements.)*

- Condition statement X 2
- Array[index] = array[index - 1]
- Index--

Working from inside to outside, multiply the statements in each loop by the number of loop iterations and add to statements outside the loop. Simplify and combine like terms. What is the total worst case t(n) for sortIt() under these conditions?

Total inner loop: n^2-n/2

Total outer: n-1

Statements inside inner: 4

Statements inside outer: 5

(4 * n^2-n/2) + 5(n - 1) → 2n^2 - 2n +5n - 5

Predicted $t_{worst}(n)$ = 2n^2 - 2n +5n -5

Compare and Resolve

Change the font of the next line to black to reveal the actual worst case growth function for sortIt():
$t_{worst}(n) = 2n^2 + 5n - 5$
How does your prediction compare with the actual worst case growth function?

My prediction was accurate to the actual worst case growth function.

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 actual worst case growth function?

Expected Average Case Scenario

Initial Analysis

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?

The average positioning of the final element that is inserted is n-1/2 because it is evenly distributed through the array and expects only half the numbers need to move. The number of inner loop iterations is now half of what it was previously, (n(n-1)/2)/2, or n^2-n/4. This is different from the worst case because in the worst case it is expected to be the complete opposite away, whereas the average expects it to be only halfway in the opposite direction of where it needs to be moved.

Again, working from the inside out, multiply statements in each loop iteration by the expected average number of loop iterations and add to statements outside the loop. Simplify and combine like terms to get the total expected t(n) growth function for sortIt() under the expected average conditions. *(Note that the compound inner loop condition could be legitimately counted as 1, 2, 3, or even 4 statements. I am compromising and counting the inner loop condition as 2 statements.)*

Average inner loop iterations: n^2-n/4

Average outer loop iterations: n-½

Statements in inner: 4

Statements in outer: 5

(4 * n^2-n/4)+(5 * n-1 / 2) → n^2 - n + 5n/2 - 5/2

Predicted $t_{avg}(n)$ = n^2 + 3/2n - 5/2

Compare and Resolve

Change the font of the next line to black to reveal the actual average growth function for sortIt():
$t_{avg}(n)$ = n^2 + 6n - 5
How does your prediction compare to the actual average growth function for sortIt()?

My prediction was pretty close, I was just off on the 5/2n and the 5/2

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 actual expected average case growth function for sortIt()?

I think that the discrepancy occurred In the calculation of the average for the outer loop, because my prediction would be accurate if it was multiplied by 2 in the average outer loop iterations.

Order

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

O(n^2)