# Analysis of Algorithms

Defend all answers based on specific references to the code. Do not count return statements or initialization of method arguments. You are strongly encouraged to walk through algorithms in the debugger and to add statement-counting code to given methods to test and refine your analysis.

## Algorithm: find()

### Minimum Statements:

* How many statements would be executed in a call to find() when the array size is zero (n == 0)?
  + If find() was called with an array size of zero the method would initialize i and then find that i is not less than the array size which would cause it to skip the body of the for loop and return -1. The initialization, check, and return constitutes 3 statements.

### Best Case Scenario:

* Under what conditions would the minimum number of statements be executed for an array where n is large?
  + Since the method returns the index where array[i] == value, and the iterator starts at 0, the best case scenario would be for array[0] == value. This would run the same amount of statements regardless of the size of n.
* Where would the target element be located in the array?
  + As stated above, the target would be located in the 0 index of the array for the best case, for a generally good case the target should be located early in the array.
* What is the growth function under these conditions?
  + Under these conditions we would see s = 4. This scenario is agnostic to the size of the array as the return will always be on the first checked index.

### Worst Case Scenario:

* Under what conditions would the maximum number of statements be executed for an array where n is large?
  + Since the method returns the index where array[i] == value, and the iterator starts at 0, the worst case scenario would be array[n] == value. This would necessitate the maximum amount of loops to match the value.
* Where would the target element be located?
  + As above, the target would be located at the n index of the array.
* What is the growth function under these conditions?
  + In this there are 2 non looping statements, the instantiation of i and the return statement. This gives us $s = 2 + x$ where x is the multiple of the looping statements times the amount of times looped. Since there are 3 looping statements and we are looping n times we get $s=2+3n$

### Expected Average Case Scenario:

* Assuming a random array of unique elements and the target element is in the array, where would a target element be located on average?
  + In a randomized data set the number we are looking for will average to the halfway value. With those assumptions we can say that the average position for the target element is array[n/2].
* What is the expected average number of statements (the expected growth function) for a call to find()?
  + Since we determined that the growth on the maximum is $s=2+3n$ we can use that in conjunction with the average position. $s=2+(3n/2) = 2+1.5n$.
* What is the runtime order (big-O) of find() based on the above growth functions?
  + $O(2+1.5n)$

## Algorithm: replaceAll()

### Minimum Statements:

* How many statements would be executed in a call to replaceAll() when the array size is zero (n == 0)?
  + replaceAll() starts by instantiating index to equal find(array, oldvalue)Since we determined that find has 3 statements when the array size is 0 this adds 3 statements. In this case index = -1. This is checked to see if it is greater than -1, as it is not we then skip looping the while loop. I believe this would be 5 statements in this case.

### Best Case Scenario:

* Under what conditions would the minimum number of statements be executed for an array where n is large?
  + The best case scenario would be for there to be no instances of oldValue in the array at all. This would allow us to skip the scary while loop that contains a nested loop inside the the find() call it contains.
* How many occurrences of the oldValue element would be in the array?
  + As above there would not be any oldValue instances within the array.
* Where would it/they be located in the array?
  + oldValue would not be contained in the array as it would avoid any loop outside of looping all the way through the array with find() once. Since find only returns -1 if the target value is not found, and the replace all loop only ends when -1 is returned, we want to do this as soon as possible because an entire run through find() is necessary to end the loop even if there was instances of oldValue in the array.
* What is the growth function under these conditions?
  + In this scenario we are providing the worst possible situation for the find() function. We know that this is $s=2+3n$. The replace() function has 2 non looping statements (the instantiation of index and the statement check for the loop, false in this case). Since we are not looping we get $s=4+3n$.

### Worst Case Scenario:

* Assuming newValue and oldValue are not equal, under what conditions would the maximum number of statements be executed for an array where n is large?
  + For this algorithm the worst case scenario is one where all values are the oldValue as this creates the situation where the while loop loops for every index of the array, calling the find() method to generate i+1 every time.
* How many occurrences of oldValue are in the array?
  + There would be n occurances of oldValue in this case as this causes the most looping.
* Where would it/they be located?
  + In this scenario we have oldValue in every index of the array.
* What is the growth function under these conditions?
  + Since I plotted a lot of data I was able to use it to calculate the trend line as $s=2n^2+10n+5$ This makes sense given the growth formula will be exponential given that the loop in replaceAll() is calling another loop that also will loop more depending on the scenario

### Expected Average Case Scenario:

* Assuming a random array of unique elements and oldValue is a value in the array, what is the average number of statements (the expected growth function) for a call to replaceAll()?
* this one is harder for me to wrap my head around because it depends on the bounds of the random integers to calculate the probability of oldValue even being contained within the array. At a certain point I thought about this enough that my work addled brain thought it was a good idea to ask Deepseek r1 for fun. This gave me T=n(ln(np)+γ). Based off of Hm the harmonic number which is very zen of it. That being said, my gut feeling was an exponential relationship. I charted this with a pivot table based off of arrays with size 1-100 and with a loop in that loop testing duplicates allowed as 1-sizeOfArray. The best fit line for the average steps based on size which gave me an exponential graph. I am possibly overthinking this one and, in an effort to choose my battles, will be moving on to the next question. (if you are reading this then I either did not return or could not come up with anything better.)
* What is the runtime order (big-O) of replaceAll() based on the above growth functions?
* The average runtime order is $o(n^2)$.

## Algorithm: sortIt()

### Minimum Statements:

* How many statements would be executed in a call to sortIt() when the array size is zero (n == 0) or one (n == 1)?
  + When running sortIt() with an array size of zero the output with statements counted is 2.

### Best Case Scenario:

* Under what conditions would the minimum number of statements be executed for an array where n is large?
  + SInce sortIt() loops when the value of the current index is less than the previous, we would want the array to already be sorted ascending each index.
* Would the algorithm execute a different number of statements if the elements in the array were already in sorted order? Reverse order? Random order? All the same value?
  + The algorithm will have far less statements if the array is already sorting ascended as it would not loop through the while loop. Any randomization will make the while loop loop, but not all the time. If the array is sorted in decending order it will force the while loop to loop each time making it the most taxing.
* What is the growth function under the best case conditions?
  + Running my testing program I see that with an ascending array we get a linear dataset. This is a linear line modelled with 2 as the intercept and the statements increasing by 4 for every incrementation of size, in this case we get $s=4n+2$.

### Worst Case Scenario:

* Under what conditions would the maximum number of statements be executed for an array where n is large? (Already in some kind of sorted order? Duplicates?)
  + The worst case scenario would be a descending array, this forces the while loop to loop over as many times as it would take to move the value to a point where the value to the left is less than it is, this forces an ever increasing amount of loops and for each index to loop the maximum amount of times.
* What is the growth function under the worst case conditions?
  + Running this in my testing program to gather data, them putting it in excel I get a trendline modelled as $s=1.5n^2+2.34n+2$.

### Expected Average Case Scenario:

* Assuming a random array of unique elements, what is the expected average number of statements (the expected growth function) for a call to sortIt()?
  + Running this through my testing program and analysing the data I get another exponential relationship, this is significantly better than the worst case though. $s=0.73n^2+4.9n+2$.
* What is the runtime order (big-O) of sortIt() based on the above growth functions?
  + In this case we are looking at an quadratic growth function for the average case. $o(n^2)$.