#

# Review: Big "Oh" notation

**The goal for this exercise** is to review the Big "Oh" notation.

 Last quarter, we looked briefly at the so-called Big "Oh" notation. The basic ideas were as follows:

1. We want a way to compare different algorithms to each other, based on evaluations like "Which one is faster?" or "Which one takes up less space in memory?".
2. We want to be able to compare algorithms early in the process of creating software – quite possibly before much/any software has been written. So we'll need a way of measuring software other than by using a clock to observe how much time it takes to run the algorithm.
3. Further, we don't want our metrics to depend on 'incidental' stuff that will change.
For example, we don't want to have to redo all our measurements, just because a new version of the compiler was released, or we ported the software to a new CPU, or because someone fixed a bug in the operating system.
So we not only want to avoid 'clock-time' measurements of our algorithms, but we also want to avoid measurements that will depend on the particular compiler that we used, or the particular CPU that we used. We're looking for general guidance that we can reuse repeatedly, without having to re-run all our measurements every time someone tweaks the compiler/CPU/etc.
4. We're willing to measure things in an approximate fashion, and we're willing to assume that our comparisons only hold true for fairly large numbers of input values.
Remember that the goal here isn't to use these measurements for performance-tuning actual software, but it's really more to decide which algorithms to use when we're still designing the software.
For example, if you're only got 10 elements in an array, BubbleSort might actually run faster than QuickSort. But for 10 million, a good QuickSort will win, and win by a lot, every time.
5. We'd like to have a mathematically rigorous definition of these metrics, so that we might prove that a given algorithm has certain properties
(Well, we won’t, but if you take a 300/400-level Algorithms class at a 4-year school, you might do such things there)

Thus, our goal will be to find a mathematical expression (a function) that describes the amount of time needed to run the algorithm to completion, as a function of the size of the input. Based on the points that were raised above, we don't actually care about the exact function is – mainly we just care about whether one algorithm will require more or less time than another, for sufficiently large inputs. In other, more math-y words, we want to compare the asymptotes of the two functions. So, we want to compare two algorithms via an "asymptotic analysis" of each one, and then comparing those results to each other.

But really, knowing what we do about asymptotes, we want to compare the asymptotes each algorithm's of the dominant term, WITHOUT any coefficients on that term. Which is where the 'Big "Oh" Notation' comes in – it's essentially a glorified way to dropping any non-dominant terms and/or coefficients.

**Let's look at an example:** Let's say that you've got an array of values, sorted from lowest to highest. Let's say you know that you expect, on average, between 200 and 300 values in the array. You've got two options for finding values in the array: a linear search, and a binary search.

The linear search consists of sequentially examining each and every space in the array, one-by-one, starting with index 0, and going up to however many elements are in the array. Let's say that you somehow knew that if there are N elements in the array (N typically being between 200 & 300, as mentioned above), that the function describing the number of steps (C# statements, CPU instructions, of whatever else you wanted to use as a 'step' in the algorithm) required to find something using a linear search is:



For our purposes, each step takes the same amount of time, so really, the time required is:



A binary search consists of examining the middle value in the array, and if the value you're looking for is larger than that, you repeat the process with the upper half of the remaining elements. If the value is smaller than the median, you repeat the process with the lower half of the remaining elements. You keep repeating until you either find the element, or else end up with no un-searched elements left. (Note: this is a quick summary – feel free to consult the Internet, or BIT 142's section on Binary Search, for a more in-depth explanation) Let's say that you somehow know that the time required by the algorithm is:



If we were to graph these two functions, we'd get a picture like this:



But let's face it – given the incredible difference between those two functions, especially for these 'large' values of N, the coefficients / non-dominant terms don't really matter. We'd get just as much info if we graphed y = N vs. y = log2N

The 'Big Oh Notation' allows us to do exactly this – the Big Oh allows us (in a mathematically rigorous way, that's outside the scope of this class), to ignore all the coefficients / non-dominant terms, like so:

 and 

Graphing these two, we get:



Keeping in mind that our goal was to figure out, at the architectural/design-phase of creating a software product which one to use, and that we expect N to be between 200 & 300, it's clear that binary search is the way to go – it'll end up doing far fewer steps, and thus end up being quicker.

So, the next question is: how does one figure out which function to use? Luckily for us, there's typically only about 5 that we need to consider, and the choice between them is usually pretty clear. They're summarized in the following table:

|  |  |
| --- | --- |
| O(?) | Explanation / Examples |
| O(1) | This called "constant time" – basically, not matter how many input elements there are (no matter how many array elements, etc), we'll never do more than X steps. You'll typically see this in situations wherein you're asked to access a single element of the array (for example, get the first element of the array & return it – it'll take the same amount of time whether there are 10, or 10 million, elements in the array)*Example of Code that run in constant time:** Array access, such as rgNums[10] = 20;
	+ But NOT array allocation – see O(N), below
* Pretty much all operators - 3 + 2, 8 \* 100, 9 + 10 \* (3+4/7)
* Calling a function
	+ Note that the function itself may take more then O(1) time, even if CALLING the function is O(1) time
 |
|  | You typically see log2N run-times whenever you have an algorithm that repeatedly throws away half of the remaining elements. Binary search is a perfect example of this – at each iteration, it ignores ½ of the remaining elements.*Example of Code that run in log base-2 time:** Binary search
* Anything similar to binary search, where the remaining set of values that the algorithm needs to deal with is halved at every step/pass of the algorithm.
 |
| O(N) | Also called "Linear time." Normally occurs whenever you have to access every single element of the array. Accessing every single element of the array twice ends up being Y = 2N, but since the O() notation allows us to ignore coefficients, it ends up being the same thing as O(N). Similarly for accessing each element three times, or each element four times, etc. As long as the number of times we access each element is limited, and unrelated to the number of elements, then we'll end up with a linear runtime.Good examples of this are finding the maximum in an unsorted array, or printing every single element of an array out.*Example of Code that run in linear time:** Any loop that goes from 0 to N
	+ // N is a variablefor( int i = 0; i < N; i++) ...
	+ If N is constant, then it’s technically O(1) time
* Array creation & initialization: When we create an array, C# puts a zero into each space in the array, and therefore it allocates the following code for us.  for (int i = 0; i < array.Length; i++) { array[i] = 0; }Because it’s a loop that goes from 0 to N = length of the array, the following line of code runs in linear time (based on the size of the array that’s being created): int [] array = new int[20];
 |
|  | We'll see this later on for QuickSort. Normally, if you do log2N work, but do it N times, you'll end up with this runtime. |

|  |  |
| --- | --- |
|  | BubbleSort, SelectionSort, etc all have this run-time. In a nutshell, we have to repeatedly go through every single one of the N elements in the array. We have do the repetition a number of times equal to size of the array.*Example of Code that run in linear time:** Any nested loops, where BOTH the outer AND the inner loop goes from 0 to N. For example, here’s BubbleSort:for (int i = 0; i < array.Length; i++){ for (int j = 1; j < array.Length; j++) { // if they're out of order,  // swap them if (array[j - 1] > array[j]) { int temp = array[j]; array[j] = array[j - 1]; array[j - 1] = temp; } }}
 |

|  |
| --- |
| **What you need to do for this exercise:**   In the starter project, within your implementation of the SmartArray\_Enums\_Test, list the O() running times for each of the methods (including the two constructors – give some thought to what the running time should be for each one, and why they’re different) in comments. Make sure to include a quick (1-2 sentence) explanation as to why you chose the running time that you did. You should do this exercise, based on your code, even if you can't get all the methods in the SmartArray to work correctly. In order to make these annotations easy to find for your instructor, please put the explanation in the code, in a comment, and put the comment immediately above the method that you are explaining. This way, there will be a comment explaining what the O() running time of the **SetAtIndex** operation is, immediately above the **SetAtIndex** method, and a comment explaining what the O() running time of the **Find** operation is, immediately above the **Find** method, etc.  Your instructor may have already left space for these comments, in which case you should make sure to add your comments into the provided space. |