Big-O Reading

Tim "Dr. T" Chamillard

Background

Video games regularly push the limits of the hardware platform on which they run. Although the actual limits are different – a game running on a mobile device has much less processor speed and memory available than a game running on a tricked-out desktop – we always want to be as efficient as possible in both these areas. When we start evaluating algorithms for their speed and memory requirements, we're trying to make sure that we're not doing our computing inefficiently.

One of the things you'll keep running into is the classic tradeoff between space and time. No, we're not talking about the space-time continuum … well, we kind of are … anyway, as game developers we mean the tradeoff between memory consumption and processor usage. You'll discover that we can sometimes make a process run really quickly if we use gobs of memory, or alternatively we can make something use a very small memory footprint if we're willing to make the processor do more calculations (and take longer doing them, of course). The first step we need to take to intelligently make that tradeoff is develop some understanding of what the processor and memory usage would be for a particular approach.

That's what algorithm analysis is all about – being able to quantify the complexity of an algorithm so we understand how much time and (more rarely) space the algorithm will take. We'll follow the standard approach of focusing on the time requirements for the algorithms we analyze, with the understanding that we'll think about space at least briefly to train ourselves not to forget that piece of the puzzle.

What's an Algorithm?

There's actually not a generally accepted definition of an algorithm, but let's use the following definition:

> "An algorithm is a detailed, step-by-step plan for solving a problem."

For example, say we're looking for the Queen of Hearts in a deck of cards. The following is one algorithm we could use to do that:

```
While there are more cards in the deck
    If the top card is the Queen of Hearts, say so and stop
    Otherwise, discard the top card
```

The algorithm above is certainly detailed enough for us to "execute" it (manually or some other way) and it contains all the steps we need to solve the problem. Although we could try to write the algorithm in a programming language, the essence of the algorithm is captured in the *pseudocode* above. Similarly, there are more formal

notations we could use to specify the algorithm, but we won't bother using them in this class.

So, given an algorithm, how do we analyze the algorithm to determine its complexity? Read on!

Worst Case

When people perform algorithm analysis, it's almost always to quantify the worst case complexity of the algorithm. If the Queen of Hearts is on the very top of the deck, for example, we can find it very easily, and it hardly takes any time at all. What if the Queen isn't at the top of the deck, though?

The worst possible place for the Queen is of course at the bottom of the deck. In that case – the worst case – we'll have to look at every single card in the deck before we find the Queen. Now, you might think that you can "fix" this problem by just starting your search at the bottom of the deck instead – then you find the Queen right away. Of course, this algorithm has a different worst case from the previous one. For this algorithm, the worst case is that the Queen is at the top of the deck, and its complexity is exactly the same as for our initial algorithm. We'll find out how to quantify that complexity in the next section.

There are a couple of important lessons here. First of all, the worst case for an algorithm is specific to that algorithm; there isn't a standardized worst case for the problem itself. Because the structure of the algorithm determines the worst case, different algorithmic approaches to a particular problem have different worst cases. Second, changing something simple (like the search order) typically doesn't change the complexity of a particular algorithm. We usually need to change something more substantial in the structure of the algorithm to change its complexity. We'll talk lots more about this when we discuss the various sorting algorithms.

But this all seems so pessimistic, doesn't it? Assuming the worst will always happen just seems so negative! Couldn't we at least try to figure out the average complexity instead? The short answer is that working with average complexities turns out to be pretty hard in practice, so almost everyone just uses the worst case. This is the most conservative thing to do because the algorithm's complexity can never be worse than the, ahem, worst case.

Big-O Notation

To quantify the essential complexity of an algorithm, we use something called Big-O notation (some people don't use the hyphen, but of course the ideas are the same). Basically, figuring out the Big-O complexity of an algorithm gives us an upper bound on how long the algorithm will take while also ignoring extraneous details that don't really affect the essential complexity of the algorithm. It's definitely time for an example.

The algorithm that looks for the Queen of Hearts starting at the top of the deck is repeated here for easy reference:

```
While there are more cards in the deck
    If the top card is the Queen of Hearts, say so and stop
    Otherwise, discard the top card
```

The first question to ask is how many times does the loop execute in the worst case? For a standard deck of cards, the answer is 52, but we're generally looking for a more general complexity analysis for the algorithm. If the deck of cards has n cards in it, how many times will the loop execute in the worst case? The answer is n times, because we might have to check every card in the deck for the Queen. So now we're in the loop. Checking if the top card is the Queen of Hearts (think of this as evaluating the Boolean expression for an if statement if that helps) will always take the same amount of time; we call this a *constant time* operation. Let's say that takes 1 unit of time (we'll find in a moment that the number we pick doesn't matter). Similarly, the "say so and stop" and "discard the top card" operations are also constant time (at least if we're smart about discarding), and we always do one or the other; let's say this step takes 2 units of time whichever branch we take. If we wanted to express the algorithm's time performance mathematically, we could now say:

$$f(n) = 3n$$

This is therefore a *linear* algorithm, because the upper bound on the algorithm's performance can be drawn as a line (rather than some other curve). But aren't all linear algorithms pretty much the same given that the upper bound is a line? Yes! They all belong to the same class of (linear) functions, so we really think of our f(n) above as the same (in terms of algorithmic complexity) as $g(n) = 42n - 17$ and $h(n) = 512n + 128$. How do we say that using Big-O notation? Here it comes … wait for it … we say that f(n) (and g(n) and h(n)) are O(n) algorithms. In case you were wondering, the capital O is shorthand for "order of."

So, to evaluate the complexity of an algorithm and express it in Big-O notation, we first need to analyze how much time each step in the algorithm will take in the worst case. We then take the resulting equation and throw away all the constants, retaining only the information related to the size of the data being processed (in our example, the n cards in the deck). That gives us our Big-O for the algorithm. We actually don't usually spend a lot of time trying to figure out each constant either. We really try to focus on the steps that depend on the size of the data being processed and pretty much ignore the rest. Don't worry, we'll be doing more examples as we move along.

One final comment before we go to the next section. What's the Big-O of the following function:

$$f(n) = n^2 + n$$

The answer is that it's $O(n^2)$. The squared term is said to *dominate* the linear term (get your mind out of the gutter), and as n grows large the squared term will account for the majority of the algorithm's processing. That means we can even throw away terms that are related to the size of the data being processed as long as there's a bigger term (also related to the size of the data being processed) that dominates. This NEVER works for multiplication, by the way – we can't take an `n log n` term and throw away the `log n` part – it only works for addition and subtraction.

Well, it's a good thing this is all pretty much just common sense, huh? Let's look at a few of the more common classes of algorithm.

Constant Algorithms

Constant algorithms are expressed in Big-O notation as O(1) algorithms. They're algorithms that execute in an amount of time unrelated to the size of the data being processed.

Here's one example of an O(1) algorithm; this algorithm simply determines if a given number is even or odd:

```
If the number is evenly divisible by 2, answer even
Otherwise, answer odd
```

Note that the algorithm still takes some time to execute (the computer has to do the division by 2), but the amount of time this takes is essentially constant. We don't know what that constant is, but we don't care either; because the processing time isn't based on the size of data being processed, we just say it's O(1).

Here's another example of an O(1) algorithm:

```
Given n, return the nth element of the weapons array
```

First, notice that we're finally talking about weapons! More importantly, though, don't be fooled by the fact that I included an n in the algorithm description. Just because n is used as the array index doesn't mean this is an O(n) algorithm. If we know which element of the array we want, we just grab it from memory in constant time. That makes this an O(1) algorithm.

Logarithmic Algorithms

Logarithmic algorithms are expressed in Big-O notation as O(log n) algorithms. They're algorithms that execute in an amount of time with an upper bound of a function dominated by the logarithm of the size of the data being processed. The logarithm here is base 2, not base 10 or the natural logarithm.

Let's look at an algorithm that searches a sorted array for a particular value. We'll use something called *binary search* as our search algorithm. Let's talk about the general idea first, then show the actual algorithm and analyze its complexity.

The big idea for binary search is that we keep breaking the data we need to search in half until we find the value we're looking for or discover that it's not there at all. The data set has to be sorted already for the algorithm to work. Let's look for the value 3 in the following set of data:

1 3 5 12 17 21 33 42 42 42 42

We start by checking to see if the middle value of the data set is a 3. In this case, the middle value is 21, so we're not done looking yet. Because the data set is sorted, we know that if 3 is in the data set, it's in the left half of the data. We then check the middle of that half of the data set. That value is 5, so we're still not done.

We then split the data set to the left of the 5 in half and find the 1. Now, those of you paying close attention observed that there was no "middle" value to the left of the 5; given that we had 2 values, we needed to pick which one to pick as the middle. Which one you pick (whether you round up or down) doesn't matter for the complexity of the algorithm, but it is a detail that matters when you actually implement the algorithm! In any case, we now have to look at the middle of the data set to the right of the 1 and to the left of the 5. Since that data set only has one piece of data, the middle of that data set is the 3, so we've found the value we were looking for and we're done.

Now let's express this as a detailed algorithm; notice that there's some extra book-keeping we need to include to make the algorithm detailed enough:

```
Set range lower bound to 0
Set range upper bound to n – 1
Set found value flag to false
While found value flag is false
    Set middle value location to range lower bound +
        (range upper bound – range lower bound) / 2
    Set middle value to the array element at middle value location
    If middle value is equal to search value, set found value flag to true
    Otherwise
        If middle value > search value
            Set range upper bound to middle value location - 1
        Otherwise
            Set range lower bound to middle value location + 1
        If range lower bound > range upper bound, break out of loop
Return found value flag
```

The algorithm simply returns true if we found the value and false if we didn't, but we could just as easily have returned the location of the value in the array or -1 if it's not in the array.

So how the heck do we analyze this algorithm to determine its complexity? The trick is to recognize that everything before the loop takes constant time and everything inside the loop takes constant time on a particular iteration of the loop. That means that all we really need to figure out is how many times we loop in the worst case. This is a recurring pattern in algorithm analysis, by the way. We regularly find ourselves looking at numbers of loop iterations (or recursions, which we'll see later) to determine algorithm complexity.

Let's figure out the maximum number of loop iterations by looking at how the size of the data set changes as we execute the algorithm. The first time we hit the loop, we have n pieces of data to search. After the first pass through the loop, we have n/2 pieces of data to search because we split the data set in half (it could actually be n/2 -1, but we don't care about that constant either). After the second pass through the loop, we have n/4 pieces of data to search. You should definitely be seeing a trend here! Because we split the data set in half each time, the maximum number of times we'll loop before finding the value or discovering it's not there at all is log n (or log n + 1 in some cases). That means this is an O(log n) algorithm.

Linear Algorithms

Linear algorithms are expressed in Big-O notation as O(n) algorithms. They're algorithms that execute in an amount of time with an upper bound of a function dominated by the size of the data being processed. As discussed above, our algorithm to look for the Queen of Hearts is a linear algorithm.

Loglinear Algorithms

Loglinear algorithms are expressed in Big-O notation as O(n log n) algorithms. The upper bound for these algorithms is a function of the size of the data being processed multiplied by the log of the size of the data being processed. We'll get to see an algorithm in this class when we talk about sorting using an algorithm called heapsort.

Quadratic Algorithms

Quadratic algorithms are expressed in Big-O notation as $O(n^2)$ algorithms. Let's take a look at one example of an algorithm in this class.

Say we wanted to sort a list of numbers. There are lots of ways we could do this – and we'll talk about a number of them throughout the course – but pretty much the simplest (and most inefficient) algorithm to use is called *bubble sort*. Let's work through an example of how it works; we'll get more formal with the algorithm itself in the sorting reading. Let's sort the following set of data:

7 6 5 3 2 4

We'll start by making one pass through the set of data, comparing two elements and swapping them if the element on the left is larger than the element on the right. For the first comparison, 7 is larger than 6, so we swap them, making the set of data

6 7 5 3 2 4

For the second comparison, 7 is larger than 5, so we swap them, making the set of data

6 5 7 3 2 4

When we're done with all the comparisons on the first pass through the data, we end up with

6 5 3 2 4 7

The result of the first pass is that the largest element in the set of data has "bubbled" its way to the end of the list. Make sure you understand the table below.

Pass Number	Data After Pass					
2	5	3	2	4	6	7
3	3	2	4	5	6	7
4	2	3	4	5	6	7

So, this doesn't seem so bad; it only took us 4 passes to sort the set of 6 pieces of data. Your intuition might tell you that this is an O(n) algorithm, but your intuition would be wrong! Before moving on to the next paragraph, try to think of what the worst case would be for this algorithm. Got it? Okay, read on.

The worst case is that the set of data is in precisely reverse order. In that case, it's going to take us n – 1 passes through the data. If that still feels like an O(n) algorithm, remember that at this point we're just counting passes, we're not counting the number of comparisons we need to make within each pass. Think of it this way. If we implemented bubble sort using nested loops (which is how it's normally done), at this point we've only looked at how many times the outer loop would execute in the worst case. What about the inner loop?

We can figure this out by looking at the number of iterations through the inner loop on the first pass through the algorithm. For n data elements, the inner loop executes n – 1 times; putting one loop inside the other means we multiply the numbers of max loop iterations to get our overall complexity, leading to $O(n^2)$ in the worst case.

But wait, the Mathletes are saying, this isn't fair. While it's true that a function in n^2 gives us an upper bound, it doesn't feel like a very tight upper bound, does it? Everyone should agree that we have n – 1 passes for the outer loop, but this n – 1 passes on the inner loop is way too harsh. After all, by the last pass we only iterate on the inner loop once, not n – 1 times!

If we're more precise about the actual number of comparisons that have to occur over the course of the entire algorithm, we actually get the following:

$$\sum_{k=1}^{n-1} k$$

Hah! See what that extra math rigor buys you? When we solve this summation, we get n(n-1)/2, which is (n^2 – n)/2, which is (oh damn) still O(n^2). The moral of this story is that we certainly need to be careful when we're analyzing algorithm complexity, but we don't have to be perfectly precise. We are looking for upper bounds, after all.

One thing that could have crossed your mind as we worked through the bubble sort algorithm is that there must be a better way to do this. There are many, many better ways to sort (bubble sort is pretty much the worst), but they all share something in common. In a computer program, we can only compare two elements at a time then decide what to do with them. When you looked at our starting set of data, you probably just immediately found the largest number and put it on the end, then found the next number and put it to the space to the left of the end, and so on. Even though it felt like you were sorting "all at once", you actually iterated over the entire list (n) finding the largest number, then iterated over all but the end of the list (n – 1) finding the next largest number, and so on. This is, in fact, an O(n^2) approach to the problem.

Exponential Algorithms

Exponential algorithms are expressed in Big-O notation as O(c^n) algorithms. They're very, very inefficient, and we'll avoid them like the plague. There are some kinds of problems for which the best known solutions are exponential, but we're not going to take on any of those problems in this class.