CSCI 3104, CU-Boulder **Prof. Aaron Clauset**
Lecture 2 **15 January 2014**

1 Big-O and the asymptotic analysis of algorithms

In algorithms, we often use *asymptotic analysis* to compare the way different algorithms use resources, either time or space. The "asymptotic" in asymptotic analysis means that we are primarily interested in the functional form of the resource utilization as the size of the input n goes to infinity.

An example, and Θ. For instance, how much space does an array consume? In low-level languages like `C`, an array is literally a contiguous block of memory and thus takes an exact amount of space. If an element in the array takes up 32 bits of space, then an array of n elements takes $32n$ bits or $4n$ bytes. If an element takes 64 bits or 128 bits, then an n element array takes $64n$ or $128n$ bits of space.

Notice, however, that each of these has a common property: they are each *linear* in the size of the array. That general property would be true no matter how many bits each element took, so long as the "size" of an element did not vary with n. This is precisely what we mean by saying that, in the case, the amount of space required by an array is $\Theta(n)$.

Technically speaking, $\Theta(n)$ means something very specific. When we say that the amount of resources consumed (time or space) is some function $f(n)$ and that $f(n) = \Theta(g(n))$, we mean that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \exists c_1 > 0, \exists c_2 > 0, \forall n \geq n_0 > 0 \ . \tag{1}$$

In words, we mean that there exists some choice of input size n_0 such that for all larger inputs, there are choices of the constants c_1, c_2 so that the function $g(n)$ is both an upper *and* a lower bound on $f(n)$. When the upper- and lower-bound functions have the same form, we call the bound *tight*, which is what Θ means in algorithms. If you are asked to justify a claim that some function is $\Theta(g(n))$, then that usually means identifying the constants c_1, c_2, n_0 that makes the statement true.

This approach to talking about functions is useful in algorithms because it lets us focus on the dominant or leading-term behavior. It allows us to quickly compare different algorithms and make judgements about which will, in the long run, be more efficient, i.e., use fewer resources. That is, $\Theta(n)$ is better than $\Theta(n^2)$, which is better than $\Theta(c^n)$ for $c > 1$, etc.

The most common bound: Big-O. Tight bounds are the goal of all algorithm analysis: they tell us exactly how fast a function grows. But, tight bounds are often hard to achieve and we must settle instead for a weaker statement. The most common of these is $O(g(n))$ which relaxes the definition of Θ by providing only the upper bound. That is, $f(n) = O(g(n))$ means

$$f(n) \leq c_2 g(n) \qquad \exists c_2 > 0, \forall n \geq n_0 > 0 \ . \tag{2}$$

In words, O (or sometimes $\mathcal{O}$) says that our function $f(n)$ grows no faster than $g(n)$, and thus $g(n)$ provides an upper bound on the resource use. This is by far the most common type of statement we will make about an algorithm.[1]

A warning, and two examples of sloppy thinking. Asymptotic analysis is a powerful way to make precise statements about the way different functions grow with n, but with great power comes great responsibility. A claim that some function is $\Theta(n)$ may be obvious, but you should be prepared to back up that claim by providing (or knowing that you can provide with a moment of work) the corresponding constants.

It is easy to think of asymptotic notation, i.e., O, Θ, Ω, o and ω, as heuristics. This can be dangerous. For instance, what is wrong with the following statement?

$$f(n) \text{ is at least } O(g(n))$$

The phrase "at least" is a lower-bound-type statement, while O is an upper-bound-type statement, so the statement overall is a contradiction: no function can grow at least as fast as a function that it grows no faster than.

Here is another problematic statement:

$$2n = O(2^n)$$

Technically, this statement is correct because all exponential functions grow faster than all linear functions. (Can you prove this?) Here, we can derive the specific values of c_2 and n_0 that makes this statement true:

$$c_2 = 1 \qquad n_0 > 1 \ .$$

To verify this, we substitute these values into the original form: $2 \cdot 1 \leq 1 \cdot 2^1$, which is true. But, this statement is not very helpful because an exponential function grows much much more quickly than a linear function. In fact

$$n = O(n^2) = O(n^3) = O(1.1^n) = O(1.2^n) = O(2^n) \ , \tag{3}$$

which illustrates that there is a lot of room (in fact, infinitely much) between $O(n)$ and $O(2^n)$. So, while technically correct, this bound is so loose as to be nearly trivial. In algorithms, we always strive to produce the tightest bound possible. Ideally, we provide tight upper and lower bounds. Lacking that, we provide one or the other (which one we provided depends on what we are trying to show; usually, an upper bound, but not always).

[1]The other definitions are also various relaxations of the definition of Θ. Ω provides a possibly tight lower bound; ω provides a strict lower bound, i.e., $>$ instead of $\geq$; o provides a strict upper bound, i.e., $<$ instead of $\leq$.

Useful things. We will use asymptotic analysis throughout the semester. The final statements we are aiming for are compact Big-O statements. Usually, we begin with more complex functional forms, which we need to transform into simpler ones, or, we are given a pair of functions and we need to determine which is bigger. Thus, there are a number of useful mathematical identities that should be familiar to you. Here are several:

1. L'Hopital's rule:
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{\partial}{\partial n} f(n)}{\frac{\partial}{\partial n} g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$
2. $(x^y)^z = x^{yz}$
3. $x^y x^z = x^{y+z}$
4. $\log_x y = z \implies x^z = y$
5. $x^{\log_x y} = y$ by definition
6. $\log(xy) = \log x + \log y$
7. $\log(x^c) = c \log x$
8. $\log_c(x) = \log x / \log c$

Atomic operations and analyzing code. In the analysis of algorithms, we are concerned only with counting *atomic operations*. As the name suggest, these are very small, "indivisible" computational steps. Knowing what counts as atomic requires a model of computation. That is, we need to define what we mean by computing something. In this class, we will use the `RAM` model of computation, which is basically the abstraction that most modern computers implement. Atomic operations are all the basic mathematical operators, including addition, subtraction, multiplication and division, along with comparison operations involving built-in variables (integers, reals, and characters), and storing a value in such a variable.

When we analyze an algorithm, we are counting the atomic operations. Crucially, however, we don't need to count *all* the operations because we only care about the asymptotic behavior of the algorithm, which is dominated by the fastest-growing term in the function. An example will illustrate this more clearly.

Here, we choose a non-recursive example. Many algorithms are represented using recursive functions, in part because recursion is a powerful abstraction technique that allows complex operations to be expressed compactly. We will revisit asymptotic analysis for recursive functions later.

Most code you will encounter outside of academia, however, is written in terms of loops. Sometimes, these can be very easy to analyze. For instance, here is a simple accumulator function, which takes as input a positive integer n and outputs the sum of the first n integers.[2]

```
computeSum(int n) {
    if n <= 1 { return 0 }  // catch negative values!
    acc = 0                 // initialize the sum
    for i = 1; i<=n; i++ {  // add things up
        acc += i
    }
    return acc              // return the sum
}
```

How many atomic operations does this function compute? To illustrate the utility of asymptotic analysis, we will count them all and then make an observation.

First, the function is passing n by value, which is a single copy (count= 1). It then makes a comparison (count= 2) followed by an assignment (count= 3). Now, we enter a loop. Setting up the loop takes an assignment followed immediately by a comparison, to check the loop termination criterion (count= 5). Now it gets more interesting.

Each pass around the loop incurs exactly the same costs: an increment of the loop counter, a comparison to check for termination, an addition and an assignment, which increases our count by 4. And, we pass exactly n times around this loop, before making a final copy operation. Thus, the total, exact count of atomic operations is $5 + 4n + 1$, where the first term counts the startup costs, the second counts the loop's costs and the third counts the exit costs.

Thus, asymptotically, this function takes $\Theta(n)$ time to compute the sum. The constants in the exact count would change if we added more operations to the startup (e.g., reading the complete works of Shakespeare),[3] or if we added more operations to each pass through the loop (e.g., solving every possible 9x9 Sudoku puzzle).[4] Sometimes, these constants can be very large, but asymptotically they are all equivalently unimportant.

Suggestions for analyzing code. As you become more comfortable thinking about time or space and looking at code, many algorithms will immediately reveal their running times or space requirements to you, e.g., you might observe that an algorithm is a pair of nested `for` loops each

[2]This function is quite dumb, and we know a simple mathematical formula for computing its result in $O(1)$ time. But, in your lives writing code, you will encounter many people who are less clever than you, and they will write slower versions of something you know how to do faster. But sometimes, you will be that person.

[3]The complete works of Shakespeare have a fixed size, and so reading them takes a fixed amount of time. Debating their meaning, however, may take forever.

[4]Also a task that with fixed size, albeit a very large one, because the size of the puzzle is fixed.

of length n, and the interior of these loops cost some constant amount of work. Thus, you would conclude that the algorithm takes $\Theta(n^2)$ time. Further, you might observe that these loops only ever allocate a constant number of variables, and thus the algorithm takes $\Theta(1)$ space.

Other algorithms, however, will be non-trivial to understand: their behavior at a particular step will depend on the sequence of decisions made before. The key idea for analyzing these algorithms, which include most of the algorithms in this class, is to identify the abstract pattern of the algorithm's performance. Identifying a property of the algorithm or data structure that is constant as it progresses is the common element in all algorithm problems.

2 On your own

1. Read Chapters 1, 2 and 3 in CLRS.