Lecture Notes on "Big-O" Notation

15-122: Principles of Imperative Computation
Jamie Morgenstern

Lecture 7
May 28, 2012

1 Introduction

Informally, we stated that linear search was, in fact, a linear-time function, that binary search "was logarithmic" (in running time), and counted out the number of operations which mergesort executed, in a somewhat sloppy fashion, and decided our implementation executed $kn\log(n) + dn + e$ for $k, d, e \in \mathbb{N}$ constant, independent of our input size, n. In this lecture, we will make formal these notions, and try to give some examples and motivation for a framework for analysis of algorithms and programs known as *asymptotic complexity*.

2 Motivation

Certain operations, such as addition, subtraction, or multiplication of two `int`s, or reading an element of an array, are said to be *constant time operations*, in that they take some fixed unit of time for your processor to calculate the result. Others, such as executing a `for`-loop or allocating a array, instead take a processor time proportional to the bound on the for-loop, or the size of the array which must be allocated. Different machines are faster at some operations than others. For this reason, asymptotic analysis does away with exact calculation of how many additions versus how many multiplications, and the relative time each takes, and instead say each is one operation which takes one unit of time. Once we've done this, and swept

under the rug exactly how many timesteps each of these different operations take, it seems somewhat foolish to care very much about the difference between 2 and 3 operations, or 100 or 200 operations. This is why asymptotic analysis works modulo multiplicative constants: n operations versus $3n$ operations versus $100n$ are all considered equivalent.

Moreover, when we were counting operations made by functions we've written, we didn't pay particularly close attention to the number of operations the base case made, or whether we executed the loop n or $n-1$ or $n+1$ times. If there is some small input, *all* reasonable programs should run quickly on them, and the overhead of printing or calling the function often overwhelms the run time of the few operations occuring for small inputs. Given some runtime of your program on a large input, the asymptotic complexity of your code should give you a rough estimate of how much longer the function would take on an even larger input. For this reason, we only consider *sufficiently large* inputs when talking about asymptotic complexity.

3 Formal Definition

Given $f, g : \mathbb{N} \to \mathbb{R}^+$, we say that $f \in O(g)$ if there exists some constants $c > 0, n_0 \geq 0$ such that for every $n \geq n_0$, $f(n) \leq cg(n)$. That is, for sufficiently large n, the *rate of growth* of f is bounded by g, up to a constant c. f, g might represent arbitrary functions, or the running time or space complexity of a program or algorithm.

4 Mergesort $\in O(n \log(n))$

In class, we said that mergesort is $O(n \log(n))$. How do we formally argue this? We drew the tree of calls, and argued that the depth of the tree was roughly bounded by $\log_2(n)$. Since all we care about is an upper bound, we can say we'll only consider inputs of size 2^i for some i, and pick the smallest i for which $2^i \geq n$. Then, the depth of the call tree is exactly $\log_2(n)$, and each path from the root is exactly that long.

We then counted the amount of work which happened at each level: splitting the array in two took some constant t operations, and merging the results of the two recursive call took $dk + e$ operations for some constants d, t, where k is the size of the input at that level. At each level i, the input size was roughly $n/2^i$ for each call, and there were roughly 2^i calls at the ith level. Once we've assumed the input size is a power of two, then we can say the input size and number of calls are exactly $n/2^i$ and 2^i, respectively,

so there is $(d(n/2^i)+t)*2^i = dn+t2^i$ work done at each level, or $\sum_{i=0}^{\log(n)} dn+t2^i = dn\log(n)+tn$. Now, we must show that $dn\log(n)+tn \in O(n\log(n))$. In order to show this, we must find some c, n_0, and show that for any $n \geq n_0$,

$$dn\log(n)+tn \leq cn\log(n)$$

.

Let's set $c = d+t, n_0 = 2$. Notice then that what me must show for all $n \geq 2$ has reduced to

$$dn\log(n)+tn \leq (d+t)n\log(n)$$

This is true iff

$$tn \leq tn\log(n)$$

But, for $n \geq 2, \log(n) \geq 1$, so $n\log(n) \geq n$ and $tn\log(n) \geq tn$, as desired.

5 Other Examples

This section presents two other examples of using big "O" notation, and their proofs. First, we show a piece of code, which we will analyze for time complexity

```
for (int i = 0; i < n; i++) {
  for (int j = i; j < n; j++) {
    binsearch(A, i, j);
  }
}
```

Recall that we said binary search was $O(\log(n))$; we will use this as a lemma and claim the above piece of code runs in time $O(n^2\log(n))$. Why is this? First, for a particular i, j, the inner loop does $O(\log(j-i))$ work. Since $j \leq n, i \geq 0$, this is upper bounded by $O(\log(n))$. Then, binary search is called a total of

$$\sum_{i=0}^{n}\sum_{j=i}^{n} 1 = \sum_{i=0}^{n}(n-i)$$

$$= n^2 - \sum_{i=0}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2-n}{2}$$

times. Thus, the total number of operations is bounded, for some n_1, c_1 determined by the O of binary search, for all $n \geq n_1$,

$$\text{Number of Operations of loop} \leq c_1 \log(n)(\frac{n^2 - n}{2}$$

Now it remains to show that there is some $c^* \geq c_1, n^* \geq n_1$ (both inequalities are necessary for the above claim to hold), such that for all $n \geq n^*$,

$$c_1 \log(n)\frac{n^2 - n}{2} \leq c^* n^2 \log(n)$$

Let's pick $c^* = c_1$, $n^* = n_1$. Then, we must show, for all $n \geq n_1$,

$$c_1 \log(n)\frac{n^2 - n}{2} \leq c_1 n^2 \log(n)$$

Observe that, since $n \geq 0$, we have

$$c_1 \log(n)\frac{n^2 - n}{2} \leq c_1 \log(n)\frac{n^2}{2}$$

and also that

$$c_1 \log(n)\frac{n^2}{2} \leq c_1 \log(n) n^2$$

But these two inequalities give us exactly what we want, namely that

$$c_1 \log(n)\frac{n^2 - n}{2} \leq c_1 n^2 \log(n)$$

.

6 Theorems you can use without proof

Theorem 1 Sum of functions *If $f \in O(g)$, $f' \in O(g')$, then*

$$f + f' \in O(g + g')$$

As a corrolary, if $g = g'$, then

$$f + f' \in O(g)$$

Theorem 2 Product of functions *If $f \in O(g)$, $f' \in O(g')$, then*

$$f \cdot f' \in O(g \cdot g')$$

Theorem 3 Power of functions *If $f \in O(g)$, then for all $\alpha > 0$,*

$$f^\alpha \in O(g^\alpha)$$

Theorem 4 Composition of functions *If $f \in O(g)$ and $f' \in O(g')$, then*

$$f \circ f' \in O(g \circ g')$$

Theorem 5 Constants *Constants, c, don't grow at all. That is, for any increasing function f of $\mathbb{N}$,*

$$c \in O(f(n))$$

Theorem 6 Polynomials versus logs *For any $\alpha > 0$, a polynomial of that degree grows asymptotically faster than a logarithm. I.e.,*

$$\log(n) \in O(n^\alpha)$$

Theorem 7 Polynomials of lower degree *Higher-degree polynomials grow more quickly. I.e., for any $\alpha, \epsilon > 0$,*

$$n^\alpha \in O(n^{\alpha+\epsilon})$$