CS/Math 240: Introduction to Discrete Mathematics 3/8/2011

Lecture 14 : Asymptotic Analysis

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

DRAFT

Last two lectures, we analyzed the running times of various algorithms we've seen throughout the course. As we have found out, getting exact results was easy for some algorithms, but took a considerable amount of effort for some other algorithms. It took us one full lecture to analyze the number of recursive calls made by one particular implementation of Euclid's algorithm, and even then we did not obtain an exact figure on the number of recursive calls. Moreover, these algorithms were all fairly simple, so you can imagine how hard it would be to do an exact analysis of a more complicated algorithm. Today we introduce an analysis technique that allows for simpler arguments at the cost of a negligible loss in precision.

14.1 Motivation for Asymptotic Analysis

We now review some of the results from last two lectures and complete the analysis of Euclid's algorithm. We first show the results that were easy to obtain, and give the harder-to-get results later. In fact, the last result won't even be exact, but will be "good enough".

In Section 14.2 we present a technique for obtaining "good enough" results, asymptotic analysis. We also specify what "good enough" means in the context of program analysis, and explain why "good enough" results are sufficient.

14.1.1 Earlier Results

14.1.1.1 Towers of Hanoi

For the towers of Hanoi problem, we defined $M(n)$ to be the number of moves needed for the instance of the problem with n disks. We found the recurrence $M(n) = 2M(n-1) + 1$, with $M(0) = 0$. We solved this recurrence to get $M(n) = 2^n - 1$.

14.1.1.2 Binary Multiplication

Next, we analyzed the binary multiplication algorithm. There we defined $S(a, b)$ to be the number of bit shifts on input a, b. We had the recurrence $S(a, b) = S(a, \lfloor b/2 \rfloor)$ with $S(0) = 0$. The solution to this recurrence was $S(a, b) = \lfloor \log_2 b \rfloor + 1$.

A note on logarithms: *Let us agree from now on that when we say* log*, we mean the logarithm with base* 2*, and drop the subscript.* With this convention, the expression for $S(a, b)$ is $\lfloor \log b \rfloor + 1$.

14.1.1.3 Simple GCD Algorithm

We also analyzed a simple algorithm for finding greatest common divisors. Our measure of complexity there was $R(a, b)$, the number of recursive calls made on input a, b. However, we did not compute $R(a, b)$. This is actually hard to do.

Instead, we focused on the largest number of recursive calls made on input (a, b) with $a, b \leq n$. We called this quantity $R'(n)$. We found the recurrence $R'(n) = R'(n-1) + 1$ with $R'(1) = 0$, and found the solution $R'(n) = n - 1$.

Keep in mind that $R'(n)$ is only an upper bound and not an exact expression for $R(a, b)$ with $a, b \leq n$. There are some inputs a, b with $a, b \leq n$ and $R(a, b) = 0$.

Even tor this simple algorithm, we gave up on finding the exact complexity, and only found an upper bound in terms of the largest of the two inputs. The situation became even more complicated in the analysis of another algorithm for finding greatest common divisors.

14.1.2 Completing the Analysis of Euclid's Algorithm

We restate the recursive version of Euclid's algorithm below for the sake of completeness.

Algorithm Euclid(a, b)

Input: $a, b \in \mathbb{N}$, $a \leq b$, $b \neq 0$
Output: $\gcd(a, b)$

(1) **if** $a = 0$ **then return** b
(2) $r \leftarrow b - \lfloor b/a \rfloor \cdot a$
(3) **return** `Euclid`(r, a)

We analyzed this algorithm last lecture, and went even more off track than in the analysis of the simple algorithm for computing greatest common divisors. We defined the quantity $N(k)$ as the smallest value of n such that there is an input a, b with $a, b \leq n$ on which `Euclid` makes k recursive calls. We found the recurrence $N(k) = N(k-1) + N(k-2)$ for $k \geq 4$, and deduced that $N(k) = F_{k+2}$, the $(k+2)$nd Fibonacci number, for $k \geq 2$. This was a nontrivial result and took us almost an entire lecture to derive. We then remarked that the number of recursive calls was "roughly" $\log(b)$. We now give an argument for this.

We know that if (a, b) is a valid input such that $a, b \leq N(k)$, `Euclid`(a, b) makes at most k recursive calls. So, on input (a, b), `Euclid` makes at most k recursive calls, where k is the smallest integer for which $a, b \leq N(k)$. Now we would like to figure out what k is given (a, b). Since $a \leq b$, we can simplify this question, and, instead, ask what is the smallest k such that $b \leq N(k)$.

Recall that

$$F_n = \frac{\varrho^n - (1-\varrho)^n}{\sqrt{5}}, \quad \text{where } \varrho = \frac{1+\sqrt{5}}{2}.$$

We said $N(k) = F_{k+2}$, so

$$b \leq \frac{\varrho^{k+2} - (1-\varrho)^{k+2}}{\sqrt{5}}. \tag{14.1}$$

As a first step towards a bound on k, we rewrite (14.1) as $\sqrt{5}b \leq \varrho^{k+2} - (1-\varrho)^{k+2}$.

We claim $(1-\varrho)^{k+2} \leq \varrho^{k+2}$. To see this, recall $\varrho \approx 1.62$, so $1 - \varrho \approx -0.62$. Now raising ϱ to a high power gives a giant number, whereas raising $1 - \varrho$ to a high power gives a number that is close to zero. Thus, we further rewrite (14.1) as $\sqrt{5}b \leq 2\varrho^{k+2}$, and $\sqrt{5}b/2 \leq \varrho^{k+2}$. Now take logarithms of both sides to get

$$\log(\sqrt{5}/2) + \log b \leq (k+2)\log(\varrho) = k\log(\varrho) + 2\log(\varrho). \tag{14.2}$$

Note that $\varrho > \sqrt{5}/2$, so $2\log(\varrho) > \log(\sqrt{5}/2)$. Then (14.2) holds if $\log b \leq k \log(\varrho)$, and

$$k \geq \frac{\log b}{\log \varrho}. \tag{14.3}$$

This tells us that the smallest integer k for which $a, b \leq N(k)$ is greater than $c\log(b)$ where $c = 1/\log(\varrho)$. So pick k to be the smallest integer that satisfies (14.3). Hence, k is roughly $c\log(b)$, and $R'(b)$ is roughly $\log(b)$.

14.2 Asymptotic Analysis

We have finally obtained some bound on the number of recursive calls made by `Euclid`. One of the tricks we used to obtain the final result was to consider a looser inequality. For example, the bound $(1-\varrho)^k \leq \varrho^k$ is an immense overestimate, but it was sufficient for obtaining a good bound on the number of recursive calls. This is because there already was another term of ϱ^k in the expression in (14.1), so our overestimate of the *lower order term* $(1-\varrho)^n$ by the *dominant term* ϱ^n (or *higher order term*) only caused a loss of a factor of 2 in the analysis.

A loss of a factor of 2 is negligible compared to $\log(b)$ which appeared in our final bound on k. A factor of 2 is easy to overcome by giving the task to a couple different computers in parallel or to a faster computer that runs twice as fast, for example. On the other hand, it may be harder to find a computer that runs $\log(b)$ times as fast for a large b, or find $O(\log(b))$ computers (not to mention that there is some overhead involved in parallel computing). This is why a loss of a constant factor is "good enough", but any other loss is not because it may blur the difference between vastly different complexities. This is one of the ideas behind asymptotic analysis: we focus on first order terms, and ignore negligible terms as well as multiplication by constants because they are a secondary issue compared to the first order terms.

Another aspect of asymptotic analysis is that it is concerned only about large inputs. The differences between complexities of two algorithms may not be noticeable on small inputs, and any computer could run the two algorithms with different, but still small running times in a short period of time. But when the inputs get larger, the slower algorithm may not have a chance of finishing before the end of the universe, even on a supercomputer, whereas a slow computer could still execute the faster algorithm in a few days, or perhaps even seconds.

Observe the different complexities of the two algorithms for computing greatest common divisors. The number of recursive calls made by Euclid's algorithm grows much slower with the input size than the number of recursive calls made by the simple algorithm. The two complexities, as functions of the input size, are not different merely by a constant factor. Their first order terms are very different. One grows logarithmically whereas the other one grows linearly, so a good asymptotic analysis can show that Euclid's algorithm is faster than the simple algorithm for computing greatest common divisors.

14.2.1 Terminology and Notation

When doing asymptotic analysis, we are interested in the relative growth of two functions f, g from $\mathbb{N}$ to $\mathbb{R}_{>0}$, where by $\mathbb{R}_{>0}$ we mean the positive real numbers. We think of one or both of these functions as running times of programs. If f represents the running time of a program P, $f(n)$ tells us how many elementary operations P takes on input (of length) n.

Now let's forget about the fact that f and g represent running times of programs on various inputs, and set down some notation and terminology that is generally applicable. The next section uses what we develop now as a tool for analyzing programs.

We start with a very restrictive notion of equivalence.

Definition 14.1. *Given $f : \mathbb{N} \to \mathbb{R}_{>0}$ and $g : \mathbb{N} \to \mathbb{R}_{>0}$, we say $f(n)$ and $g(n)$ are* asymptotically equivalent *if*

$$(\forall \epsilon \in \mathbb{R}_{>0})(\exists N \in \mathbb{N})(\forall n \geq N)\ \ 1 - \epsilon \leq \frac{f(n)}{g(n)} \leq 1 + \epsilon. \tag{14.4}$$

We use the notation $f(n) \sim g(n)$ to denote that $f(n)$ and $g(n)$ are asymptotically equivalent.

In words, as n increases, the ratio of $f(n)$ and $g(n)$ approaches 1. Another way to formulate Definition 14.1 is $\lim_{n\to\infty} f(n)/g(n) = 1$.

Definition 14.1 captures one of the goals we have set for ourselves. It characterizes the behavior of two functions as their inputs get large.

Example 14.1: $F_n \sim \frac{\varrho^n}{\sqrt{5}}$.

Consider $f(n) = F_n$ and $g(n) = \frac{\varrho^n}{\sqrt{5}}$. Recall that $F_n = \frac{\varrho^n - (1-\varrho)^n}{\sqrt{5}}$ where $\varrho = (1 + \sqrt{5})/2$.

Now let's look at the ratio of f and g.

$$\frac{f(n)}{g(n)} = \frac{\frac{\varrho^n - (1-\varrho)^n}{\sqrt{5}}}{\frac{\varrho^n}{\sqrt{5}}} = \frac{\varrho^n - (1-\varrho)^n}{\varrho^n} = 1 - \frac{(1-\varrho)^n}{\varrho^n} = 1 - \left(\frac{1-\varrho}{\varrho}\right)^n. \tag{14.5}$$

Notice that $(1 - \varrho)/\varrho < 1$, so raising it to a high power yields a number that is close to zero. Thus, as n increases, the term $[(1 - \varrho)/\varrho]^n$ gets closer and closer to zero, which means (14.5) gets closer and closer to one.

To show that $f(n) \sim g(n)$, consider $\epsilon \in \mathbb{R}_{>0}$, and pick an N such that $[(1 - \varrho)/\varrho]^N < \epsilon$. Since 1 doesn't change with n, and $[(1 - \varrho)/\varrho]^n$ gets closer to zero as n increases, we get that $f(n)/g(n)$ is bounded between $1 - \epsilon$ and $1 + \epsilon$ for all $n \geq N$. Thus, $f(n) \sim g(n)$. ⊠

Example 14.2: Consider $f(n) = 12n^2 + 10^9 n + \frac{1}{n}$ and $g(n) = 12n^2$. Then $f(n) \sim g(n)$.

To see this, observe that as n increases, $12n^2$ grows much faster than the other terms in $f(n)$. Thus, when n is large, $10^9 n$ becomes negligible compared to $12n^2$ even though the coefficient in front of n may look like a giant number. A common expression for this is that $12n^2$ *dominates* $10^9 n$. Similarly, $12n^2$ dominates $\frac{1}{n}$.

Now we argue more formally. We can write

$$\frac{f(n)}{g(n)} = \frac{12n^2 + 10^9 n + \frac{1}{n}}{12n^2} = 1 + \frac{10^9}{12} \cdot \frac{1}{n} + \frac{1}{12} \cdot \frac{1}{n^3}.$$

Since both the terms $10^9/12n$ and $1/12n^3$ decrease towards zero as n increases, we can find an N for which Definition 14.1 is satisfied using the same argument as in the previous example.

Also notice that $f(n) \not\sim n^2$. The ratio of $f(n)$ and n^2 approaches 12 as n goes to infinity, and the ratio won't get close to one. ⊠

Example 14.3: $F_n \not\sim 12n^2$.

The Fibonacci sequence grows much faster than the quadratic $12n^2$. Let $f(n) = F_n$ and $g(n) = 12n^2$. For small values of n, we actually have $f(n) < g(n)$, but that quickly changes and $f(n)$ starts growing much faster than $g(n)$. One way to see this is that $f(n)$ increases by a factor of ϱ if n increases by 1, whereas $12n^2$ increases by a factor that is close to 1 (just write out $12(n+1)^2/12n^2$ to see this) if n increases by 1. ⊠

We see from the examples above that Definition 14.1 accomplishes some of our goals, but not all of them. It allows us to capture the behavior of functions as their inputs grow. However, as we saw in Example 14.2, there is still dependence on constant factors. In particular, we saw $f(n) \sim 12n^2$,

but $f(n) \not\sim n^2$. Since $f(n)$, $12n^2$, and n^2 grow like quadratics, we would like them to be similar, but asymptotic equivalence doesn't allow that. The next definition fixes this shortcoming of asymptotic equivalence.

Definition 14.2 (Big Theta)**.** *Given $f : \mathbb{N} \to \mathbb{R}_{>0}$ and $g : \mathbb{N} \to \mathbb{R}_{>0}$, we say $f(n)$* is Theta of $g(n)$ *if*

$$(\exists c, d \in \mathbb{R}_{>0})(\exists N \in \mathbb{N})(\forall n \geq N) \;\; c \leq \frac{f(n)}{g(n)} \leq d. \tag{14.6}$$

We write $f(n) = \Theta(g(n))$ to denote that $f(n)$ is Theta of $g(n)$.

Definition 14.2 no longer requires f and g to approach each other in the limit. Instead, it requires that their ratio be bounded above and below by some constants as n gets large. This allows us to ignore constant factors because the Θ notation "absorbs" them.

Let's see how the examples we used for asymptotic equivalence change. First, if $f(n) \sim g(n)$, then certainly $f(n) = \Theta(g(n))$. To see this, pick $\epsilon = 1/2$ (we can pick any $\epsilon > 0$ to make this argument work; $\epsilon = 1/2$ is just an example). Since $f(n) \sim g(n)$, there is an N such that for all $n \geq N$, $1/2 \leq f(n)/g(n) \leq 3/2$. So take that N, and pick $c = 1/2$ and $d = 3/2$ to show $f(n) = \Theta(g(n))$.

Example 14.4: By the previous paragraph, $F_n = \Theta(\varrho^n/\sqrt{5})$ and $12n^2 + 10^9 n + 1/n = \Theta(12n^2)$.

In addition, we also have $12n^2 + 10^9 n + 1/n = \Theta(n^2)$ because we can pick $N = 1$, $c = 12$ and $d = 10^9 + 13$ in order to satisfy Definition 14.2. Another option is $c = 12$, $d = 13$, $N = 10^9$. This second choice illustrates that we only care about larger values of n in asymptotic analysis. It also allowed us to pick c and d that are closer to each other.

Finally, $F_n \neq \Theta(12n^2)$ because F_n grows exponentially, and we cannot bound the ratio of F_n and $12n^2$ by any constant. ⊠

We may also be interested in only one of the bounds in (14.6). For example, if we want to guarantee that a program terminates within a number of steps that is at most some function of the input, we only care about the upper bound. On the other hand, if we want to prove to someone that the number of steps is at least some function of the input, we only care about the lower bound. Thus, we make the following two definitions.

Definition 14.3 (Big Oh)**.** *Given $f : \mathbb{N} \to \mathbb{R}_{>0}$ and $g : \mathbb{N} \to \mathbb{R}_{>0}$, we say $f(n)$* is Oh of $g(n)$ *if*

$$(\exists d \in \mathbb{R}_{>0})(\exists N \in \mathbb{N})(\forall n \geq N) \;\; \frac{f(n)}{g(n)} \leq d. \tag{14.7}$$

We write $f(n) = O(g(n))$ to denote that $f(n)$ is Oh of $g(n)$.

Definition 14.4 (Big Omega)**.** *Given $f : \mathbb{N} \to \mathbb{R}_{>0}$ and $g : \mathbb{N} \to \mathbb{R}_{>0}$, we say $f(n)$* is Omega of $g(n)$ *if*

$$(\exists c \in \mathbb{R}_{>0})(\exists N \in \mathbb{N})(\forall n \geq N) \;\; c \leq \frac{f(n)}{g(n)}. \tag{14.8}$$

We write $f(n) = \Omega(g(n))$ to denote that $f(n)$ is Omega of $g(n)$.

Observe that if $f = \Theta(g)$, then $f = O(g)$ and $f = \Omega(g)$. This follows because conditions (14.7) and (14.8) are weaker than condition (14.6). Also, if both $f = O(g)$ and $f = \Omega(g)$ hold, then $f = \Theta(g)$ because combining the inequalities from (14.7) and (14.8) gives us the inequality (14.6).

Example 14.5: By the previous paragraph, we have $F_n = O(\varrho^n/\sqrt{5})$ and $F_n = \Omega(\varrho^n/\sqrt{5})$. We also get $12n^2 + 10^9 n + 1/n = O(12n^2)$ and $12n^2 + 10^9 n + 1/n = \Omega(12n^2)$, as well as $12n^2 + 10^9 n + 1/n = O(n^2)$ and $12n^2 + 10^9 n + 1/n = \Omega(n^2)$.

Since F_n grows way faster than $12n^2$, $F_n \neq O(12n^2)$, but it is true that $F_n = \Omega(12n^2)$. Along the same vein, $12n^2 = O(F_n)$ and $12n^2 \neq \Omega(F_n)$. ⊠

Now we have enough notation, so let's see its use in program analysis.

14.2.2 Asymptotic Analysis of Euclid's Algorithm

The analysis of last lecture combined with Section 14.1.2 proved the following theorem.

Theorem 14.5. *The number of recursive calls made by* `Euclid` *on input* (a, b) *is* $O(\log b)$.

It took us an entire lecture to prove this theorem. We now show a simpler argument that proves Theorem 14.5. This should serve as an example of the fact that asymptotic analysis is much easier to do than exact analysis.

We need one small result before we give a shorter proof of Theorem 14.5.

Claim 14.6. *After line 2 of* `Euclid` *executes,* $r \leq b/2$.

Proof. We argue by cases.

Case 1: $a > b/2$. In this case $\lfloor b/a \rfloor = 1$, which gives us $r = b - \lfloor b/a \rfloor \cdot a = b - a$. Thus, we have $r = b - a$, and since $a > b/2$, $r < b/2$.

Case 2: $a \leq b/2$. Recall that dividing by a produces a remainder that is less than a. Since r is the remainder after dividing b by a, $r \leq a \leq b/2$. □

And now we are ready for a shorter proof of Theorem 14.5.

Proof of Theorem 14.5. Consider the first two recursive calls made by `Euclid` on input (a, b). The first argument in the first recursive call is $a' \leq b/2$, and the second argument is $b' = a$. The recursive call `Euclid(`a', b'`)` makes a second recursive call, with first argument $a'' \leq b'/2 \leq a/2$, and second argument $b'' = a' \leq b/2$. We see, therefore, that after two recursive calls, the first and the second arguments to `Euclid` are at most half their initial values.

Recursion ends as soon as the first argument goes below 1 because it's zero in that case. After $2k$ recursive calls, the first argument is at most $a/2^k$, which will be less than 1 if $a/2^k < 1$, or, in other words, if $k > \log(a)$. We will have $k > \log(a)$ if $k > \log(b)$ because $b \geq a$. And now we see that after $2\log(b)$ recursive calls, the first argument is less than one, so it's zero, and the chain of recursive calls ends. □