Asymptotic Growth (2)

Michael L. Littman

September 3rd, 1998

1 ANNOUNCEMENTS

Advanced students should seriously consider CPS 230 (Lars Arge).
Meets same time in our old room.
More email addresses??

2 RECURSION

2.1 Review

We saw two multiplication algorithms for $a \times b$.

- "Naive" implemented multiplication by successive addition. Running time appears linear in a.
- "Russian" implemented multiplication by a base-2 version of the standard grade school multiplication algorithm. Running time appears logarithmic in a.

Today we'll analyze the algorithm formally.

2.2 Recursive Russian

Recursion just means solving a problem by reducing it to a smaller instance of the same problem.
"Reducing" just means calling a subroutine. "Smaller problem" means different things, but the basic idea is that we are making measurable progress to the final answer.

```
int RussianRec(int a, int b)
//  implements Russian Peasant Algorithm recursively
{
  if (a == 0) return 0;
  else if ( 1 == a % 2)
    return(b+ RussianRec(a >> 1, b) << 1);
  else
    return(RussianRec(a >> 1, b) << 1);
}
```

2.3 Discussion

RussianRec has no explicit loop. Instead, we express the answer in terms of itself!
This is not circular because:

- There is a "termination" variable, a.
- This variable makes steady progress towards the termination condition.

2.4 Concrete Example

$$\begin{array}{rrcrcrl} & 9\cdot 3 & = & 3+2(4\cdot 3) & = & 27 & \\ \downarrow & 4\cdot 3 & = & 2(2\cdot 3) & = & 12 & \uparrow \\ \downarrow & 2\cdot 3 & = & 2(1\cdot 3) & = & 6 & \uparrow \\ \downarrow & 1\cdot 3 & = & 3+2(0\cdot 3) & = & 3 & \uparrow \\ \downarrow & 0\cdot 3 & = & 0 & \rightarrow & 0 & \end{array}$$

2.5 Exponentiation

I had planned to present the classic "exponentiation by repeated squaring algorithm" separately today. But then I realized that it's exactly the same as recursive Russian!
Homework: Write a recursive algorithm for computing a^b (where both a and b are positive integers). Analyze its correctness and efficiency. The number of recursive calls it makes should be logarithmic.

2.6 Divide and Conquer

When a is even...

$$\begin{aligned} ab &= \overbrace{b+b+b+b+b+\cdots+b+b}^{a} \\ &= \overbrace{b+b+\cdots+b}^{a/2}+\overbrace{b+b+\cdots+b}^{a/2} \\ &= 2\left(\overbrace{b+b+\cdots+b}^{a/2}\right) \end{aligned}$$

When a is odd...

$$\begin{aligned} ab &= \overbrace{b+b+b+b+b+b+\cdots+b+b}^{a} \\ &= b+\overbrace{b+b+\cdots+b}^{(a-1)/2}+\overbrace{b+b+\cdots+b}^{(a-1)/2} \\ &= b+2\left(\overbrace{b+b+\cdots+b}^{(a-1)/2}\right) \end{aligned}$$

We solve the problem by breaking it into roughly equal-size subproblems, solving them separately, then combining the results (in this case, there's just one subproblem).
Rule #2 of Good Algorithm Design: Divide-and-Conquer!

2.7 Analysis

Correctness [Step 2] follows from the derivation we gave.
We'll measure efficiency in terms of the number of recursive calls as a function of a. This is a good predictor of runtime as long as ab fits in a single integer variable. (Assumes the addition takes unit time.)

2.8 A Recurrence Relation

We want to know the number of iterations (now recursive calls) it takes to multiply a and b. We can figure this out fairly directly from the recursive expression of the algorithm. Let $T(a)$ be the number of iterations of Russian when run on input a.
(Aside: why isn't b needed here?)
We can argue:

- If $a = 1$, $T(a) = 1$,
- If $a > 1$ and a is odd, $T(a) = 1 + T((a-1)/2)$,
- If $a > 1$ and a is even, $T(a) = 1 + T(a/2)$.

Now, it is a simple matter to prove by induction that $T(a) = 1 + \lfloor \log(a) \rfloor$.

2.9 Induction Proof

Theorem: For an integer $a > 1$, we can divide a by two (and round down) $1 + \lfloor \log(a) \rfloor$ times before we get down to zero.
Proof: For $a > 1$, let $T(a)$ be the number of times we can divide a by 2 (and round down) before we get down to zero.
We can express T recursively as: $T(1) = 1$, and for $a > 1$, $T(a) = 1 + T(\lfloor (a/2) \rfloor)$.
We want to show that $T(a) = 1 + \lfloor \log(a) \rfloor$.

- Base Case: $a = 1$.

$$\begin{aligned} T(a) &= 1 \\ &= 1 + \lfloor 0 \rfloor \\ &= 1 + \lfloor \log(1) \rfloor \\ &= 1 + \lfloor \log(a) \rfloor. \end{aligned}$$

  Correct.

- Next, note that any number greater than one, divided by 2 and rounded down gives a result strictly less than the original number and greater than or equal to one. (You could prove this by induction, too.)
- Inductive step: Assume that for all $a' < a$, the theorem holds.

$$\begin{aligned} T(a) &= T(\lfloor a/2 \rfloor) + 1 \\ &= 1 + \lfloor \log(a/2) \rfloor + 1 \\ &= 2 + \lfloor \log(a) - \log(2) \rfloor \\ &= 2 + \lfloor \log(a) - 1 \rfloor \\ &= 1 + \lfloor \log(a) \rfloor. \end{aligned}$$

  Correct!
- QED.

2.10 About Proofs

The previous proof by induction was relatively formal. I want you to use this form initially when I ask for proofs by induction in the first homework. This will help familiarize you with the important parts of such a proof.
However, very few (if any) proofs are completely formal.
A proof is something one human being uses to convince another (intelligent but skeptical) human being that something is true.
Nothing magic about this. The important rule is that, instead of convincing someone by appealing to sympathy or threats, a proof begins with axioms as the basic truths and uses logic as the mechanism for deriving one true statement from another.

2.11 Summary

We've now seen several different algorithms, analyzed their correctness and their efficiency exactly in terms of an iteration count. What progress!
Next, we'll generalize some of what we learned.

3 ASYMPTOTIC GROWTH

3.1 Comparing Algorithms

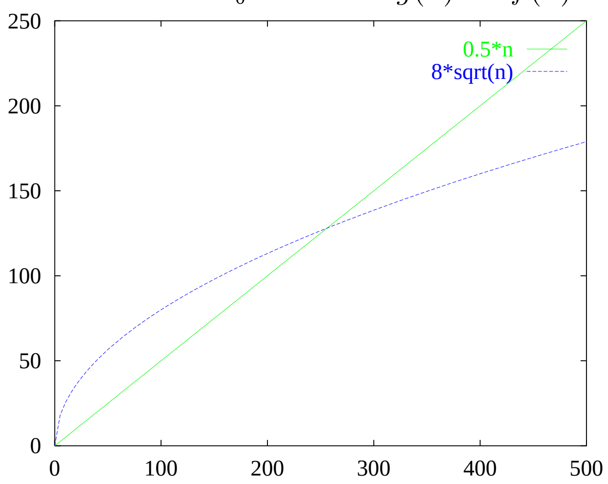
We want to compare algorithms to decide which we prefer.

1. But, there are many dimensions along which we might compare. So, we decide that what really matters is *running time*.

2. But, running time depends on input. So, we decide that what really matters is running time as a function of *input size*.
3. But, this can be hard to characterize. So, we decide that what really matters is *worst-case* running time as a function of input size.
4. But, one algorithm might be better on small inputs and the other on large inputs. So, we decide that what really matters is worst-case running time as a function of input size for *large inputs*.
5. But, this can still be quite hard to determine precisely. So, we decide that what really matters is worst-case running time as a function of input size for large inputs, *ignoring constant factors*.

3.2 Asymptotic Growth

This leads us directly (!) to a notion of asymptotic growth.
Function $g(n)$ is asymptotically bigger than function $f(n)$ if, for any scaling factor $c > 0$, there's some threshold n_0 such that $g(n) > cf(n)$ for all $n \geq n_0$.



3.3 Example

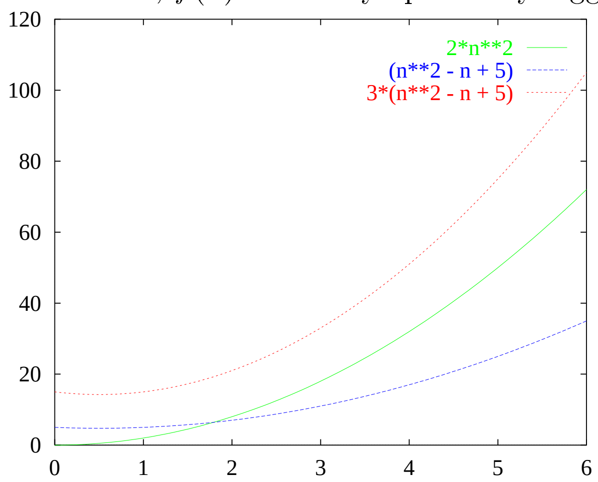
By this definition, $g(n) = 1/2n$ is asymptotically bigger than $f(n) = 8\sqrt{n}$. Why? Well, give me any scaling factor c.

$$
\begin{aligned}
g(n_0) &> cf(n_0) \\
1/2n_0 &> c(8\sqrt{n_0}) \\
n_0 &> 16c\sqrt{n_0} \\
n_0/\sqrt{n_0} &> 16c \\
\sqrt{n_0} &> 16c \\
n_0 &> 256c^2
\end{aligned}
$$

So, any n_0 at least that big will satisfy the conditions of the definition.

3.4 Counterexample

On the other hand, $g(n) = 2n^2$ is not asymptotically bigger than $f(n) = n^2 - n + 5$.
Take the constant $c = 3$. We have $g(n) < cf(n)$ for all n.
On the other hand, $f(n)$ is not asymptotically bigger than $g(n)$ either.



3.5 Big Theta

The notion of asymptotic growth can be used to partition the space of functions.
Two functions $f(n)$ and $g(n)$ are "equal" if neither is asymptotically bigger than the other.
We write this as $f(n) = \Theta(g(n))$.
Formally, $f(n) = \Theta(g(n))$ if and only if there exists scaling factors c_1 and c_2 and threshold n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
It's an $f(n)$ sandwich on $g(n)$ bread!

3.6 Examples

- $1/2n = \Theta(n)$
- $8\sqrt{n} = \Theta(\sqrt{n})$
- $n^2 - n + 5 = \Theta(n^2) = 2n^2$
- $F^{-1}(n) = \Theta(\log(n))$
- $7n^4 + 2n^3 - 4n \log n + \sqrt{n} = \Theta(n^4)$
- $\pi^2 = \Theta(1)$

3.7 Base of Logs

$\log_{10}(n) = \Theta(\log_2(n))$

$$
\begin{aligned}
f(n) &= \log_{10}(n) \\
10^{f(n)} &= n \\
\log_2(10^{f(n)}) &= \log_2(n) \\
f(n)\log_2(10) &= \log_2(n) \\
f(n) &= \log_2(n)/\log_2(10)
\end{aligned}
$$

So for $c_1 = (1/2)/\log_2(10)$, $c_2 = 2/\log_2(10)$, $n_0 > 0$,
$0 \le c_1 \log_2(n) \le f(n) \le c_2 \log_2(n)$ for all $n \ge n_0$.

3.8 Complete Set

Big Theta gives us a notion of asymptotic equality. We also have symbols for all of the other comparative notions as well.

$$
\begin{array}{lcl}
f(n) = o(g(n)) & \approx & f(n) < g(n) \\
f(n) = O(g(n)) & \approx & f(n) \le g(n) \\
f(n) = \Theta(g(n)) & \approx & f(n) = g(n) \\
f(n) = \Omega(g(n)) & \approx & f(n) \ge g(n) \\
f(n) = \omega(g(n)) & \approx & f(n) > g(n)
\end{array}
$$

3.9 Big O

Everyone has probably heard of big-O.
When we say the running time of a sorting algorithm is $O(n^2)$, we are giving an asymptotic upper bound on the running time for all inputs of size n.
This is often what we want. However, saying that the running time is $\Theta(n^2)$ is often a more precise way of saying the same thing.
Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$.

3.10 Some Technical Details

The "big" and "little" functions behave almost exactly like their standard real-number counterparts: transitivity, reflexivity, symmetry, and transpose symmetry.
However, one standard property does not hold: trichotomy. There are pairs of functions $f(n)$ and $g(n)$ such that none of $f(n) = o(g(n))$, $f(n) = \Theta(g(n))$, nor $f(n) = \omega(g(n))$ holds.
The book gives an example of $n^{1+\sin n}$ which can't be compared directly to n because it often acts like n^2.

3.11 Applying to Algorithms

Also, in using these functions to describe algorithms, there are a few things to note.

- If we say the algorithm's running time is $O(n^2)$, it means its worst-case running time is $O(n^2)$, but its best case might be better.
- If we say the algorithm's running time is $\Omega(n)$, it means its best-case running time is $\Omega(n)$, but its worst-case might be worse.
- Therefore, an algorithm with running time $\Theta(n \log n)$ takes $\Theta(n \log n)$ on all inputs of size n.

3.12 Math in Big Theta World

Here are some examples of reductions we can do when we're inside a big theta.

- $1 + 1 = 1$
- $10n + 6 = n$
- $2^{n+1} = 2^n$
- $5\sqrt{n} \log n + 21 \log^5 n = \sqrt{n} \log n$
- $2^{2n} = 4^n$

In general, we'd like to report functions in their simplest possible form.

3.13 An Example with Proof

$6n \log n + \sqrt{n} \log^2 n = \Theta(n \log n)$.
Need to find n_0, c_1, and c_2, such that
$c_1 n \log n \leq 6n \log n + \sqrt{n} \log^2 n \leq c_2 n \log n$
$c_1 = 5$, $c_2 = 7$, $n_0 = \ldots$
Note: Implications go up. That is, we want to find something at the bottom that will logically imply the statement we're starting with.

$$\begin{aligned}
5n_0 \log n_0 &\leq 6n_0 \log n_0 + \sqrt{n_0} \log^2 n_0 \\
5n_0 \log n_0 &\leq 6n_0 \log n_0, n_0 > 1 \\
0 &\leq n_0 \log n_0, n_0 > 1 \\
n_0 &> 1
\end{aligned}$$

$$
\begin{aligned}
6n_0 \log n_0 + \sqrt{n_0} \log^2 n_0 &\leq 7n_0 \log n_0 \\
6n_0 + \sqrt{n_0} \log n_0 &\leq 7n_0 \\
\sqrt{n_0} \log n_0 &\leq n_0 \\
\log n_0 &\leq \sqrt{n_0} \\
n_0 &> 0
\end{aligned}
$$

So, $n_0 = 3$ suffices.