Asymptotic analysis, algorithm runtimes, GCDs

CS38 Recitation Notes
William Hoza

April 6, 2016

Asymptotic analysis

There is a handout posted to the CS38 website which carefully lays out the definitions of O, Ω, Θ, o, ω. We won't repeat those definitions here, but we'll do a couple of examples.

Proposition 1.

$$\ln \binom{2n}{n} \in \Theta(n).$$

Proof. First, we need to show that $\ln \binom{2n}{n} \in O(n)$. Let $[k] = \{1, \ldots, k\}$. Since $\binom{2n}{n}$ is the number of size-n subsets of $[2n]$, and there are 2^{2n} subsets total of $[2n]$, $\binom{2n}{n} \leq 2^{2n}$. Therefore, since ln is monotonic,

$$\ln \binom{2n}{n} \leq \ln(2^{2n}) = \frac{\log_2(2^{2n})}{\log_2 e} = \frac{2}{\log_2 e} \cdot n \in O(n).$$

Now, we need to show that $\ln \binom{2n}{n} \in \Omega(n)$. For any subset $S \subseteq [n]$ (of any size), there is a set $\widetilde{S} \subseteq [2n]$ of size n such that $\widetilde{S} \cap [n] = S$. Therefore, just by considering these sets $\widetilde{S}$, we see that $\binom{2n}{n} \geq 2^n$. Thus,

$$\ln \binom{2n}{n} \geq \ln(2^n) = \frac{1}{\log e} \cdot n \in O(n). \qquad \square$$

Since $\log_2(x)$ and $\ln(x)$ only differ by a constant factor, the base of your logarithm usually doesn't matter in the context of asymptotic analysis; $O(\log_2 n) = O(\ln n)$, so we just write $O(\log n)$. But you have to be careful, because constant factors *upstairs* matter (i.e. in the exponent):

Proposition 2.

$$e^{\ln^2 n} \in o\left(2^{\log_2^2 n}\right).$$

Proof. By log rules,

$$2^{\log_2^2 n} = n^{\log_2 n} = (n^{\ln n})^{1/\ln 2}$$
$$e^{\ln^2 n} = n^{\ln n}.$$

Note that $1/\ln 2 > 1$ and $n^{\ln n} \to \infty$ as $n \to \infty$. Therefore, for any $c > 0$ (no matter how small), for all sufficiently large n, $n^{\ln n} \leq c(n^{\ln n})^{1/\ln 2}$. $\square$

Algorithm runtimes

Recall the Fibonacci sequence F_N is defined recursively by

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{N+2} &= F_N + F_{N+1}. \end{aligned}$$

Comptuational problem: Given N, compute F_N.

The definition of the Fibonacci sequence practically *is* an algorithm for this problem, namely the naïve recursive algorithm.

Naïve recursive algorithm:

1. If $N \leq 1$, return N.
2. Recursively compute $a = F_{N-2}$ and $b = F_{N-1}$, and return $a + b$.

(For your problem sets, you should prove correctness of any algorithm you present, but we'll skip that step here.) This algorithm is terribly slow, because in the recursion steps, it ends up computing the same Fibonacci numbers over and over again. In particular, if we let $T(N)$ be the number of additions performed by this algorithm on input N, then (for $N \geq 2$) we have $T(N) = 1 + T(N-1) + T(N-2)$. Since T is clearly monotone, this implies that $T(N) \geq 2T(N-2)$, and hence $T(N) \in 2^{\Omega(N)}$. So the number of arithmetic steps performed by this algorithm grows *exponentially in* N.

We can dramatically improve on the naïve recursive algorithm by storing the Fibonacci numbers that we compute, so that we don't needlessly recompute them. This is a very simple example of *dynamic programming*.

Dynamic programming algorithm:

1. Initialize an array F of integers with $F[0] = 0$ and $F[1] = 1$.
2. For $i = 2$ to N, set $F[i] = F[i-1] + F[i-2]$.
3. Return $F[N]$.

(Again, we'll skip the correctness proof.) The number of arithmetic steps performed by this algorithm is only $O(N)$. Can we get another dramatic, exponential improvement? The transformation $\mathbb{R}^2 \to \mathbb{R}^2$ defined by

$$\begin{pmatrix} a \\ b \end{pmatrix} \mapsto \begin{pmatrix} b \\ a+b \end{pmatrix}$$

is *linear*, represented by the matrix

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

In particular,

$$M \begin{pmatrix} F_N \\ F_{N+1} \end{pmatrix} = \begin{pmatrix} F_{N+1} \\ F_{N+2} \end{pmatrix}.$$

By repeated applications of this rule, we see that

$$M^N \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} F_N \\ F_{N+1} \end{pmatrix}.$$

From this perspective, the dynamic programming algorithm essentially computes $M^N \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ by performing N *matrix-vector* multiplications. (Start with the vector $v = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, and repeatedly set $v := Mv$.) But instead, we can compute M^N via *matrix-matrix multiplications*, and then perform just one matrix-vector multiplication at the end. This leads to huge savings in the number of arithmetic steps we need to perform, because of a trick called *repeated squaring*:

Repeated squaring algorithm:

1. Initialize an array P of 2×2 matrices with $P[0] = M$.
2. For $i = 1$ to $\lceil \log N \rceil$, set $P[i] = P[i-1]^2$, so that $P[i] = M^{2^i}$.
3. Initialize a 2×2 matrix $A =$ the identity matrix.
4. For $i = 0$ to $\lceil \log N \rceil$, set $A := A \cdot P[i]^{N_i}$, where N_i is the ith bit in the binary representation of N.
5. Return the first coordinate of $A \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

(Again, we'll skip a detailed proof of correctness, but note that the final value of A is $M^{\sum_i N_i 2^i} = M^N$.) Only a constant number of arithmetic operations are required to multiply two 2×2 matrices, so this repeated squaring algorithm only uses $O(\log N)$ arithmetic steps! This algorithm appears to be *much* better than the dynamic programming algorithm!

Alas, the speedup is essentially an illusion, as we'll see. Remember, an algorithm's runtime is only well-defined with respect to a particular *model* of computation and a particular *implementation* of that algorithm in terms of the chosen model.

The computational model for CS38

In CS21, the main model of computation was the Turing machine, which was largely motivated by pseudo-philosophical considerations of a mathematician at her desk, computing with paper and pencil. For CS38, the Turing machine is not the right model. We are now motivated by engineering concerns; we would like our model to more closely reflect the behavior of actual silicon-and-plastic computers. For example, accessing an element of an array should take constant time, whereas on a Turing machine, you have to wait for the read head to meander across the tape.

The "correct" model of computation for CS38 is called the RAM model ("random access machine".) It wouldn't be in the spirit of CS38 to give a really detailed definition of the RAM model, but here's the basic idea. We imagine an infinite sequence of *registers*, indexed by natural numbers, which can store arbitrary integers. A *program* consists of a sequence of instructions (think like low-level assembly code.) Each of these primitive instructions is assumed to take unit time. The instructions available include things like "copy the contents of register 6 into register 1", "jump to line 12 if the value of register 3 is nonzero", "halt", etc. Indirect addressing is supported, i.e. an instruction can say something like "copy into register 1 the contents of the register whose *index* is stored in register 6". (Registers can hold *pointers*.)

How about arithmetic? Some really basic arithmetic is available as primitive instructions, like incrementing, decrementing, and bit shifting. How about addition and multiplication? In the *bit model*, these operations need to be implemented in terms of operations on the individual bits of the operands, and do *not* take constant time. For example, the grade-school algorithm for adding two n-bit numbers takes time $O(n)$, and the grade-school algorithm for multiplying two n-bit numbers takes time $O(n^2)$ (though this can be improved upon.)

The bit model is the more realistic model. *In the real world, you cannot do arithmetic in constant time,* no matter what your programming instincts tell you. For CS38 problem sets, unless otherwise specified, you

should assume the bit model. But *sometimes* it is useful to work in the *word model*. In the word model, we have addition and multiplication available as primitive, unit-cost instructions – but only if the operands are "smallish," specifically if they have value poly(n) where n is the size of the input, or equivalently if they can be represented using $O(\log n)$ bits.

Finally, there is the outlandish model where arithmetic is unit-cost regardlesss of operand size. In this model, you can solve NP-complete and even PSPACE-complete problems in polynomial "time"![1] This is, in effect, the model we were adopting when we just counted the number of arithmetic operations performed by our Fibonacci algorithms.

Now, let's analyze the actual *runtimes* of the last two Fibonacci algorithms in the bit model. Recall that the Fibonacci numbers grow exponentially: $F_N \in 2^{\Theta(N)}$. Therefore, in the dynamic programming algorithm, the time required to compute $F[i]$ from $F[i-1]$ and $F[i-2]$ is $\Theta(i)$. So the overall runtime is $\Theta(N^2)$ (recall that $\sum_{i=1}^{N} i$ is $\Theta(N)$.)

For the repeated squaring algorithm, note that

$$M^N = \begin{pmatrix} F_{N-1} & F_N \\ F_N & F_{N+1} \end{pmatrix}.$$

Therefore, if we use the grade-school multiplication algorithm, the time to compute $P[i]$ from $P[i-1]$ is $\Theta((2^i)^2)$, and the time to perform the ith multiplication involved in forming A is also $\Theta((2^i)^2)$. So the overall runtime is once again $\Theta(N^2)$, because $\sum_{i=1}^{\lceil \log N \rceil} 2^{2i} \in \Theta(2^{2 \log N})$. (A geometric series is dominated by its largest term.) So the dynamic programming algorithm and the repeated squaring algorithm have the same asymptotic runtimes, when you take into account the cost of arithmetic! (Again, in your problem sets, you should be more detailed in your runtime analyses.) Is there any hope of designing algorithm for computing F_N which legitimately has runtime $o(N)$? No - it's going to take $\Omega(N)$ steps just to write down the answer.

A GCD problem

Exercise: Design a quadratic-time algorithm which, given nonnegative integers $a_1, \ldots, a_k$, computes $\mathrm{LCM}(a_1, \ldots, a_k)$, using the two-argument Euclidean GCD algorithm as a subroutine. You may assume that the greatest common divisor $\mathrm{GCD}(a, b)$, the product $a \cdot b$, and the quotient $\lceil a/b \rceil$ can all be computed in $O(\log(a) \cdot \log(b))$ time.

Solution sketch: Observe that $\mathrm{LCM}(a_1, \ldots, a_k) = \mathrm{LCM}(a_1, \mathrm{LCM}(a_2, \ldots, a_k))$. Furthermore,

$$\mathrm{LCM}(a, b) = \frac{ab}{\mathrm{GCD}(a, b)}.$$

So the following algorithm is clearly[2] correct:

1. Initialize $\ell = a_k$.
2. For $i = k - 1$ down to 1:
   (a) Compute $g = \mathrm{GCD}(a_i, \ell)$.
   (b) Compute $q = a_i / g$.
   (c) Set $\ell := q \cdot \ell$.
3. Output ℓ.

What is the runtime of this algorithm? Let $b_i = \lceil \log a_i \rceil$. We can bound the numbers of bits of the integers involved in iteration i of the loop on line 2 as follows:

[1] Schönhage, Arnold. *On the power of random access machines.* Springer Berlin Heidelberg, 1979.

[2] Again, this would not be good enough for your problem sets

- $\mathrm{LCM}(a_{i+1}, \ldots, a_k) \leq a_{i+1} \cdots a_k$, so the number of bits needed for ℓ is at most $b_{i+1} + \cdots + b_k$.
- $\mathrm{GCD}(a, b) \leq b$, so the number of bits needed for g is also at most $b_{i+1} + \cdots + b_k$.
- $a_i/g \leq a_i$, so the number of bits needed for q is at most b_i.

Therefore, the total runtime of the algorithm is

$$\begin{aligned}
&O\left(\sum_{i=1}^{k} \text{time for gcd} + \text{time for division} + \text{time for multiplication}\right) \\
&= O\left(\sum_{i=1}^{k} 3 \cdot b_i \sum_{j=i+1}^{k} b_j\right) \\
&= O\left(\sum_{i=1}^{k} \sum_{j=1}^{k} b_i b_j\right) \\
&= O\left(\left(\sum_{i=1}^{k} b_i\right)^2\right) \\
&= O\left((\text{size of input})^2\right).
\end{aligned}$$