CS 3510 Algorithms 1/09/2023

Lecture 1: Introduction, Fibonacci, and Big-O

Lecturer: Abrahim Ladha *Scribe(s): Saigautam Bonam*

Why are algorithms so important? Primarily it is a useful, interesting, and foundational theory. It can also help you make a lot of money.

An algorithm is a process to compute something. The runtime of an algorithm is the number of steps it takes as a function of the input size.

1 Fibonacci

Consider the problem of finding the nth Fibonacci number. These are $0, 1, 1, 2, 3, 5, 8, \ldots$ and so on, beginning from F_0. There is a well-known recurrence with a base case of $F_0 = 0$ and $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$. This recurrence gives us an obvious algorithm:

```
def f1(n):
    if n == 0, 1:
        return n
    else:
        return f1(n - 1) + f1(n - 2)
```

How many steps does this algorithm take? Let $T(n)$ be the number of steps the algorithm takes on input n. Note that the time is dependent on its recursive calls, so

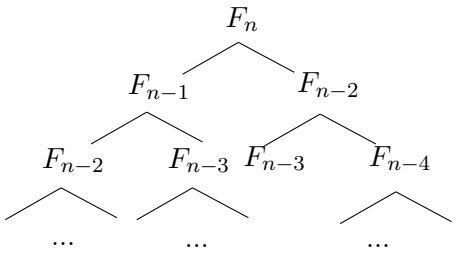
$$T(n) = T(n-1) + T(n-2) + 3$$

Note that 3 is just a constant that describes the amount of extra work outside of the recursive calls. We'll see soon that the actual number doesn't matter.

We have that $T(n-1) > T(n-2)$, then $T(n) > 2T(n-2) + 3$. If we use the same idea, we will get $T(n) > 4T(n-4) + 3$ and so on. This ultimately gets us to the following inequality:

$$T(n) > 2^{n/2}$$

This means $T(n)$ grows exponentially with respect to n This is very bad and slow!. Practically, I ran this for 43 days to compute F_{70} on a server and someone turned off the server before it finished. Slow algorithms don't simply mean you wait longer for the answer, but you may experience the heat death of the universe before you get an answer. You may also notice by the recurrence we can see that $T(n) > F_n$, but the Fibonacci's also grow exponentially.



Note in the recursion tree, we actually recompute a massive amount of what is needed. As a human, this is not the way we compute Fibonacci numbers! We do so iteratively, writing down and re-using previous answers. Our pen-and-paper process of computing the Fibonacci's is motivation for our next algorithm:

```
def f2(n):
    if n == 0:
        return 0
    arr = [0] * (n + 1)
    arr[0] = 0
    arr[1] = 1
    for i in range(2, n + 1):
        arr[i] = arr[i - 1] + arr[i - 2]
    return arr[n]
```

We loop n times, and at each loop iteration we perform one addition. For now let's overestimate this addition takes n time (based on the number of bits of each number), and so we get that an upper bound for this algorithm is $T(n) < n^2$. This shows the algorithm is faster than the previous one.

Let's examine another approach to Fibonacci. Consider the following:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix}$$

Multiplication by by the matrix computes Fibonacci numbers!

```
def f3(n):
    A = [[0 1][1 1]]
    v = [0 1]
    compute A^n
    compute A^n v
    return F_n
```

$T(n) = n - 1$ matrix multiplications, one vector multiplication and some constant work.

$$T(n) = (n-1)MMs + (1)VM + c$$

Let's not worry about the costs of a 2×2 matrix multiplication yet, but the number, just for now. Can we reduce the number of matrix multiplications? By repeatedly squaring like $A = A \cdot A$ we only need to perform $\log(n)$ matrix multiplications. We will go into this algorithm for exponentiation in lecture three.

$$T(n) = (\log n)MMs + (1)VM + c$$

You may recall diagonalization of a matrix. If a matrix A is diagonalizable, there exists D, P where $A = PDP^{-}1$, where

$$D = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

We only care about this since

$$\begin{aligned} A^n &= (PDP^{-1})^n \\ &= (PDP^{-1})(PDP^{-1})(PDP^{-1})\dots \\ &= PD(P^{-1}P)D(P^{-1}P)D\dots \\ &= PD^nP^{-1} \end{aligned}$$

That's helpful to us because

$$D^n = \begin{bmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{bmatrix}$$

Although we don't know it yet, computing PD^nP^{-1} is much more efficient than computing A^n.

```
def f4(n):
    D = ...
    P = ...
    P^-1 = ...
    v = ...
    compute D^n
    F_n = PD^nP^-1 v
    return F_n
```

What is our runtime? we have two matrix multiplications, a vector multiplication, and two integer exponentiations to power n.

$$T(n) = (2)MMs + (1)VM + (2)exps + c$$

We don't have the tools yet to know this is faster. But the moral here is that a deeper knowledge of mathematical tools can only improve run time. In fact, using this, we can derive a closed formula for F_n. Rather than plug in n, then compute the matrix product, compute the matrix produce in terms of some variable n and then simplify it. Computing $A^n = PD^nP^{-1}$ algebraically, we get the closed form

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Note that analyzing the runtime becomes more difficult. I want you to think about what it costs to compute powers of roots.

2 Big-O

Let us discuss how to better analyze runtime. What is a "basic computer step"? Is it an instruction? We consider the runtime in the size of the input, denoted as n. It could be an array of size n or numbers with n bits. Computing the exact number of steps can be unnecessarily complicated. We use big O notation to get to the heart of the matter.

For $f, g : \mathbb{N} \to \mathbb{R}$, we say $f(n)$ is $\mathcal{O}(g(n))$ if there is a constant c such that $f(n) \leq c(g(n))$ for large enough n. Note we care about asymptotics, and y may be less than f on small n. Most texts say $f(n) = \mathcal{O}(g(n))$, but I like to say "is". This big-O is a property of f rather than its equivalence.

We say $f(n)$ is $o(g(n))$ (little-oh) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Although this is a formal definition, you may think of $f(n)$ is $o(g(n))$ to mean that f grows **strictly** less than g asymptotically. f is never $o(f)$.

We say $f(n)$ is $\Omega(g(n))$ if there exists a constant c such that $f(n) \geq c(g(n))$ for large enough n.

We say that $f(n)$ is $\Theta(g(n))$ if $f(n) = \Omega(g(n))$ and $f(n) = \mathcal{O}(g(n))$. They are matching upper and lower bounds.

We say that $f(n)$ is $\omega(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ diverges.

Although you should use the formal definitions of all of these, it may be helpful to remember them like this:

$$\mathcal{O} \to f \leq g$$
$$\Omega \to f \geq g$$
$$o \to f < g$$
$$\omega \to f > g$$
$$\Theta \to f = g$$

Recall the following common bounds you might have seen already:

$$\mathcal{O}(1), \mathcal{O}(\log n), \mathcal{O}(n), \mathcal{O}(n \log n), \mathcal{O}(n^2), \mathcal{O}(2^n), \mathcal{O}(n!)$$

and more. Note that most algorithms will be at least $\Omega(n)$, as it usually takes at least n time to process the input (for example, an array of n integers). Any algorithm that doesn't look at the whole input may be faster (for example, binary search).