CS3510 L1

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

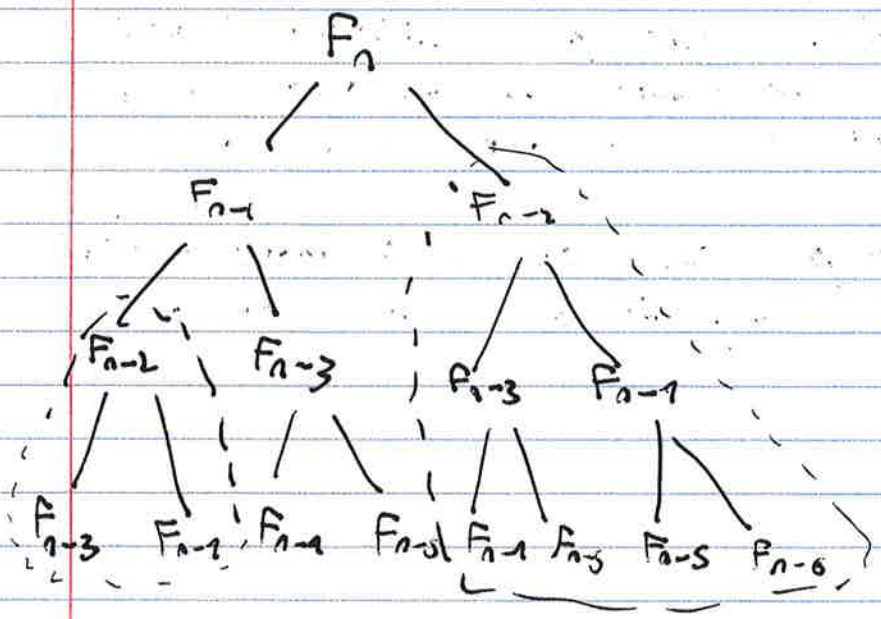
An algorithm is a process to compute something. The runtime of an algorithm is the number of steps it takes as a function of the input size.

Consider the problem of finding the n^{th} Fibonacci number. They are 0, 1, 1, 2, 3, 5, 8, 13, and so on, beginning from F_0. The recurrence is well known 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 it takes on input n. Note the time is dependent on its recursive calls, so $T(n) = T(n-1) + T(n-2) + O(1)$. Note that since $T(n-1) > T(n-2)$, then $T(n) > T(n-2) + T(n-2) + O(1) = 2T(n-2)$. It follows then that $T(n) > 4T(n-4)$ and $8T(n-6)$ so $T(n) > 2^{n/2}$. You may also note $T(n) > F_n$. Meaning $T(n)$ grows exponentially! This is very bad and slow. Practically, I ran this for 43 days to compute $F_1(90)$ and then someone turned the computer off before it finished.

2



Note in the recursion tree, we actually recompute a massive amount of tasks needed. Also note although this is the recurrence, this is not how you compute the Fibonacci's pen-and-paper style. You do so iterations, writing down and reusing previous numbers.

```
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, perform one addition, for now let us estimate this as taking time n and return to 2. n is an overestimation, but we get the bound that $T(n) < n^2$ This shows the algorithm is superior than f1.

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

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

Multiplication by this matrix computes fibonacci numbers! with the vector as the base case.

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

```
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 mults + 1 vector mult. lets not worry what those cost yet. Can we reduce the number of matrix mults? by repeatedly doubling i.e. $A = A \cdot A$, we can we can only need to perform $\log(n)$ many matrix mults! we will go into this later.

You may recall diagonalization of a matrix. If a matrix A is diagonalizable, there exists a matrix P, D where $A = PDP^{-1}$ where D only has (eigenvalues) on its diagonal like $D = \begin{bmatrix}\lambda_1&0\\0&\lambda_2\end{bmatrix}$. we care about this since $A^n = (PDP^{-1})^n$

$= (PDP^{-1})(PDP^{-1})(PDP^{-1})\ldots = PD(P^{-1}P)D(P^{-1}P)D(P^{-1}P)\ldots$

$= PD^nP^{-1}$ This matters as D being diagonal means $D^n = \begin{bmatrix}\lambda_1^n&0\\0&\lambda_2^n\end{bmatrix}$

24

```
def P4(n)
    D = ...
    P = ...
    P^-1 = ...
    v = ...
    compute D^n
    compute P D^n P^-1 v
    return F_n
```

~~This~~ T(n) = 2 matrix mults + 2 exponentiations to power n + 1 vector mult.

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 times. In fact, using this, we can derive a closed formula for F_n. Compute $P\begin{bmatrix}\lambda_1^n & \\ & \lambda_2^n\end{bmatrix}P^{-1}$ only in terms of n, then evaluate last. This gives us the equation

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

Note that now analyzing the run time becomes more difficult.

5

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, we usually denote the size as n. Maybe if it is an array of n elements or a number of 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 $O(g(n))$ if there is a constant c such that $f(n) \leq c \cdot g(n)$ for large enough n. Note we care about asymptotics, and g may be less than f on small n. Most texts say $f(n) = O(g(n))$, 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 larger enough n

We say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$. There are matching upper and lower bounds.

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

6

Although you should use the formal definitions, it may be helpful to think of them like this

O	$f \leq g$
o	$f < g$
Ω	$f \geq g$
ω	$f > g$
Θ	$f = g$

Hopefully you recall some of the following
$O(1)$, $O(\log\log n)$ $O(\log n)$ $O(n)$ $O(n \log n)$ $O(n^2)$

$O(2^n)$ $O(n!)$ and more. Note that most algorithms will be atleast $\Omega(n)$, as it takes this much time to read the input. Any sub linear time algorithm doesn't look at the whole input. Such as binary search.

You will exercise your use of the definitions in the next homework.