Reminder: asymptotic notation

(copy fib(n))

- $f(n)$ is $O(g(n))$:

  $\exists c, n_0 > 0 \quad \forall n \geq n_0 \quad f(n) \leq c \cdot g(n)$

- $f(n)$ is $\Omega(g(n))$:

  $\exists c, n_0 > 0 \quad \forall n \geq n_0 \quad f(n) \geq c \cdot g(n)$

- $f(n)$ is $\Theta(g(n))$:

  $f(n)$ is $O(g(n))$ and ($f(n)$ is) $\Omega(g(n))$

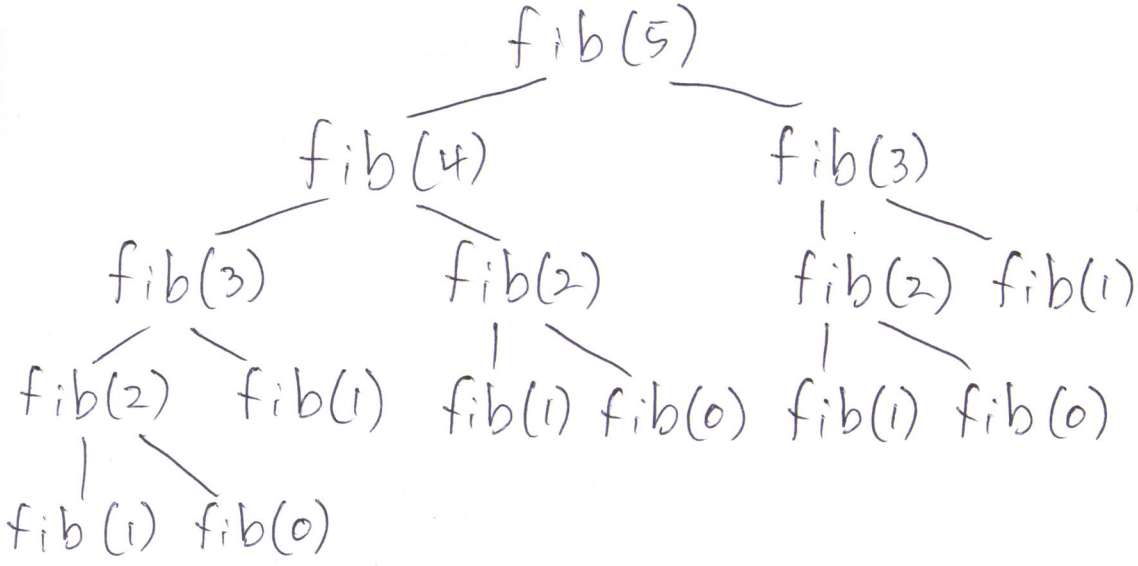
- ~~Let~~These let us ignore constant factors and lower-order terms.
- They shift the focus to the dependency on n, when n gets very large.

Last lecture, we showed that fib(n) takes time $\Omega(2^{n/2})$.

~~Thus, fib(n) takes exponential time.~~

Why is it so slow?

Consider the recursion tree for fib(5):



Lots of duplicate work.

We can do better:

Algorithm fib2(n):

```
if n≤1 then return n          ... O(1)
else
    initialize array f[0..n]  ... O(n)
    f[0]←0; f[1]←1;           ... O(1)
    for i←2 to n do           ... n×
        f[i]←f[i-1]+f[i-2]    ...   O(1)
    return f[n]               ... O(1)
```

(Pre-view of dynamic programming)

- Correct? Y
- Terminates? Y
- Efficiency? ~~linear in n~~ $O(n)$.

Divide-and-Conquer

To solve a problem:

1. divide the problem into smaller instances of the same problem
2. conquer: Recursively solve each smaller ~~problem~~ instance from step 1.
3. merge the solutions into a solution of the original instance.

The hard parts are step 1 and 3:

- How do we decompose the larger instance into smaller ones?
- How do we combine the smaller solutions into a solution for the big ~~problem~~ instance?

Example: Find the minimum element in an array $A[1..n]$.

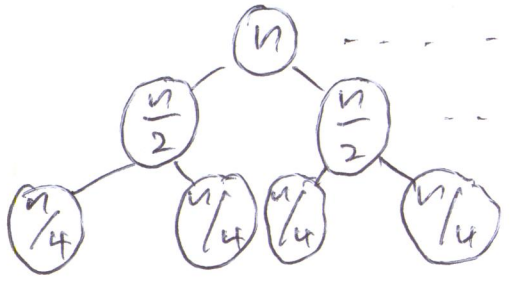
- Divide? $A[1..n/2]$, $A[n/2+1..n]$
- Merge? min

Algorithm findMin(A[1..n])
if n = 1 then return A[1]
else a ← findMin(A[1..n/2])
b ← findMin(A[n/2..n])
return min(a, b)

- Correct?
  For n=1, yes.
  For n>1, assume that alg. is correct for ~~Lenck~~ smaller inputs. ~~If a and b~~ Then a and b are the true minima of the first and second half of the array. Since the minimum element has to be in one of the two, min(a,b) is the minimum element.
- Terminates? Yes - problems always get smaller.
- Efficient?

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ 2 \cdot T(n/2) + O(1) & \text{o.w.} \end{cases}$$

Draw the recursion tree:



$1 \cdot O(1)$
$2 \cdot O(1)$
$4 \cdot O(1)$
$2^i \cdot O(1)$
1 1 1 1 1 1 1 1 1 ... $n \cdot O(1)$

The amount of work done at ~~each~~ level i is $2^i \cdot O(1)$.

The ~~the~~ number of levels is determined by when the node size ($n/2^i$) becomes 1.

$$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$$

So there are $\log_2 n$ levels. That makes the total amount of work:

$$\sum_{i=0}^{\log_2 n} (2^i \cdot O(1)) = O(1) \cdot \sum_{i=0}^{\log_2 n} 2^i$$

Recall: $\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1}$

$$= O(1) \cdot \frac{2^{\log_2 n + 1} - 1}{2 - 1}$$

$$= O(1) \cdot (2^{\log_2 n + 1} - 1)$$

$$= O(1) \cdot (2 \cdot 2^{\log_2 n} - 1)$$

$$= O(1) \cdot (2 \cdot n - 1)$$

$$= O(2n - 1)$$

$$= O(n)$$