Big-Oh Notation

- Suppose we have two different algorithms/programs which both ~~do~~ compute the same thing; i.e. their final outputs are the same on all inputs

- On an input of size n,

  Algorithm A_1 executes ~~T(n)~~ $f(n)$ instructions.

  Alg. A_2 executes $g(n)$ instructions.

<u>Question</u>: Which algorithm is faster?

$\Rightarrow$ A_1 might be faster than A_2 on some inputs, while A_2 might be faster than A_1 on other inputs

$\Rightarrow$ Often, we will be able to make one of three statements:

- On <u>large</u> inputs, A_1 is significantly faster
- On <u>large</u> inputs, A_2 is significantly faster
- On <u>large</u> inputs, A_1 and A_2 are "in the same ballpark"
  (e.g. A_1 is twice as fast as A_2 or A_2 is 1,000,000,000 x as fast as A_1.)

What is important to us is how the ratio $\frac{f(n)}{g(n)}$ behaves as n grows.

def Let $\mathbb{R}^+ = \{ r \in \mathbb{R} \mid r \geq 0 \}$ be the set of non-negative real values ~~at~~ and let $f, g : \{1, 2, \ldots\} \to \mathbb{R}^+$ be functions

We say:

- $\boxed{f(n) = O(g(n))}$ if there are positive numbers n_0 and c such that $\forall n \geq n_0 \quad f(n) \leq c \cdot g(n)$.

- $\boxed{f(n) = \Omega(g(n))}$ if there are positive numbers n_0 and c such that

$$\forall n \geq n_0 \quad f(n) \geq c \cdot g(n)$$

- $\boxed{f(n) = \Theta(g(n))}$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

<u>Remark</u>: These definitions are a little bit complex to ~~mean~~ deal with the possibility that $g(n) = 0$ for some n.

If $g(n) > 0$ for all sufficiently large n,

then

$f(n) = O(g(n)) \iff$ for all sufficiently large n, $\frac{f(n)}{g(n)} \le$ a constant

$f(n) = \Omega(g(n)) \iff$ for all sufficiently large n, $\frac{f(n)}{g(n)} \ge$ a positive const

$f(n) = \Theta(g(n)) \iff$ for all sufficiently large n,

pos. const $\le \frac{f(n)}{g(n)} \le$ const

Ex • $f(n) = 28n$, $g(n) = \frac{1}{100} n^2$.

Note $\frac{f(n)}{g(n)} = \frac{28n}{\frac{1}{100} n^2} = \frac{2800}{n} \le 2800$

so $f(n) = O(g(n))$

- $f(n) = \binom{n}{3}$, $g(n) = n^3$.

$$\frac{f(n)}{g(n)} = \frac{n(n-1)(n-2)/6}{n^3} = \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \frac{1}{6} \leq \frac{1}{6}$$

so $f(n) = O(g(n))$.

Also, if $\cancel{\frac{n-1}{n}}$ $\frac{n-2}{n} \geq \frac{1}{2}$ (i.e. $n \geq 4$), then

$$\frac{f(n)}{g(n)} = \frac{1}{6} \cdot \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \geq \frac{1}{6} \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} \geq \frac{1}{24}$$

so $f(n) = \Omega(g(n))$.

Therefore $f(n) = \Theta(g(n))$.

- $f(n) = 3^n$, $g(n) = 2^n$

$$\frac{f(n)}{g(n)} = \frac{3^n}{2^n} = (1.5)^n \geq 1$$

so $f(n) = \Omega(g(n))$.

Because it is not true that $\frac{f(n)}{g(n)} \leq \text{const}$, "$f(n) = O(g(n))$" is false.

- $$f(n) = \begin{cases} n & n \text{ is odd} \\ n^2 & n \text{ is even} \end{cases}$$

$$g(n) = \begin{cases} n^2 & n \text{ is odd} \\ n & n \text{ is even} \end{cases}$$

$$\frac{f(n)}{g(n)} = \begin{cases} \frac{1}{n} & n \text{ is odd} \\ n & n \text{ is even} \end{cases}$$

n	1	2	3	4	5	6	7	8	...
$f(n)/g(n)$	1	2	1/3	4	1/5	6	1/7	8	

Note: the even values mean that

$$\frac{f(n)}{g(n)} \leq \text{const}$$

does not hold.

The odd values mean that

$$\frac{f(n)}{g(n)} \geq \text{positive const}$$

does not hold. So both "$f(n) = O(g(n))$" and

"$f(n) = \Omega(g(n))$" are false.

Facts

- $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

Think:

$f(n) = O(g(n))$ means $f \leq g$

$f(n) = \Omega(g(n))$ means $f \geq g$

$f(n) = \Theta(g(n))$ means $f \sim g$

Big-Oh asymptotic definitions are how theoretical computer sciences answers the question "Which algorithm is faster?"

Recursion Trees

- Let $T(n) = \begin{cases} 0 & n=1 \\ 2\cdot T(\frac{n}{2}) + n & n \geq 2 \text{ is even} \\ 2\cdot T(\frac{n-1}{2}) + n & n \geq 2 \text{ is odd.} \end{cases}$

- This is the run-time of fast sorting algs.
- How do we solve this recurrence?

First, ~~try~~ compute $T(n)$ for a few small n

n	1	2	3	4	5	6	7	8	9	10
$T(n)$	0	2	3	8	9	12	13	24	25	28

One thing you will notice is that when $n = 2^k$ is a power of two, $T(n) = T(2^k)$ only depends on $T(\frac{n}{2}), T(\frac{n}{4}), T(\frac{n}{8}), \ldots, T(1)$.

· This suggests we unroll the recursion // worry about solving $T(n)$ when n is a power of two. Let $n = 2^k$; note $\lg n = \lg 2^k = k$.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4T\left(\frac{n}{4}\right) + n + n$$

$$\cancel{= 2\left(2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right)\right) +}$$

$$= 4\left(2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n = 8 \cdot T\left(\frac{n}{8}\right) + 3n$$

$$\vdots$$

$$= 2^j \cdot T\left(\frac{n}{2^j}\right) + j \cdot n$$

Continuing until $j = k$, we get

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + k \cdot n$$

$$= n \cdot T(1) + k \cdot n$$

$$= n \cdot 0 + k \cdot n = k \cdot n = (\lg n) \cdot n$$

Exercise:

(1) Prove by induction on k that if $n=2^k$,

$$T(n) = n \lg n.$$

(2) Prove by induction on $n \geq 2$ that

$$T(n) \geq T(n-1).$$

Because $T(n)$ is non-decreasing and we know the value of $T(n)$ when n is a power of two, we already know enough about $T(n)$ for most purposes.

Let β be the smallest power of two with $\beta \geq n$. (Note $\beta \leq 2n$.)

$$T(n) \leq T(\beta) = \beta \lg \beta \leq 2n \lg(2n).$$

Let α be the largest power of two with $\alpha \leq n$. (Note $\alpha \geq \frac{n}{2}$.)

$$T(n) \geq T(\alpha) = \alpha \lg \alpha \geq \frac{n}{2} \lg\left(\frac{n}{2}\right).$$

Therefore

$$\frac{n}{2}\lg\left(\frac{n}{2}\right) \leq T(n) \leq 2n\lg 2n.$$

Exercise: Check that $T(n) = \Theta(n\lg n)$.

(Remember that in Lecture 13, we saw a sorting alg with runtime $T'(n) = \Theta(n^2)$.)

- It is not always the case that unrolling the recursion results in a recognizable pattern.

- Recursion Trees provide a systematic way to solve certain types of recurrences.

[illegible]

- Often, changing the base cases or "unimportant" parts of a recurrence T will result in another recurrence S with

$$S(n) = \Theta(T(n)).$$

- It is common to drop these unimportant details.

- Learning what is important and what is not takes practice.

<u>Ex</u>:

$$S_\alpha(n) = \begin{cases} \alpha & n = 1 \\ 2S_\alpha(\frac{n}{2}) + \frac{n}{2} & n \geq 2 \text{ \& even} \\ 2 \cdot S_\alpha(\frac{n-1}{2}) + \frac{n}{2} & n \geq 2 \text{ is odd} \end{cases}$$

<u>Remark</u>: $T(n) = S_0(n)$

If n is a power of two, then

$$S_\alpha(n) = n \cdot \alpha + n \lg n$$

and so regardless of the base case α, $S_\alpha(n) = \Theta(n \lg n)$. Thus, the particular base case chosen is unimportant.

Also,
$$R(n) = \begin{cases} 0 & n=1 \\ 2\cdot R(\frac{n}{2})+n & n \geq 2 \text{ is even} \\ 2\cdot R(\frac{n+1}{2})+n & n\geq 2 \text{ is odd} \end{cases}$$

also has solution $R(n) = \Theta(n \lg n)$.

Recursion Trees

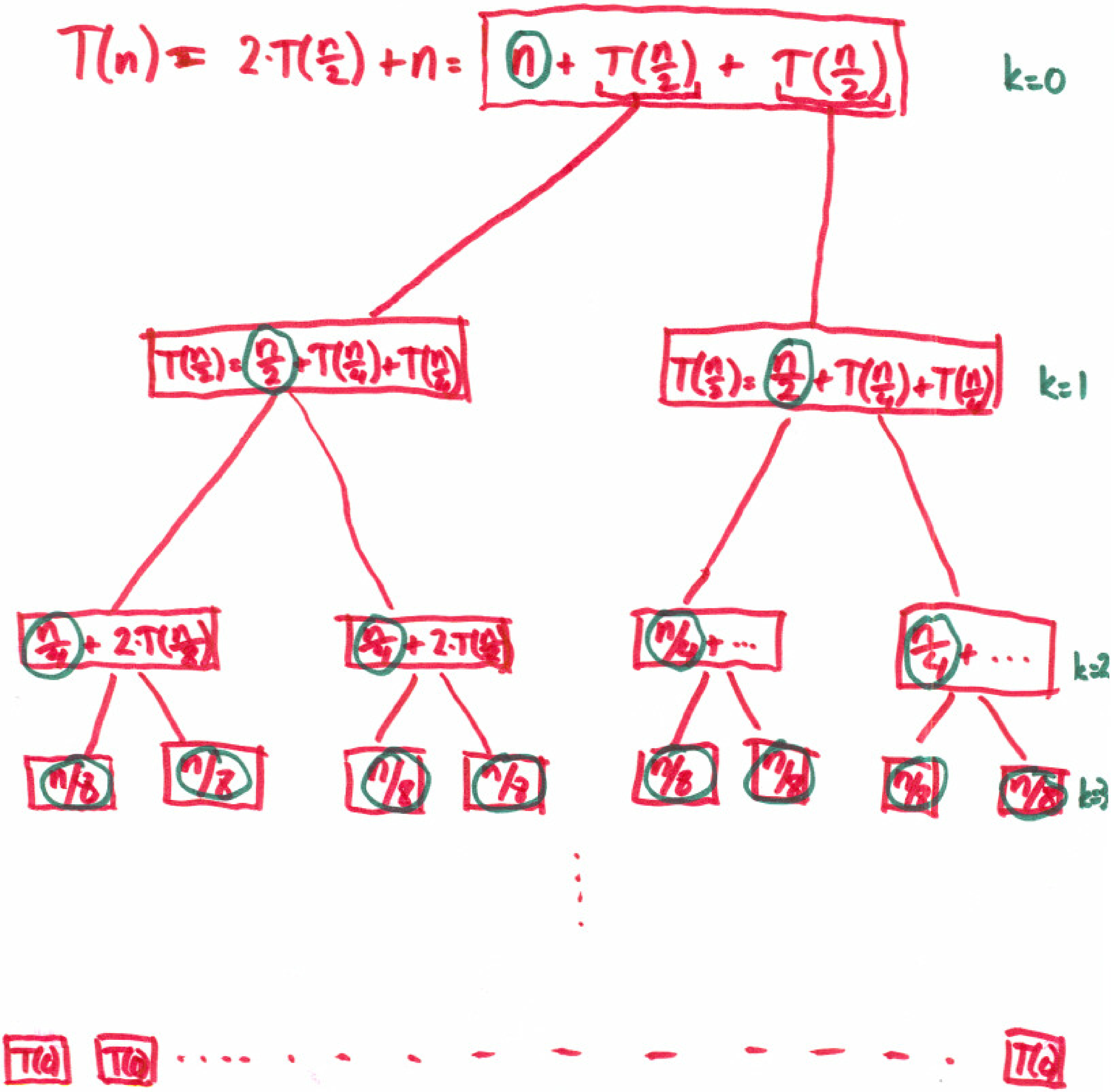
$T(n)$

- So, when n is odd, it does not matter if we round $\frac{n}{2}$ up (as does $R(n)$) or down (as does $S(n)$ and $T(n)$) when we make our recursive calls.

- Throwing away unimportant details, we write $T(n) = 2\cdot T(\frac{n}{2})+n$ and say $T(n) = \Theta(n \lg n)$.

Recursion Tree for $T(n) = 2\cdot T(\frac{n}{2}) + n$

- Recursion trees help us organize terms when we unroll the recursion. (Assume n is a power of two.)



depth	Contribution/node	# nodes	total at depth
0	n	1	$n \cdot 1 = n$
1	$n/2$	2	$n/2 \cdot 2 = n$
2	$n/4$	4	$n/4 \cdot 4 = n$
⋮			
k	$n/2^k$	2^k	$n/2^k \cdot 2^k = n$
⋮			
$d = \lg n$	$T(1)$	$2^d = n$	$T(1) \cdot n$

So, summing the contribution at each depth,

$$T(n) = \sum_{k=0}^{d-1} n + T(1) \cdot n$$

$$= d \cdot n + T(1) \cdot n$$

$$= n \lg n + T(1) \cdot n$$

$$= \Theta(n \lg n)$$