

Lecture 3

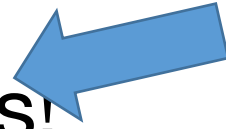
Recurrence relations and how to solve them!

Last time....

- Sorting: InsertionSort and MergeSort
- What does it mean to work and be fast?
 - Worst-Case Analysis
 - Big-Oh Notation
- Analyzing correctness of iterative + recursive algs
 - Induction!
- Analyzing running time of recursive algorithms
 - By drawing out a tree and adding up all the work done.

Today

- Recurrence Relations!



- How do we calculate the runtime of a recursive algorithm?

- The Master Theorem

- A useful theorem so we don't have to answer this question from scratch each time.

- The Substitution Method

- A different way to solve recurrence relations, more generally applicable than the Master Method.

Running time of MergeSort


- Let $T(n)$ be the running time of MergeSort on a length n array.
- We know that $T(n) = O(n \log(n))$.
- We also know that $T(n)$ satisfies:

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

```
MERGESORT(A):  
  n = length(A)  
  if n ≤ 1:  
    return A  
  L = MERGESORT(A[:n/2])  
  R = MERGESORT(A[n/2:])  
  return MERGE(L, R)
```

Running time of MergeSort

- Let $T(n)$ be the running time of MergeSort on a length n array.
- We know that $T(n) = O(n \log(n))$.
- We also know that $T(n)$ satisfies:

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$$


Last time we showed that the time to run MERGE on a problem of size n is $O(n)$. For concreteness, let's say that it's at most $11n$ operations.

```
MERGESORT(A):  
  n = length(A)  
  if n ≤ 1:  
    return A  
  L = MERGESORT(A[:n/2])  
  R = MERGESORT(A[n/2:])  
  return MERGE(L, R)
```

Recurrence Relations

Note (read after class):
 $T(n) \leq 2 \cdot T(\frac{n}{2}) + 11 \cdot n$ (with a \leq) is
also a recurrence relation. A
recurrence relation with an
“=” exactly defines a function; a
recurrence relation with an inequality
only bounds it.

- $T(n) = 2 \cdot T(\frac{n}{2}) + 11 \cdot n$ is a recurrence relation.
- It gives us a formula for $T(n)$ in terms of $T(\text{less than } n)$
- The challenge:
Given a recurrence relation for $T(n)$, find
a closed form expression for $T(n)$.
- For example, $T(n) = O(n \log(n))$

Technicalities I

Base Cases

- Formally, we should always have **base cases** with recurrence relations.

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$ with $T(1) = 1$

is not the same function as

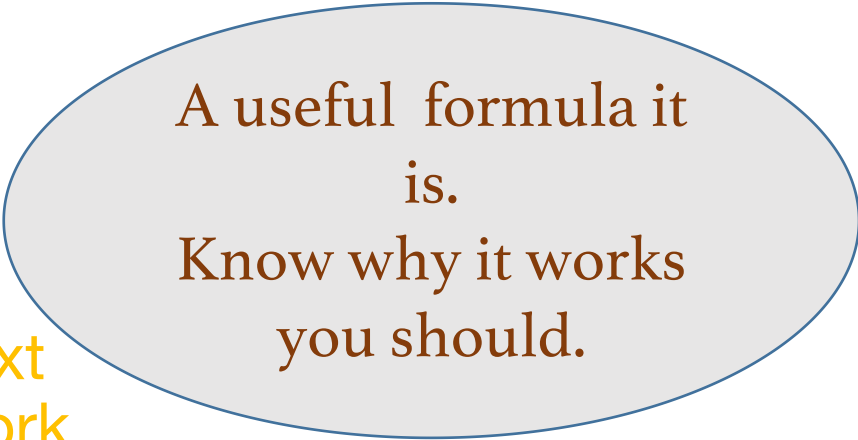
- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 11 \cdot n$ with $T(1) = 1000000000$

- However, no matter what T is, $T(1) = O(1)$, so sometimes we'll just omit it.

Why does $T(1) = O(1)$?

The master theorem

- A formula for many recurrence relations.
 - We'll see an example next lecture where it won't work.
- Proof: “Generalized” tree method.



A useful formula it
is.
Know why it works
you should.

The master theorem

We can also take n/b to mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$ and the theorem is still true.

- Suppose that $a \geq 1, b > 1$, and d are constants (independent of n).

- Suppose $T(n) = a \cdot T(\frac{n}{b}) + O(n^d)$. Then

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Show the Ω, Θ versions after lecture.

Three parameters:

a : number of subproblems

b : factor by which input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions.

Many symbols those are....

Technicalities II

Integer division

- If n is odd, I can't break it up into two problems of size $n/2$.

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n)$$

- However (see CLRS, Section 4.6.2 for details), one can show that the Master theorem works fine if you pretend that what you have is:

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

- From now on we'll mostly ignore floors and ceilings in recurrence relations.

Examples

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- Needlessly recursive integer mult.

- $T(n) = 4 T(n/2) + O(n)$
- $T(n) = O(n^2)$

$$\begin{array}{l} a = 4 \\ b = 2 \\ d = 1 \end{array} \quad a > b^d$$



- Karatsuba integer multiplication

- $T(n) = 3 T(n/2) + O(n)$
- $T(n) = O(n^{\log_2(3)} \approx n^{1.6})$

$$\begin{array}{l} a = 3 \\ b = 2 \\ d = 1 \end{array} \quad a > b^d$$



- MergeSort

- $T(n) = 2T(n/2) + O(n)$
- $T(n) = O(n \log(n))$

$$\begin{array}{l} a = 2 \\ b = 2 \\ d = 1 \end{array} \quad a = b^d$$



- That other one

- $T(n) = T(n/2) + O(n)$
- $T(n) = O(n)$

$$\begin{array}{l} a = 1 \\ b = 2 \\ d = 1 \end{array} \quad a < b^d$$



Proof of the master theorem

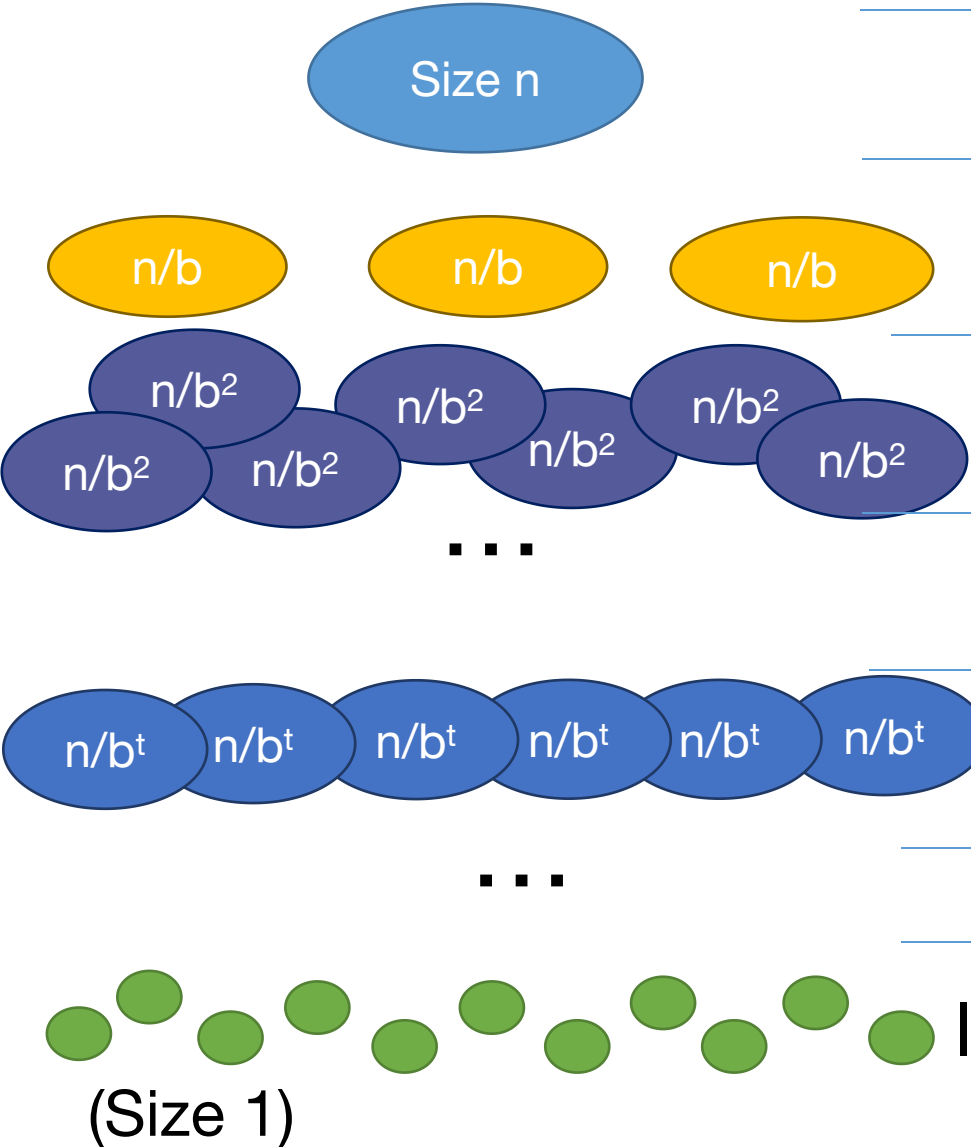
- We'll do the same recursion tree thing we did for MergeSort, but be more careful.
- Suppose that $T(n) \leq a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$.

Hang on! The hypothesis of the Master Theorem was that the extra work at each level was $O(n^d)$, but we're writing cn^d ...

That's true ... the hypothesis should be that $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. For simplicity, today we are essentially assuming that $n_0 = 1$ in the definition of big-Oh. It's a good exercise to verify why that assumption is without loss of generality.

Recursion tree

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$



Level	# problems	Size of each problem	Amount of work at this level
0	1	n	
1	a	n/b	
2	a^2	n/b^2	
...			
t	a^t	n/b^t	
...			
$\log_b(n)$	$a^{\log_b(n)}$	1	

Recursion tree




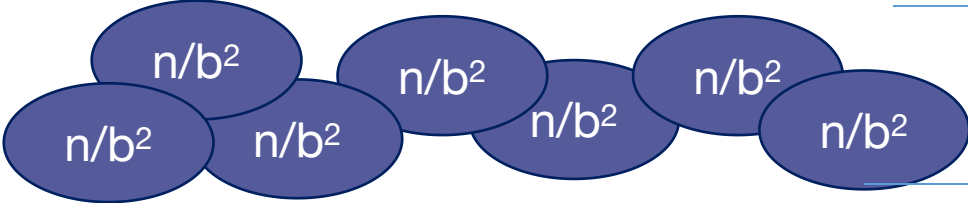
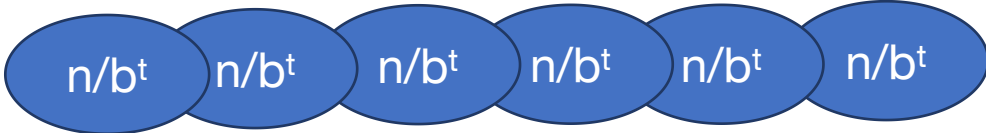
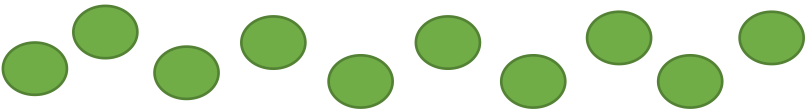
Help me fill this in!
level?

1 minutes: think
share (wait)

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$

How much work at each level?

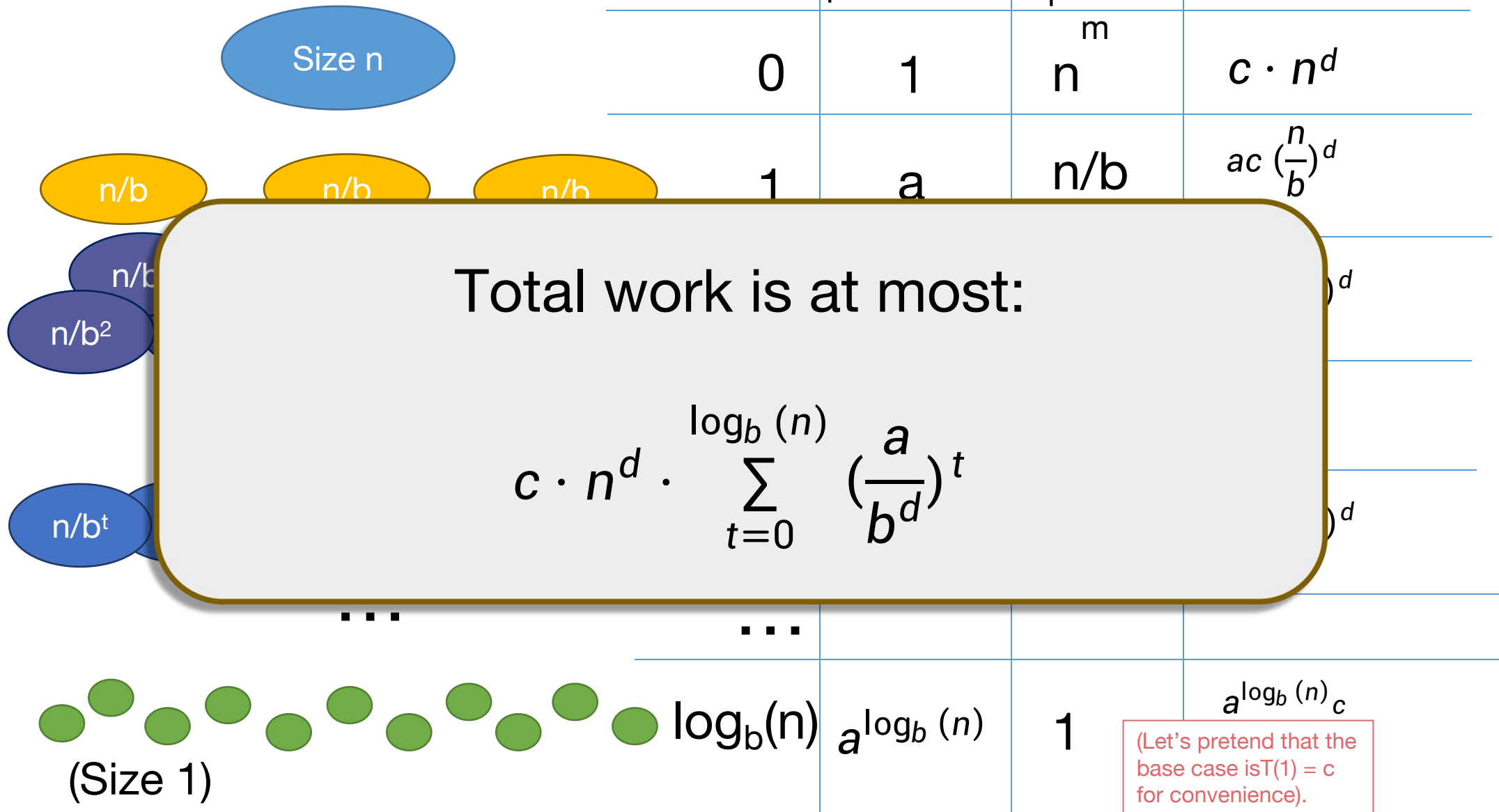
How much work total?

	Level	# problems	Size of each problem	Amount of work at this level
Size n	0	1	n	$c \cdot n^d$
	1	a	n/b	$a c \left(\frac{n}{b}\right)^d$
	2	a^2	n/b^2	$a^2 c \left(\frac{n}{b^2}\right)^d$
...				
	t	a^t	n/b^t	$a^t c \left(\frac{n}{b^t}\right)^d$
...				
 (Size 1)	$\log_b(n)$	$a^{\log_b(n)}$	1	$a^{\log_b(n)} c$

(Let's pretend that the base case is $T(1) = c$ for convenience).

Recursion tree

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$



Now let's check all the cases

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Case 1: $a = b^d$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- $$\begin{aligned} T(n) &= c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t \\ &= c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} 1 \\ &= c \cdot n^d \cdot (\log_b(n) + 1) \\ &= c \cdot n^d \cdot \left(\frac{\log(n)}{\log(b)} + 1\right) \\ &= \Theta(n^d \log(n)) \end{aligned}$$

← Equal to 1!

Case 2: $a < b^d$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- $T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$

~~Less than 1!~~

Aside: Geometric sums

- What is $\sum_{t=0}^N x^t$?
- You may remember that $\sum_{t=0}^N x^t = \frac{x^{N+1}-1}{x-1}$ for $x \neq 1$.
- Morally:

$$x^0 + x^1 + x^2 + x^3 + \dots + x^N$$

If $0 < x < 1$, this term
dominates.

$$1 \leq \frac{x^{N+1}-1}{x-1} \leq \frac{1}{1-x}$$

(Aka, $\Theta(1)$ if x is constant and N is growing).

(If $x = 1$, all
terms the
same)

If $x > 1$, this term dominates.

$$x^N \leq \frac{x^{N+1}-1}{x-1} \leq x^N \cdot \left(\frac{x}{x-1}\right)$$

(Aka, $\Theta(x^N)$ if x is constant and N is growing).

Case 2: $a < b^d$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- $$\begin{aligned} T(n) &= c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t \\ &= c \cdot n^d \cdot [\text{some constant}] \\ &= \Theta(n^d) \end{aligned}$$

~~Less than 1!~~

Case 3: $a > b^d$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- $$\begin{aligned} T(n) &= c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t \\ &= \Theta\left(n^d \left(\frac{a}{b^d}\right)^{\log_b(n)}\right) \\ &= \Theta(n^{\log_b(a)}) \end{aligned}$$

Larger than 1!



Convince yourself
that this step is legit!

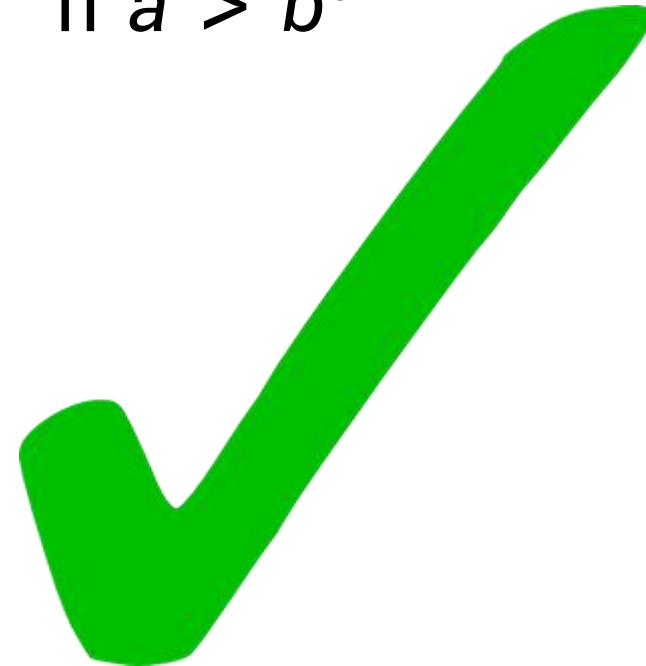


We'll do this step on the board!



Now let's check all the cases

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$



Even more generally, for $T(n) = aT(n/b) + f(n)$...

Theorem 3.2 (Master Theorem). *Let $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ be a recurrence where $a \geq 1$, $b > 1$. Then,*

- *If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, $T(n) = \Theta\left(n^{\log_b a}\right)$.*
- *If $f(n) = \Theta\left(n^{\log_b a}\right)$, $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.*
- *If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$.*

Figure out how to adapt
the proof we gave to
prove this more general
version!

[From CLRS – you won't
need to know this fancier
version for this class unless
you want to.]

Understanding the Master Theorem

- Let $a \geq 1$, $b > 1$, and d be constants.
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- What do these three cases mean?

The eternal struggle



Branching causes the
number of problems to
explode!
The most work is at the
bottom of the tree!

The problems lower
in the tree are
smaller! The most
work is at the top
of the tree!

Consider our three warm-ups

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

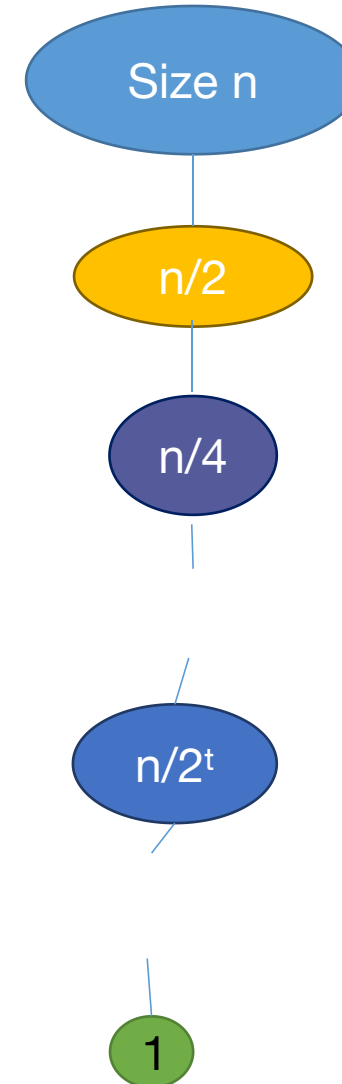
2. $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$

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

First example: tall and skinny tree

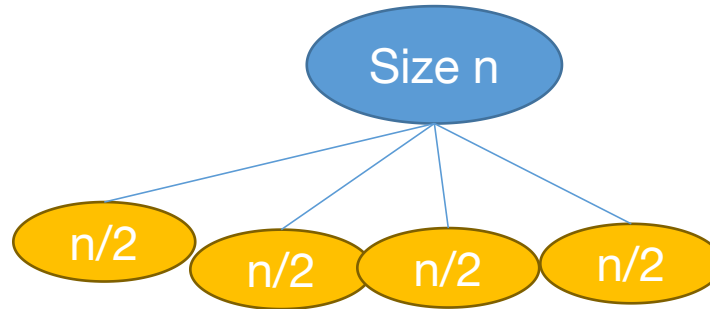
1. $T(n) = T\left(\frac{n}{2}\right) + n, \quad (a < b^d)$

- The amount of work done at the top (the biggest problem) swamps the amount of work done anywhere else.
- $T(n) = O(\text{work at top}) = O(n)$



Third example: bushy tree

$$3. \quad T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n, \quad (a > b^d)$$



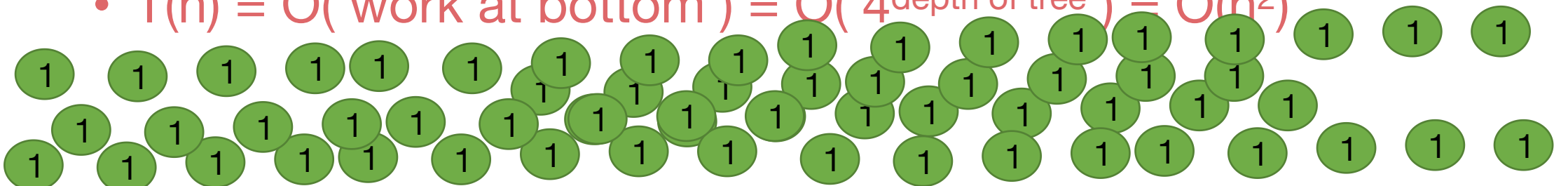
WINNER



Most work
at the
bottom of
the tree!

- There are a HUGE number of leaves, and the total work is dominated by the time to do work at these leaves.

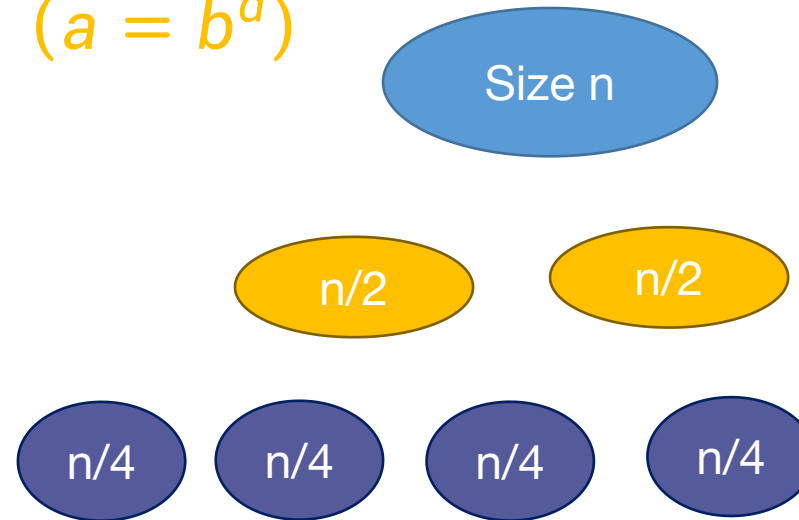
$$• \quad T(n) = O(\text{work at bottom}) = O(4^{\text{depth of tree}}) = O(n^2)$$



Second example: just right

$$2. \quad T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \quad (a = b^d)$$

- The branching just balances out the amount of work.
- The same amount of work is done at every level.



- $T(n) = (\text{number of levels}) * (\text{work per level})$
- $= \log(n) * O(n) = O(n \log(n))$



What have we learned?

- The “Master Method” makes our lives easier.
- But it’s basically just codifying a calculation we could do from scratch if we wanted to.

The Substitution Method

- Another way to solve recurrence relations.
- More general than the master method.
- **Step 1:** Generate a guess at the correct answer.
- **Step 2:** Try to prove that your guess is correct.
- **(Step 3: Profit.)**

The Substitution Method

first example

- Let's return to:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \text{ with } T(0) = 0, T(1) = 1.$$

- The Master Method says $T(n) = O(n \log(n))$.
- We will prove this via the Substitution Method.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \text{ with } T(1) = 1.$$

Step 1: Guess the answer

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

Expand
 $T\left(\frac{n}{2}\right)$

Simplify

Expand
 $T\left(\frac{n}{4}\right)$

Simplify

You can guess the answer however you want: meta-reasoning, a little bird told you, wishful thinking, etc. One useful way is to try to “unroll” the recursion, like we’re doing here.

Guessing the pattern: $T(n) = 2^t \cdot T\left(\frac{n}{2^t}\right) + t \cdot n$

Plug in $t = \log(n)$, and get

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

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \text{ with } T(1) = 1.$$

Step 2: Prove the guess is correct.

- Inductive Hypothesis: $T(n) = n(\log(n) + 1)$.
- Base Case ($n=1$): $T(1) = 1 = 1 \cdot (\log(1) + 1)$
- Inductive Step:
 - Assume Inductive Hyp. for $1 \leq n < k$:
 - Suppose that $T(n) = n(\log(n) + 1)$ for all $1 \leq n < k$.
 - Prove Inductive Hyp. for $n=k$:
 - $T(k) = 2 \cdot T\left(\frac{k}{2}\right) + k$ by definition
 - $T(k) = 2 \cdot \left(\frac{k}{2} (\log\left(\frac{k}{2}\right) + 1)\right) + k$ by induction.
 - $T(k) = k(\log(k) + 1)$ by simplifying.
 - So Inductive Hyp. holds for $n=k$.
- Conclusion: For all $n \geq 1$, $T(n) = n(\log(n) + 1)$

We're being sloppy here about floors and ceilings...what would you need to do to be less sloppy?

Step 3: Profit

- Pretend like you never did Step 1, and just write down:
- Theorem: $T(n) = O(n \log(n))$
- Proof: [Whatever you wrote in Step 2]

What have we learned?

- The substitution method is a different way of solving recurrence relations.
- **Step 1**: Guess the answer.
- **Step 2**: Prove your guess is correct.
- **Step 3**: Profit.
- We'll get more practice with the substitution method next lecture!

Another example (if time)

(If no time, that's okay; we'll see these ideas in Lecture 4)

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$
- $T(2) = 2$
- Step 1: Guess: $O(n \log(n))$ (divine inspiration).
- But I don't have such a precise guess about the form for the $O(n \log(n))$...
 - That is, what's the leading constant?
- Can I still do Step 2?

Step 2: Prove it, working backwards to figure out the constant

- **Guess:** $T(n) \leq C \cdot n \log(n)$ for some constant C
TBD.
- **Inductive Hypothesis (for $n \geq 2$) :** $T(n) \leq C \cdot n \log(n)$
- **Base case:** $T(2) = 2 \leq C \cdot 2 \log(2)$ as long as $C \geq 1$
- **Inductive Step:**

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$$
$$T(2) = 2$$

Inductive step

- Assume that the inductive hypothesis holds for $n < k$.
- $T(k) = 2T\left(\frac{k}{2}\right) + 32k$
- $\leq 2C\frac{k}{2} \log\left(\frac{k}{2}\right) + 32k$
- $= k(C \cdot \log(k) + 32 - C)$
- $\leq k(C \cdot \log(k))$ as long as $C \geq 32$.
- Then the inductive hypothesis holds for $n=k$.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$$
$$T(2) = 2$$

Step 2: Prove it, working backwards to figure out the constant

- **Guess:** $T(n) \leq C \cdot n \log(n)$ for some constant C TBD.
- **Inductive Hypothesis (for $n \geq 2$):** $T(n) \leq C \cdot n \log(n)$
- **Base case:** $T(2) = 2 \leq C \cdot 2 \log(2)$ as long as $C \geq 1$
- **Inductive step:** Works as long as $C \geq 32$
 - So choose $C = 32$.
- **Conclusion:** $T(n) \leq 32 \cdot n \log(n)$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 32 \cdot n$$
$$T(2) = 2$$

Step 3: Profit.

- Theorem: $T(n) = O(n \log(n))$
- Proof:
 - **Inductive Hypothesis:** $T(n) \leq 32 \cdot n \log(n)$
 - **Base case:** $T(2) = 2 \leq 32 \cdot 2 \log(2)$ is true.
 - **Inductive step:**
 - Assume Inductive Hyp. for $n < k$.
 - $T(k) = 2T\left(\frac{k}{2}\right) + 32k$
 - $\leq 2 \cdot 32 \cdot \frac{k}{2} \log\left(\frac{k}{2}\right) + 32k$ By the def. of $T(k)$
 - $= k(32 \cdot \log(k) + 32 - 32)$ By induction
 - $= 32 \cdot k \log(k)$
 - This establishes inductive hyp. for $n=k$.
 - **Conclusion:** $T(n) \leq 32 \cdot n \log(n)$ for all $n \geq 2$.
 - By the definition of big-Oh, with $n_0 = 2$ and $c = 32$, this implies that $T(n) = O(n \log(n))$

What have we learned?

- The substitution method is a different way of solving recurrence relations.
- **Step 1:** Guess the answer.
- **Step 2:** Prove your guess is correct.
- **Step 3:** Profit.
- We'll get more practice with the substitution method next lecture!

Why two methods?

- Sometimes the Substitution Method works where the Master Method does not.
- More on this next time!

Next Time

- What happens if the sub-problems are different sizes?
- And when might that happen?