Lecture 4: Analyzing Complexity via Recurrences

These slides include material originally prepared by Dr.Ron Cytron, Dr. Jeremy Buhler, and Dr. Steve Cole.

Announcements: Lab 3

- Pre-lab due **Tuesday** (tonight!) at 11:59 PM
- Code and Post-lab due **Friday** at 11:59 pm
  - Pre- and post-labs via Gradescope (usual writeup standards)
  - Code in your Bitbucket repo
  - Please **verify that your work has been checked in** by looking at your repo via the browser, and double-checking Gradescope
    - Avoid pull-before-push failure
- Academic Integrity (the other AI)
    - Many, many legitimate resources
    - Don't panic, even at last minute—reach out instead
      - Zero credit w/explanation is much, much better than an AI case
    - Previous semesters ==> many cases from Lab 3 reported to Dean's office
    - This semester ==> can we go for zero??

Announcements: Exam 1

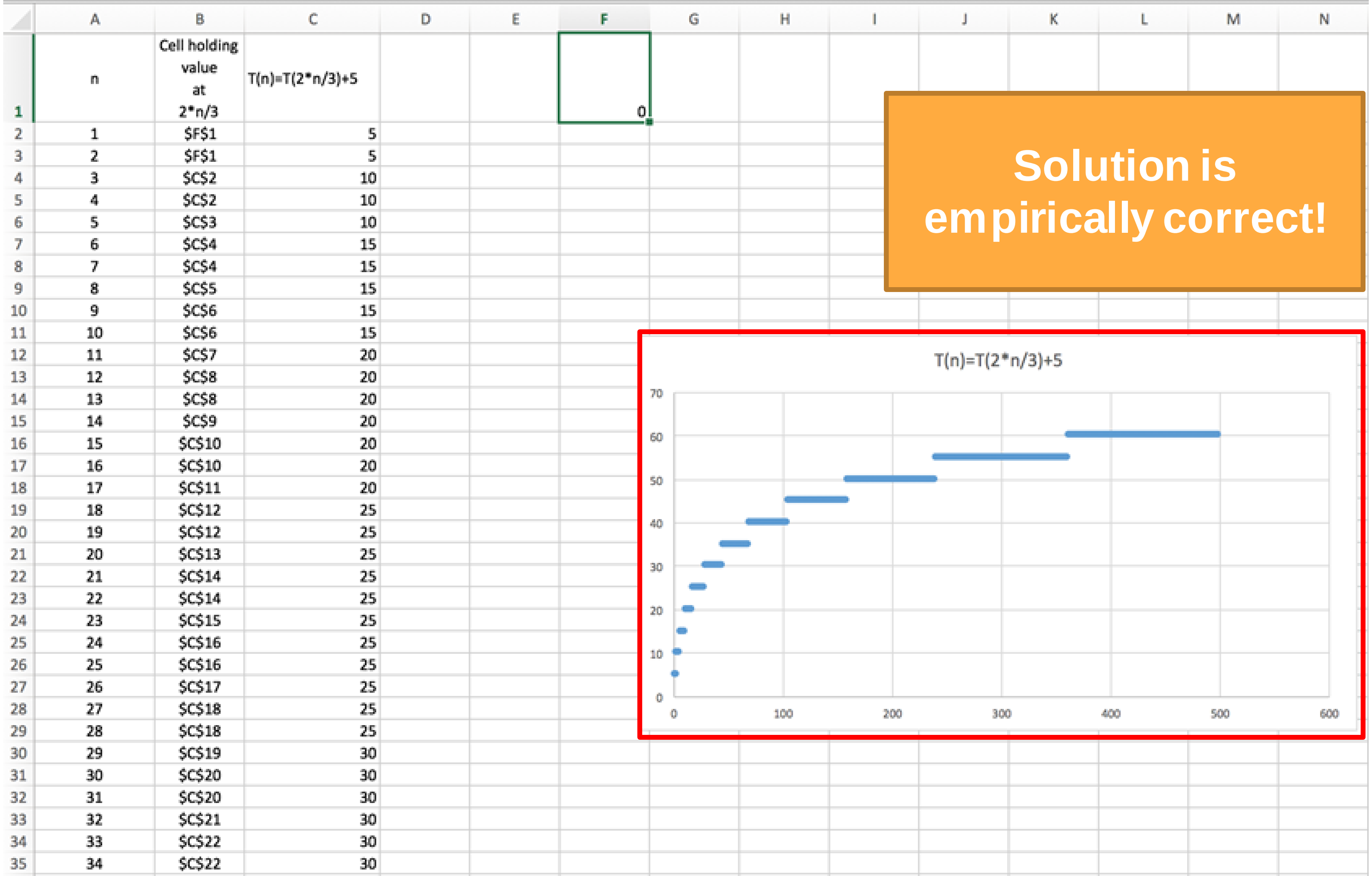
- **Wednesday 2/20 6:30-8:30 PM – rooms TBA (Piazza)**
- Please see Piazza for details (forthcoming), especially if you must reschedule for religious or other acceptable reasons
- Covers **Lectures** and **Studios 0-4**
- Exam review Sunday, Feb. 17, 2-5 pm Louderman 458 (instead of recitation)
- A practice exam will be posted this week

Last Time: Cost of `heapify`

- We gave a recursive procedure for heapify
- We defined its running time to be **T(n)** on a heap of size n
- We derived a *recursive formula* (***recurrence***) for T(n)

$$\mathbf{T(n) = T(2n/3) + k}$$

We magically solved this recurrence: $T(n) = \Theta(\log n)$



	A	B	C	F
1	n	Cell holding value at 2*n/3	T(n)=T(2*n/3)+5	0
2	1	F1	5	
3	2	F1	5	
4	3	C2	10	
5	4	C2	10	
6	5	C3	10	
7	6	C4	15	
8	7	C4	15	
9	8	C5	15	
10	9	C6	15	
11	10	C6	15	
12	11	C7	20	
13	12	C8	20	
14	13	C8	20	
15	14	C9	20	
16	15	C10	20	
17	16	C10	20	
18	17	C11	20	
19	18	C12	25	
20	19	C12	25	
21	20	C13	25	
22	21	C14	25	
23	22	C14	25	
24	23	C15	25	
25	24	C16	25	
26	25	C16	25	
27	26	C17	25	
28	27	C18	25	
29	28	C18	25	
30	29	C19	30	
31	30	C20	30	
32	31	C20	30	
33	32	C21	30	
34	33	C22	30	
35	34	C22	30	

Solution is empirically correct!

Strategies We Will Consider

- **Problem: given a recurrence for T(n), find a closed-form asymptotic complexity function that satisfies the recurrence.**

- Possible strategies
  - Guess and check (a.k.a. substitution)
  - Recursion tree accounting (for certain kinds of recurrence)
  - Master Method (next time)

Guess and Check

- Guess an **exact** (not asymptotic) function f(n) for T(n)
- Prove that f(n) satisfies the recurrence for all n > 0
- Proof is inductive on n
- [Requires that we know a base case for the recurrence]

Example

- $T(n) = T(n-1) + k$

- Let’s say $T(1) = k$

Example

- $T(n) = T(n-1) + k$
- Let's say $T(1) = k$
- **What solution should we guess?**

Example

- $T(n) = T(n-1) + k$
- Let’s say $T(1) = k$
- Intuitively, we add k every time n goes up by 1, so T(n) is something like nk.

Example

- $T(n) = T(n-1) + k$

- Let’s say $T(1) = k$

- Intuitively, we add k every time n goes up by 1, so T(n) is something like nk.

- **Claim: $T(n) = nk$**

Example: Proof [T(n) = T(n-1) + k]

- **Claim: T(n) = nk**
- **By induction on n**

Example: Proof [T(n) = T(n-1) + k]

- **Claim: T(n) = nk**
- **By induction on n**

- Bas (n=1): T(1) = k = 1*k ← claim holds!

Example: Proof [$T(n) = T(n-1) + k$]

- **Claim: $T(n) = nk$**
- **By induction on n**

- Bas ($n=1$): $T(1) = k = 1*k$ ← claim holds!

- Ind ($n > 1$): assume true for $m < n$.

Example: Proof [T(n) = T(n-1) + k]

- **Claim: T(n) = nk**
- **By induction on n**

- Bas (n=1): T(1) = k = 1*k ← claim holds!

- Ind (n > 1): assume true for m < n.
- T(n) = T(n-1) + k = (n-1)k + k

Example: Proof [$T(n) = T(n-1) + k$]

- **Claim: $T(n) = nk$**
- **By induction on n**

- Bas ($n=1$): $T(1) = k = 1*k$ ← claim holds!

- Ind ($n > 1$): assume true for $m < n$.
- $T(n) = T(n-1) + k =$ **$(n-1)k$** $+ k$

By IH, we can **algebraically substitute T(m) by** proposed **f(m)** for $m < n$ on the RHS

Example: Proof [$T(n) = T(n-1) + k$]

- **Claim: $T(n) = nk$**
- **By induction on n**

- Bas ($n=1$): $T(1) = k = 1*k$ ← claim holds!

- Ind ($n > 1$): assume true for $m < n$.
- $T(n) = T(n-1) + k = (n-1)k + k = nk$ ← claim holds!

- Conclude that $T(n)$ indeed $= nk = \Theta(n)$

A Slightly More Interesting Example

- **Binary search**: an algorithm for finding a value in a sorted array
- **Problem**: Given sorted array A of size n, and a ***query value*** x…
- If x occurs in A, return an index j s.t. A[j] = x
- If x does not occur in A, return special value "**notFound**"

3	**5**	**6**	**17**	**22**	**23**	**30**	**48**
0	1	2	3	4	5	6	7

Algorithm Idea

- Divide the array in half, and look at the middle element A[mid]
- If A[mid] < x, x must be in the __________ half of A if it appears at all.

Algorithm Idea

- Divide the array in half, and look at the middle element A[mid]
- If A[mid] < x, x must be in the **upper** half of A if it appears at all.
- If A[mid] > x, x must be in the _____ half of A if it appears at all.

Algorithm Idea

- Divide the array in half, and look at the middle element A[mid]
- If A[mid] < x, x must be in the **upper** half of A if it appears at all.
- If A[mid] > x, x must be in the **lower** half of A if it appears at all.
- In either case, recursively look for x in the appropriate half of A.

Example of Binary Search

Binary search

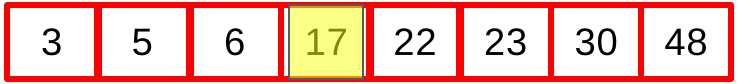
- Looking for 3
  - Try a middle element
  - From there, discard ½
  - Repeat

3	5	6	17	22	23	30	48

Example of Binary Search

Binary search

- Looking for 3
    - Try a middle element
    - From there, discard ½
    - Repeat

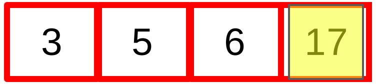


3	5	6	17	22	23	30	48

Example of Binary Search

Binary search

- Looking for 3
  - Try a middle element
  - From there, discard ½
  - Repeat



3	5	6	17

Example of Binary Search

Binary search

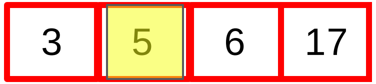
- Looking for 3
    - Try a middle element
    - From there, discard ½
    - Repeat

3	5	6	17

Example of Binary Search

Binary search

- Looking for 3
  - Try a middle element
  - From there, discard ½
  - Repeat

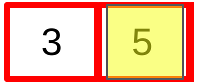


3	5	6	17

Example of Binary Search

Binary search

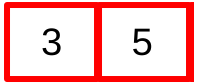
- Looking for 3
    - Try a middle element
    - From there, discard ½
    - Repeat



Example of Binary Search

Binary search

- Looking for 3
  - Try a middle element
  - From there, discard ½
  - Repeat

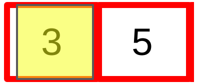


3	5

Example of Binary Search

Binary search

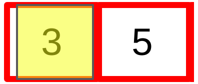
- Looking for 3
  - Try a middle element
  - From there, discard ½
  - Repeat



Example of Binary Search

Binary search

- Looking for 3
  - Try a middle element
  - From there, discard ½
  - Repeat
- Found it!



Binary Search

- You’ll study the code and correctness of binary search more deeply in Studio 5.

- For today, let’s focus on a rough running time analysis.

Binary Search

- We start with an array of size n.

- At each step, we
    - do constant work (compare midpoint of A to x)
    - cut the problem size in half
    - recur on the appropriate half

Binary Search

- We start with an array of size n. **T(n)**
- At each step, we
    - do constant work (compare midpoint of A to x) **c**
    - cut the problem size in half
    - recur on the appropriate half **T(n/2)**

Binary Search Recurrence

- **T(n) = T(n/2) + c**
- What's the base case?
- If not specified, assume it is *some* constant for T(1)
- *Which* constant doesn't affect asymptotic solution
  → pick for convenience
- *(More on this in Studio 4)*

Guessing Running Time

- Is $T(n)$ constant-time?
- Let's guess $T(n) = c$
- Pick $T(1) = c$ to make base case match [constant for $T(1)$ doesn't matter!]
- $T(n) = T(n/2) + c$ = ??? [what does substitution yield?]

Guessing Running Time

- Is T(n) constant-time?
- Let's guess $T(n) = c$
- Pick $T(1) = c$ to make base case match [constant for T(1) doesn't matter!]
- $T(n) = T(n/2) + c = c + c = 2c$
- But we are trying to prove that $T(n) = c$, so proof failed!

Guessing Running Time

- Is T(n) constant-time?
- Let’s guess T(n) = c
- Pick T(1) = c to make base case match [constant for T(1) doesn’t matter!]
- T(n) = T(n/2) + c = c + c = 2c
- But we are trying to prove that T(n) = c, so proof failed!
- *(And indeed, can see that no other constant > 0 would work either)*

Guessing Running Time, Try #2

- Is T(n) linear-time?
- Let’s guess $T(n) = cn$
- Pick $T(1) = c$ to make base case match [constant for T(1) doesn’t matter!]
- $T(n) = T(n/2) + c =$???

Guessing Running Time, Try #2

- Is $T(n)$ linear-time?
- Let's guess $T(n) = cn$
- Pick $T(1) = c$ to make base case match [constant for $T(1)$ doesn't matter!]
- $T(n) = T(n/2) + c = cn/2 + c$ ← not cn as desired! Proof fails, *but*…

Guessing Running Time, Try #2

- Is $T(n)$ linear-time?
- Let's guess $T(n) = cn$
- Pick $T(1) = c$ to make base case match [constant for $T(1)$ doesn't matter!]
- $T(n) = T(n/2) + c = cn/2 + c = c(n/2 + 1) \leq cn$ *for* $n \geq 2$

Guessing Running Time, Try #2

- Is $T(n)$ linear-time?
- Let's guess $T(n) = cn$
- Pick $T(1) = c$ to make base case match [constant for $T(1)$ doesn't matter!]
- $T(n) = T(n/2) + c = cn/2 + c = c(n/2 + 1) \leq cn$ *for* $n \geq 2$
- ***Conclude that*** $T(n) \leq cn$ ***for all*** n.
- ***Therefore,*** $T(n) = ???$ ***[asymptotically]***

Guessing Running Time, Try #2

- Is T(n) linear-time?
- Let's guess T(n) = cn
- Pick T(1) = c to make base case match [constant for T(1) doesn't matter!]
- T(n) = T(n/2) + c = cn/2 + c = *c(n/2 + 1) ≤ cn for n ≥ 2*
- ***Conclude that T(n) ≤ cn for all n.***
- ***Therefore, T(n) = O(n)*** ***← proving ≤ implies upper bound***

Guessing Running Time, Try #2

- Is T(n) linear-time?
- Let's guess T(n) = cn
- Pick T(n't matter!]
- T(n) =
- ***Conclu***
- ***Therefore, T(n) = O(n)*** ***←-proving ≤ implies upper bound***

We need to prove either an equality (as before) or *both* an upper *and* a lower bound to prove Θ.

Guessing Running Time, Try #3

- Is T(n) logarithmic-time?
- Let’s guess $T(n) = c \log_2 n$
- If $T(1) = c$… $c \log_2 1 = 0 \neq c$. Whoops.

Guessing Running Time, Try #3

- Is T(n) logarithmic-time?
- Let's guess $T(n) = c \log_2 n$
- $T(1) = c \log_2 1 = 0 \neq c$. Whoops.

Guessing Running Time, Try #3

- Is T(n) logarithmic-time?
- Let's guess $T(n) = c \log_2 n$
- $T(2) = c \log_2 2 = c$. So induction will start at $n = 2$ (fine for asymptotic!)
- $T(n) = T(n/2) + c$

$$
\begin{aligned}
&= c \log_2(n/2) + c \\
&= c(\log_2 n - \log_2 2) + c \\
&= c \log_2 n - c + c \\
&= c \log_2 n \qquad \leftarrow \text{Yay, it worked! So } T(n) = \Theta(\log n)
\end{aligned}
$$

Guessing Running Time, Try #3

- Is T(n) lo
- Let’s g
- T(2) = tic!)
- T(n) =

  =

  =

  = c

  = $c \log_2 n$ ← Yay, it worked! So $T(n) = \Theta(\log n)$

EXERCISE

Apply the same logic to show that

$T(n) = T(2n/3) + k$

has solution $T(n) = \Theta(\log n)$

Guessing Running Time, Try #3

- Is T(n) lo
- Let’s g
- T(2) = … tic!)
- T(n) =

  =

  =

  = c

  = $c \log_2 n$ ← Yay, it worked! So $T(n) = \Theta(\log n)$

EXERCISE

Apply the same logic to show that

$T(n) = T(2n/3) + k$

has solution $T(n) = \Theta(\log n)$

[hint: try a guess involving $\log_{3/2} n$]

Pros and Cons of Guess and Check

- \+ For **any recurrence**, given right guess, can prove that it is correct.
- \+ Can use separate upper-, lower-bound proofs to prove Θ result.

Pros and Cons of Guess and Check

- \+ For **any recurrence**, given right guess, can prove that it is correct.
- \+ Can use separate upper-, lower-bound proofs to prove Θ result.

- \- **You must start from a correct guess**
- \- Guessing the right constants and lower-order terms to make the induction work can be quite challenging

Pros and Cons of Guess and Check

+ For **any recurrence**, given right guess, can prove that it is correct.
+ Can use separate upper-, lower-bound proofs to prove Θ result.

- **You must start from a correct guess**
- Guessing the right constants and lower-order terms to make the induction work can be quite challenging

Can we take the guess-work out of solving recurrences?

Pros and Cons of G ck

- \+ For **any recurrenc** that it is correct.
- \+ Can use separa e Θ result.

- \- **You must star**
- \- Guessing the r make the induction work

Can we take the g ng recurrences?

In general, no.

(It’s a bit like finding values for *c* and *n_0* in a Big-Oh proof)

Pros and Cons of G ck

- + For **any recurrenc** that it is correct.
- + Can use separa e Θ result.

- - **You must star**
- - Guessing the r make the induction work

In general, no.

But for certain common cases, there’s a way.

Can we take the g ng recurrences?

A Very Common Case

- You have an algorithm FOO that runs on an input of size n.
- FOO does some *local* work.
- FOO makes some recursive calls on inputs whose size is a fraction of n.

```
FOO(A[1..n])
    FOO(A[1..n/2])
    Print(A)
    FOO(A[n/2+1..n])
```

A Very Common Case

- FOO takes time **T(n) on input of size n**.
- FOO does some *local* work taking time **f(n)**.
- FOO makes **a** recursive calls on inputs of size **n/b.**

```
FOO(A[1..n])
    FOO(A[1..n/2])
    Print(A)
    FOO(A[n/2+1..n])
```

A Very Common Case

- FOO takes time **T(n) on input of size n**.
- FOO does some *local* work taking time **f(n)**.
- FOO makes **a** recursive calls on inputs of size **n/b.**

a = 2   **b = 2**

f(n) = cn

```
FOO(A[1..n])
    FOO(A[1..n/2])
    Print(A)
    FOO(A[n/2+1..n])
```

A Very Common Case

- FOO takes time **T(n) on input of size n**.
- FOO does some *local* work taking time **f(n)**.
- FOO makes **a** recursive calls on inputs of size **n/b.**

$$T(n) = aT(n/b) + f(n)$$

A Very Common Case

- FOO takes time **T(n) on input of size n**.
- FOO does some *local* work taking time **f(n)**.
- FOO makes **a** recursive calls on inputs of size **n/b.**

$$T(n) = aT(n/b) + f(n)$$

Assumes T(n) = constant for small enough n – see Studio 4 for more on this

Examples That Fit the Paradigm

- Binary search: $T(n) = T(n/2) + c$

- Merge sort: $T(n) = 2T(n/2) + cn$

- Strassen’s matrix multiply: $T(n) = 7T(n/2) + cn^2$

- Maximum subarray: $T(n) = 2T(n/2) + c$

New Strategy

- We could, in principle, expand the recurrence to a sum of terms (as we sketched for heapify) and add them up
- E.g., $T(n) = T(n/2) + c = (T(n/4) + c) + c = ((T(n/8) + c) + c) + c = \ldots$

$$= \underbrace{c + c + c + \ldots + c}_{\textbf{How many times?}}$$

New Strategy

- We could, in principle, expand the recurrence to a sum of terms (as we sketched for heapify) and add them up

- E.g., $T(n) = T(n/2) + c = (T(n/4) + c) + c = ((T(n/8) + c) + c) + c = \ldots$

$$= \underbrace{c + c + c + \ldots + c}_{\textbf{About } \log_2 n \textbf{ times}}$$

$$= \boldsymbol{\Theta(\log n)}$$

New Strategy

- We co erms (as we
- E.g., T c = …

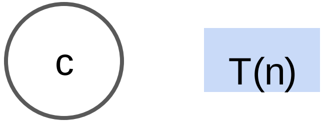
We'll develop a way to compute this sum for any recurrence of form **$T(n) = aT(n/b) + f(n)$**

Idea: Draw a Picture!

- We’ll draw a tree showing all the terms in the recurrence.
- It’s called a recursion tree.
- Each node records work of one term in expansion of recurrence.
- Add up work over all nodes to get total work.

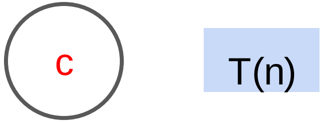
Example: T(n) = T(n/2) + c

- Root contains first term of expansion



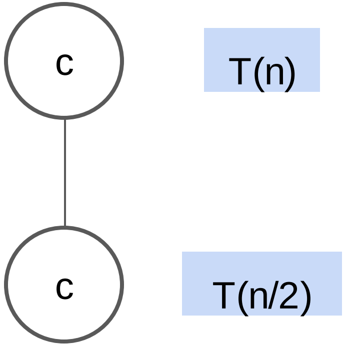
Example: $T(n) = T(n/2) + c$

- Root contains first term of expansion



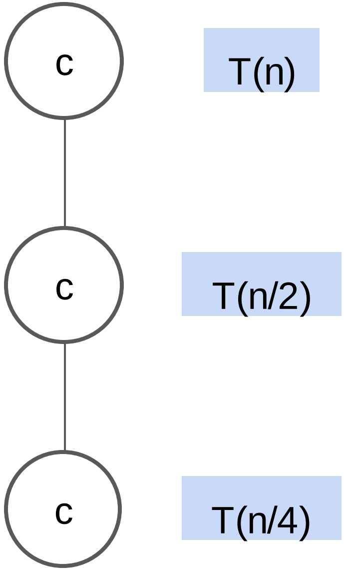
Example: T(n) = T(n/2) + c

- Expand once to get second term



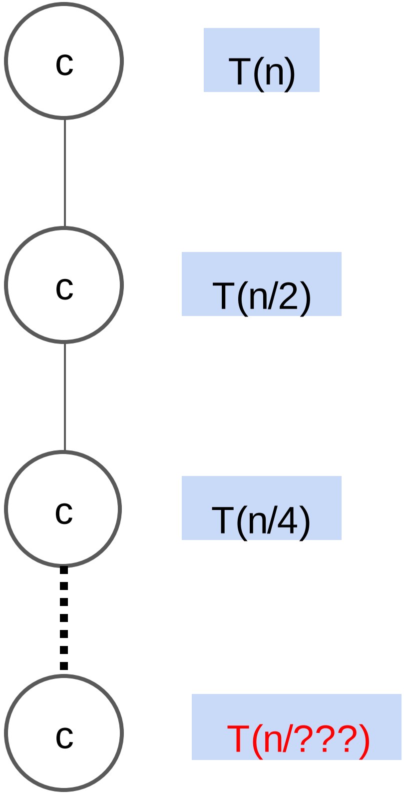
Example: T(n) = T(n/2) + c

- Now repeat…



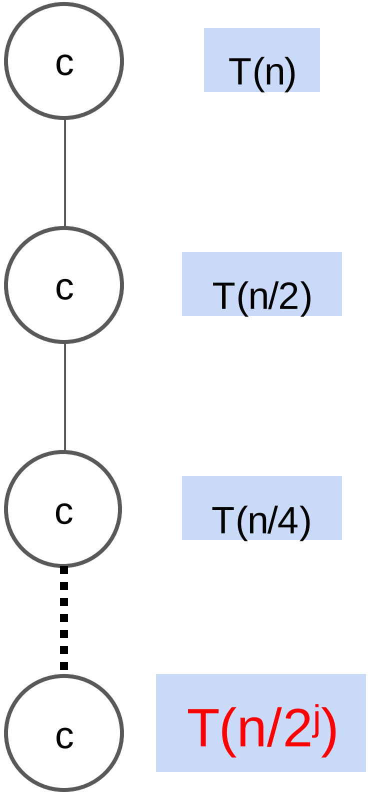
Example: T(n) = T(n/2) + c

- What's the *generic term* (after j steps)?



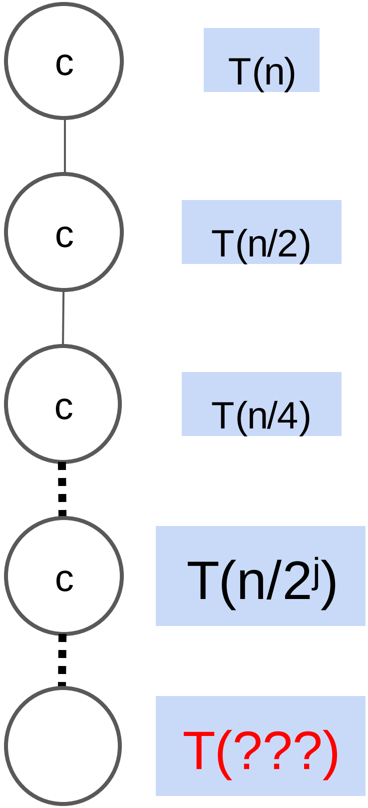
Example: T(n) = T(n/2) + c

- What's the *generic term* (after j steps)?
- We divide by 2, j times, hence **$T(n/2^j)$**
- Each term is still just "c"



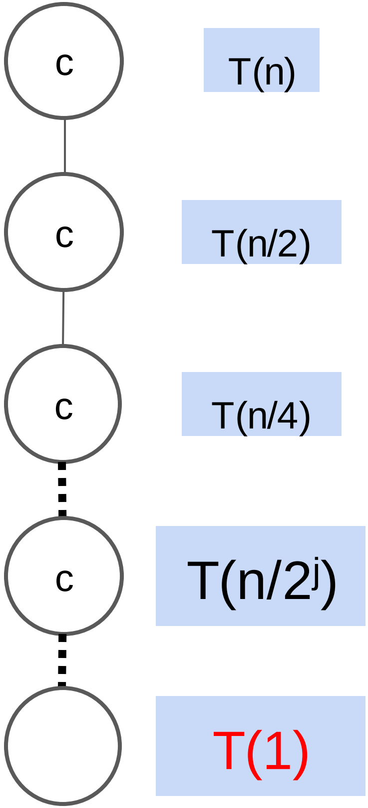
Example: T(n) = T(n/2) + c

- What’s *the last term*?
- (Assume n is power of 2)
- What is the last term?



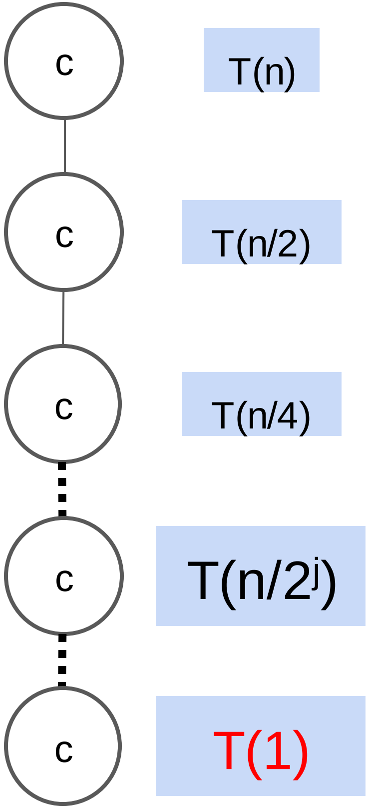
Example: $T(n) = T(n/2) + c$

- What’s *the last term?*
- (Assume n is power of 2)
- We stop at T(1)
- What is T(1)?



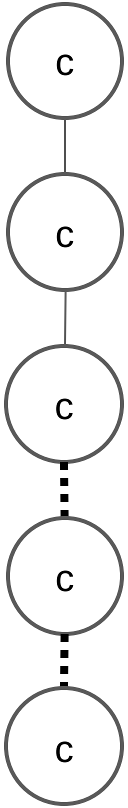
Example: T(n) = T(n/2) + c

- What's *the last term*?
- (Assume n is power of 2)
- We stop at T(1)
- What is T(1)?
- We assumed T(1) = c



Accounting

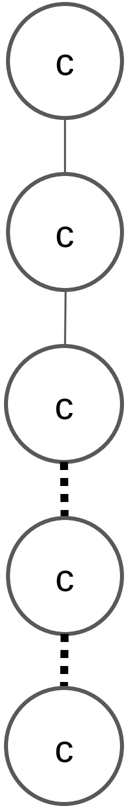
$T(n) = T(n/2) + c$



Depth	Problem Size	Local Work
0	n	c
1	n/2	c
2	n/4	c
j	$n/2^j$	c
???	1	c

Accounting

$T(n) = T(n/2) + c$

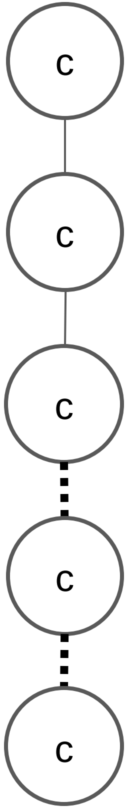


Depth	Problem Size	Local Work
0	n	c
1	$n/2$	c
2	$n/4$	c
j	$n/2^j$	c
$\log_2 n$	1	c

Max depth = # of divisions by 2 needed to get from n down to 1.

Accounting

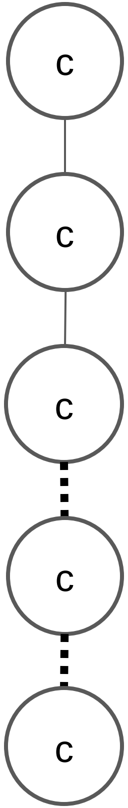
$$T(n) = T(n/2) + c$$



Depth	Problem Size	Local Work
0	n	c
1	$n/2$	c
2	$n/4$	c
j	$n/2^j$	c
$\log_2 n$	1	c

Total work is sum of local work in each row

Accounting
$T(n) = T(n/2) + c$

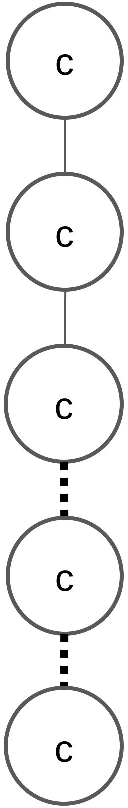


Depth	Problem Size	Local Work
0	n	c
1	n/2	c
2	n/4	c
j	$n/2^j$	c
$\log_2 n$	1	c

$$\sum_{j=0}^{\log_2 n} c$$

Accounting

$T(n) = T(n/2) + c$



Depth	Problem Size	Local Work
0	n	c
1	$n/2$	c
2	$n/4$	c
j	$n/2^j$	c
$\log_2 n$	1	c

$$\sum_{j=0}^{\log_2 n} c$$

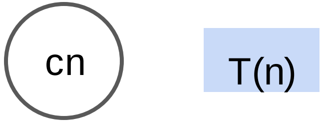
$$= c\,(\log_2 n + 1)$$
$$= \Theta(\log n)$$

Recursion Tree Methodology

- Given recurrence…
- Sketch the tree (figure out its height!)
- Figure out problem size and local work/node at each level
- Sum local work over whole tree

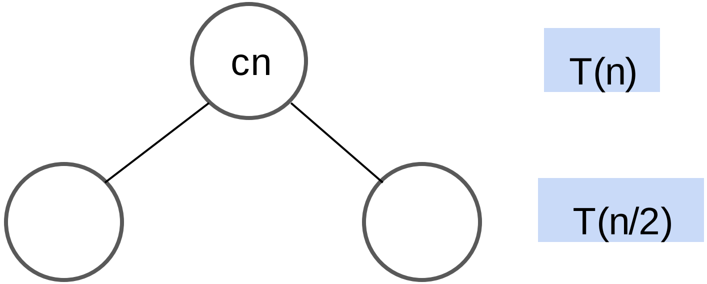
Example: $T(n) = 2T(n/2) + cn$ [Assume $T(1) = d$]

- Root contains first term of expansion



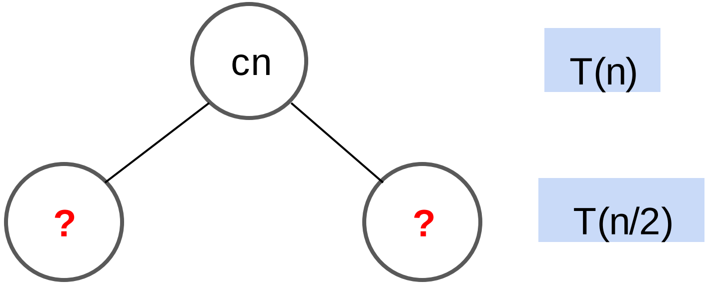
Example: T(n) = 2T(n/2) + cn [Assume T(1) = d]

- There are two subproblems at next level



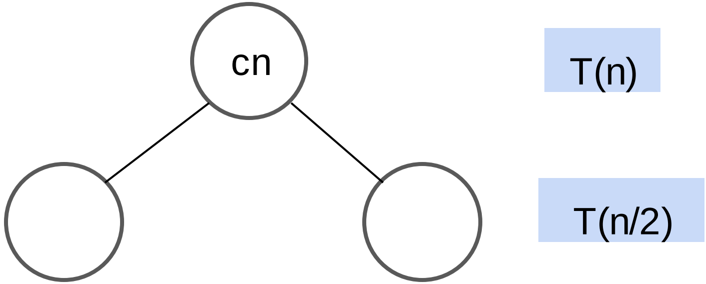
Example: T(n) = 2T(n/2) + cn [Assume T(1) = d]

- How much work in each node?



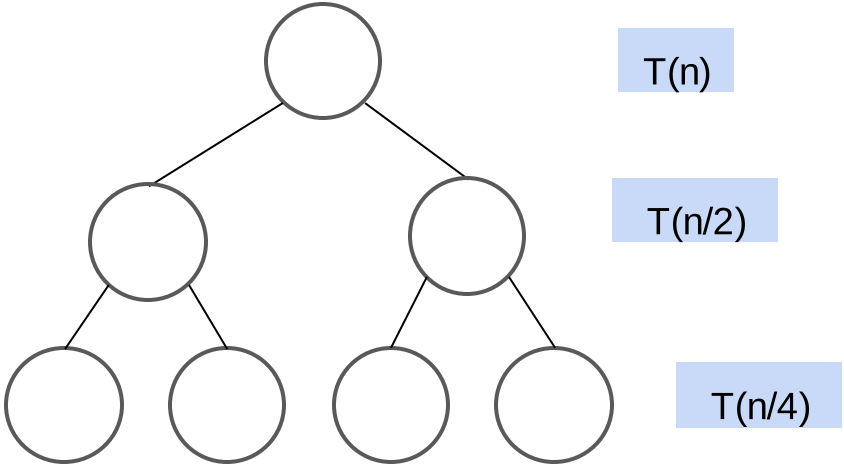
Example: T(n) = 2T(n/2) + cn   [Assume T(1) = d]

- How much work in each node?
- cn/2 [but it doesn't fit in the circles]
- Let's just draw the tree…



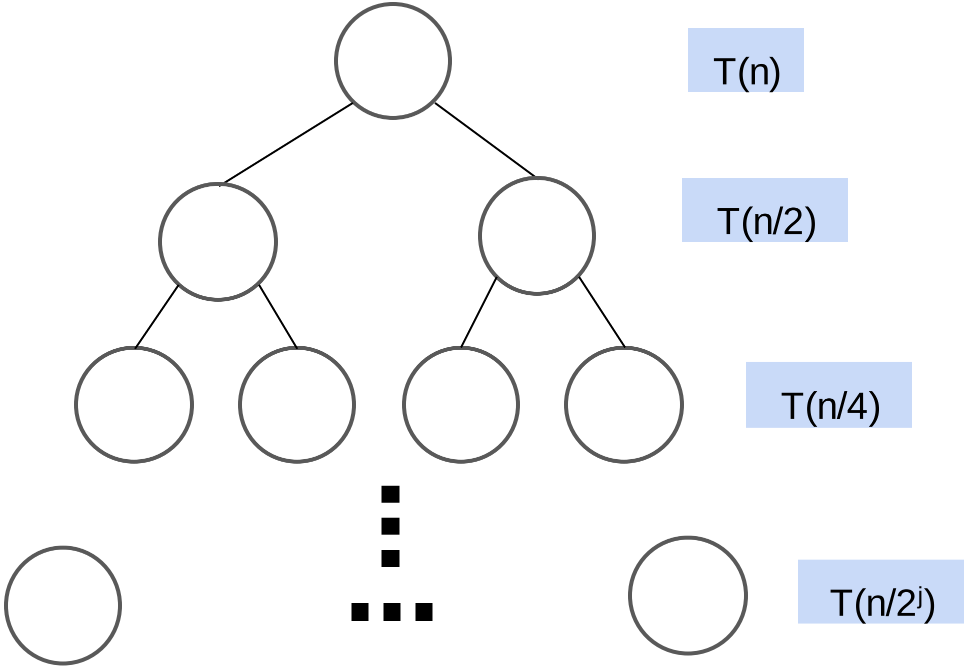
Example: $T(n) = 2T(n/2) + cn$ [Assume $T(1) = d$]

- Let's just draw the tree…
- # of nodes doubles at each level ($a = 2$)



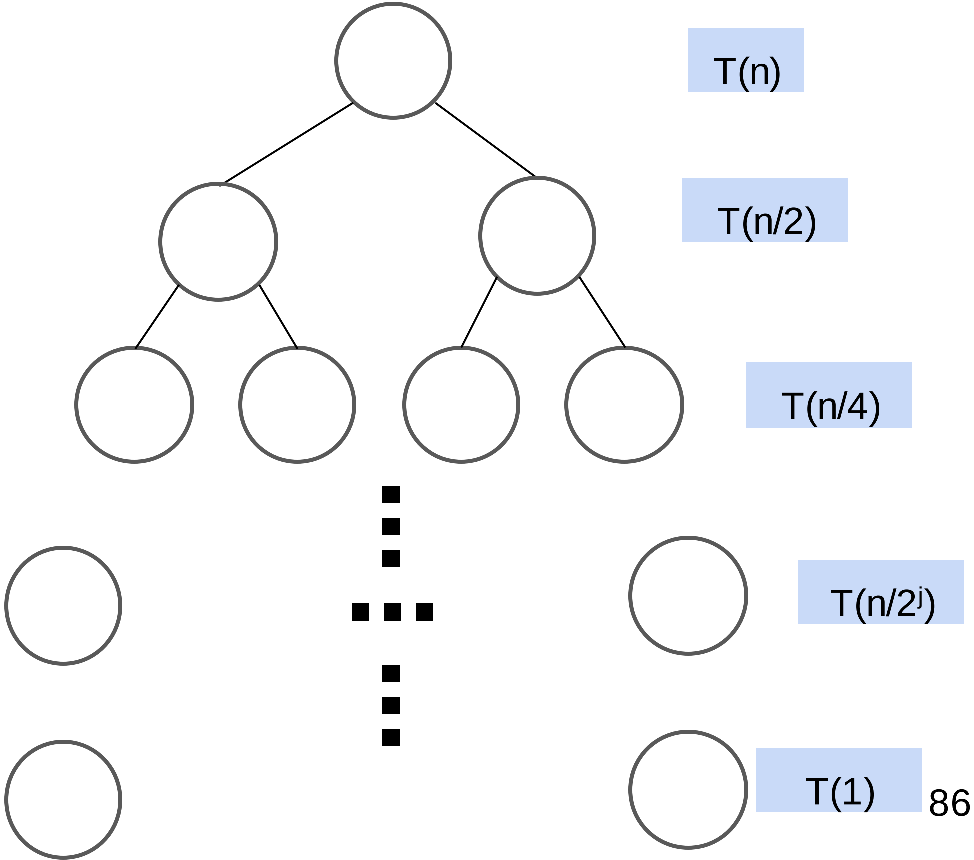
Example: T(n) = 2T(n/2) + cn [Assume T(1) = d]

- Let's just draw the tree…
- # of nodes doubles at each level (a = 2)
- After j steps, we have 2^j nodes at level j



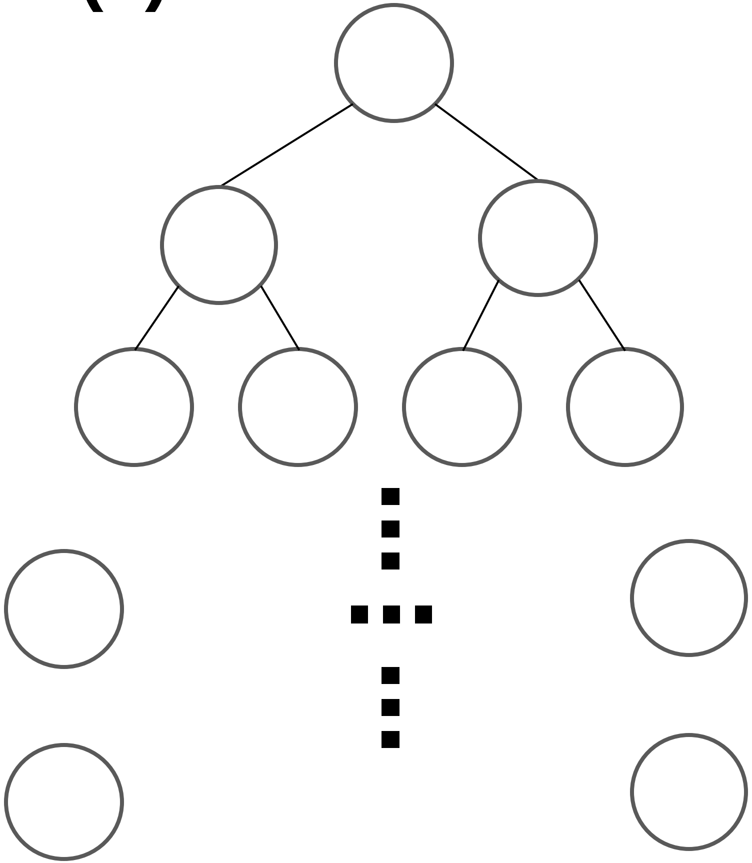
Example: $T(n) = 2T(n/2) + cn$ [Assume $T(1) = d$]

- Let’s just draw the tree…
- # of nodes doubles at each level ($a = 2$)
- After j steps, we have 2^j nodes at level j
- Bottom out at $T(1)$ again



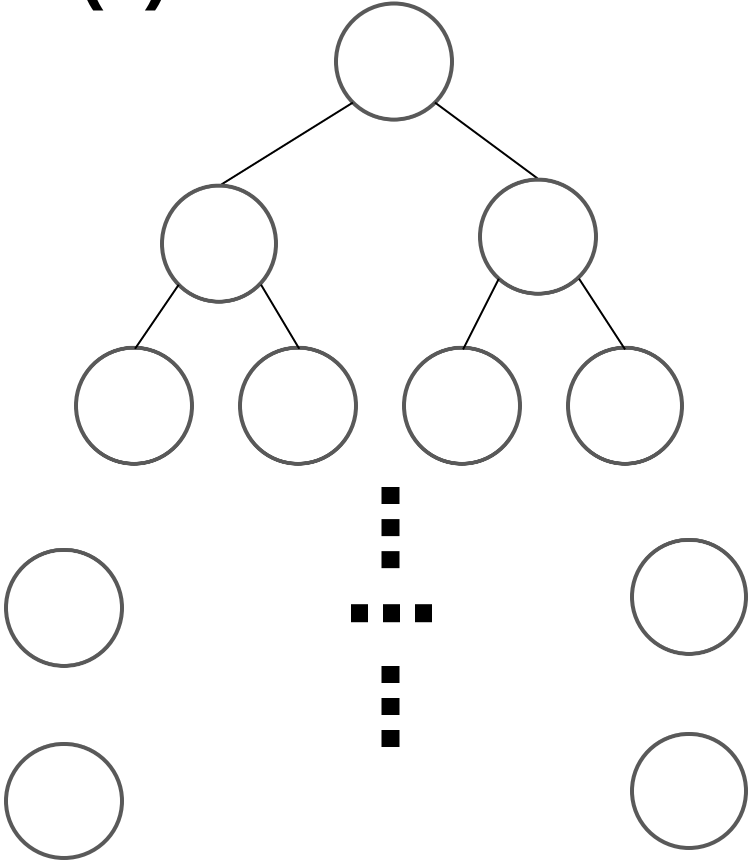
$T(n) = 2T(n/2) + cn$

$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	n/2	2	
2	n/4	4	
j	$n/2^j$	???	
$\log_2 n$	1		

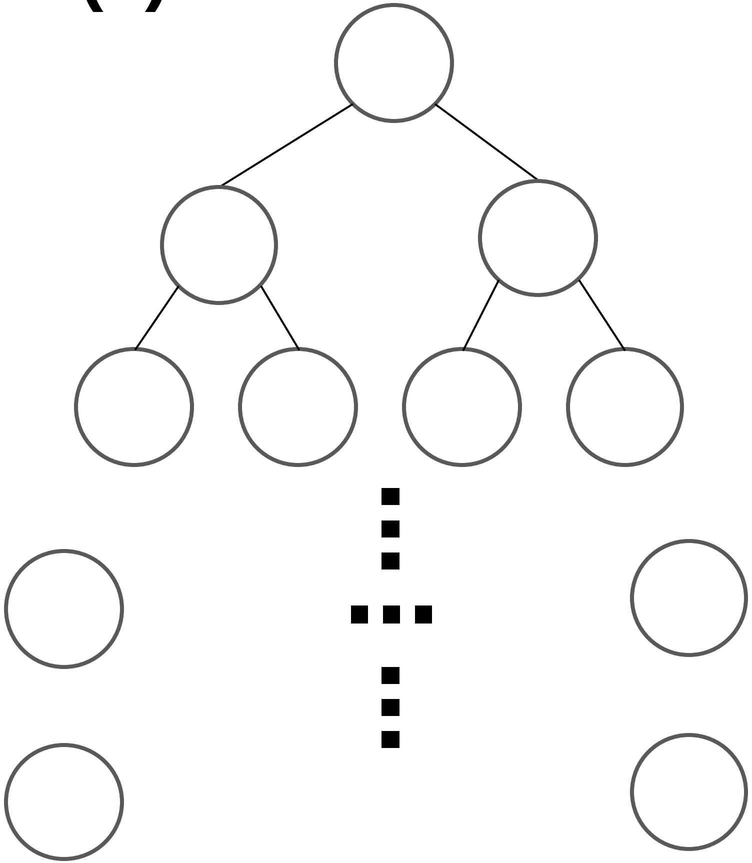
$T(n) = 2T(n/2) + cn$
$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	n/2	2	
2	n/4	4	
j	$n/2^j$	2^j	
$\log_2 n$	1	???	

$T(n) = 2T(n/2) + cn$

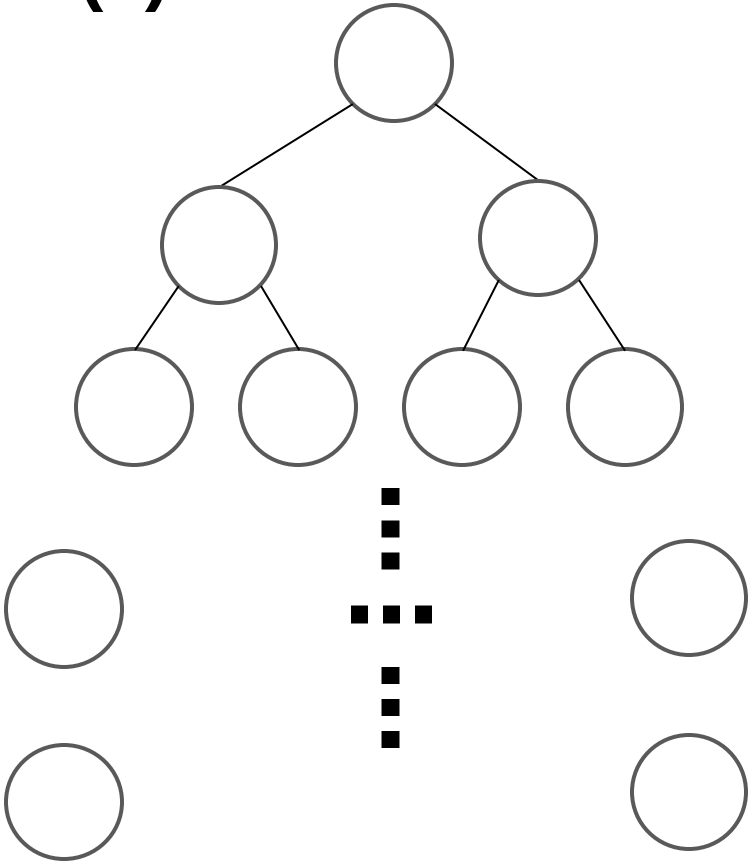
$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	n/2	2	
2	n/4	4	
j	$n/2^j$	2^j	
$\log_2 n$	1	$2^{\log_2 n}$	

$$T(n) = 2T(n/2) + cn$$

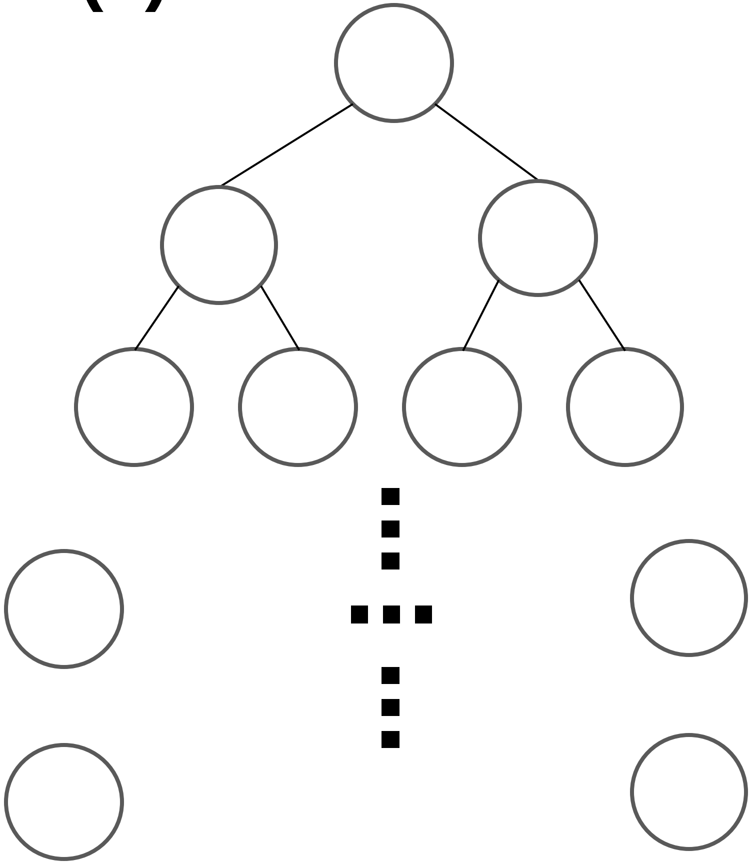
$$T(1) = d$$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	$n/2$	2	
2	$n/4$	4	
j	$n/2^j$	2^j	
$\log_2 n$	1	**n**	

$T(n) = 2T(n/2) + cn$

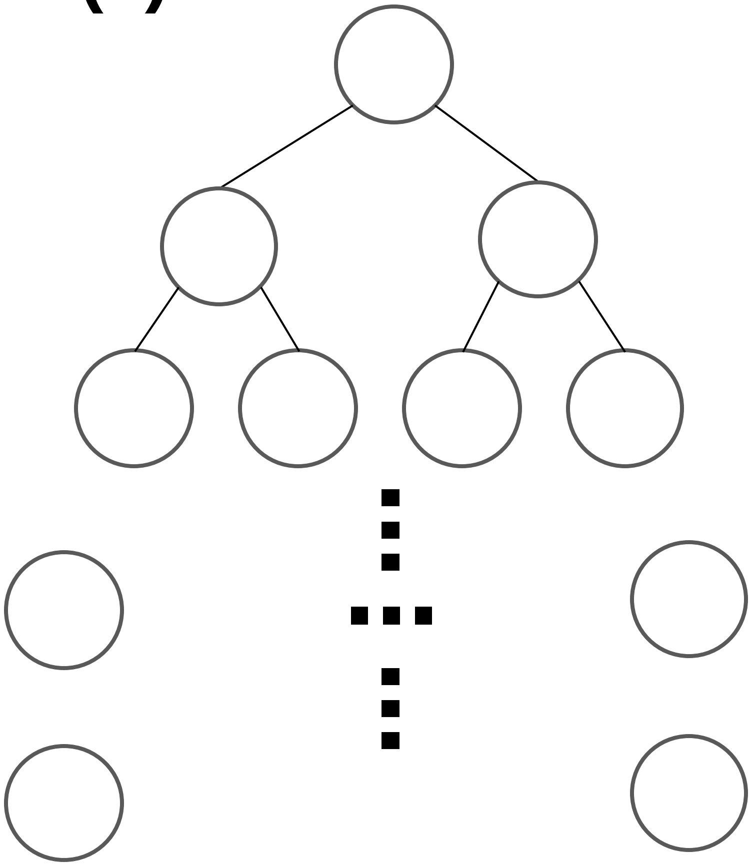
$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	cn
1	n/2	2	cn/2
2	n/4	4	???
j	$n/2^j$	2^j	
$\log_2 n$	1	n	

$$T(n) = 2T(n/2) + cn$$

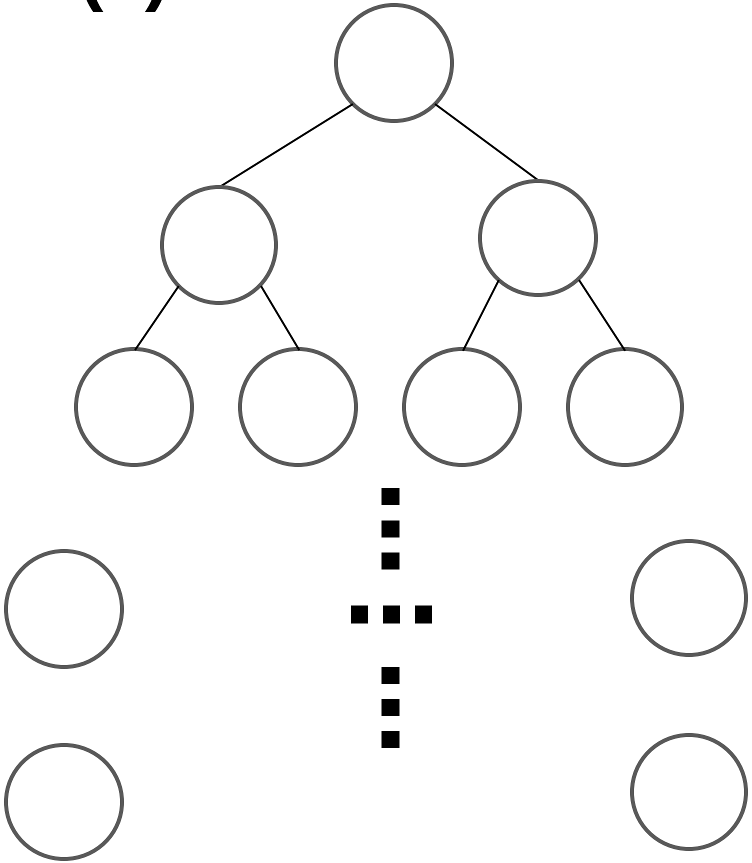
$$T(1) = d$$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	cn
1	n/2	2	cn/2
2	n/4	4	cn/4
j	$n/2^j$	2^j	???
$\log_2 n$	1	n	

$T(n) = 2T(n/2) + cn$

$T(1) = d$

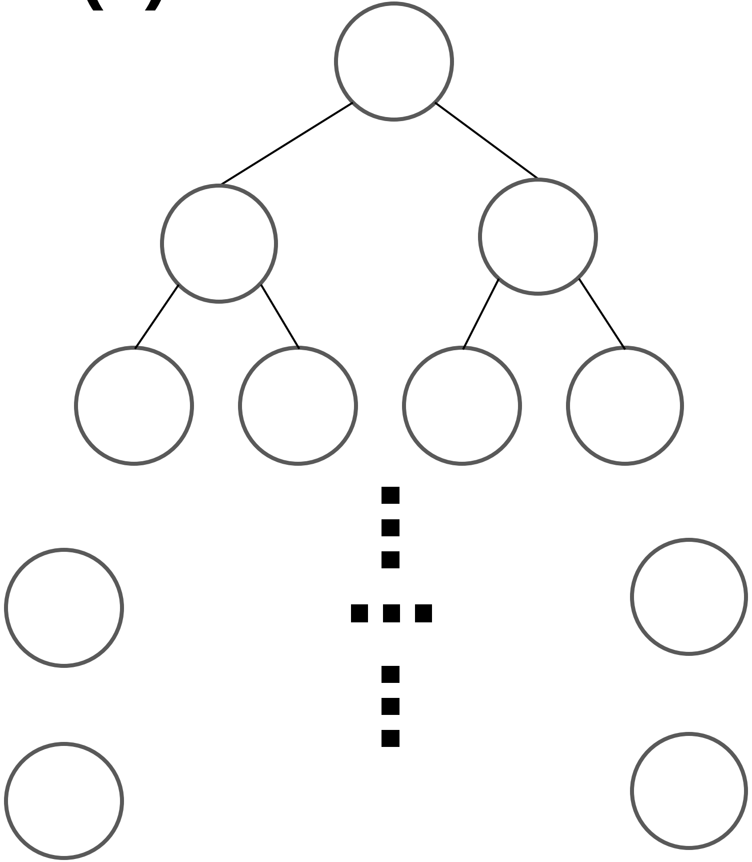


Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	cn
1	n/2	2	cn/2
2	n/4	4	cn/4
j	$n/2^j$	2^j	$cn/2^j$
log			

Substitute problem size into f(n)

$T(n) = 2T(n/2) + cn$

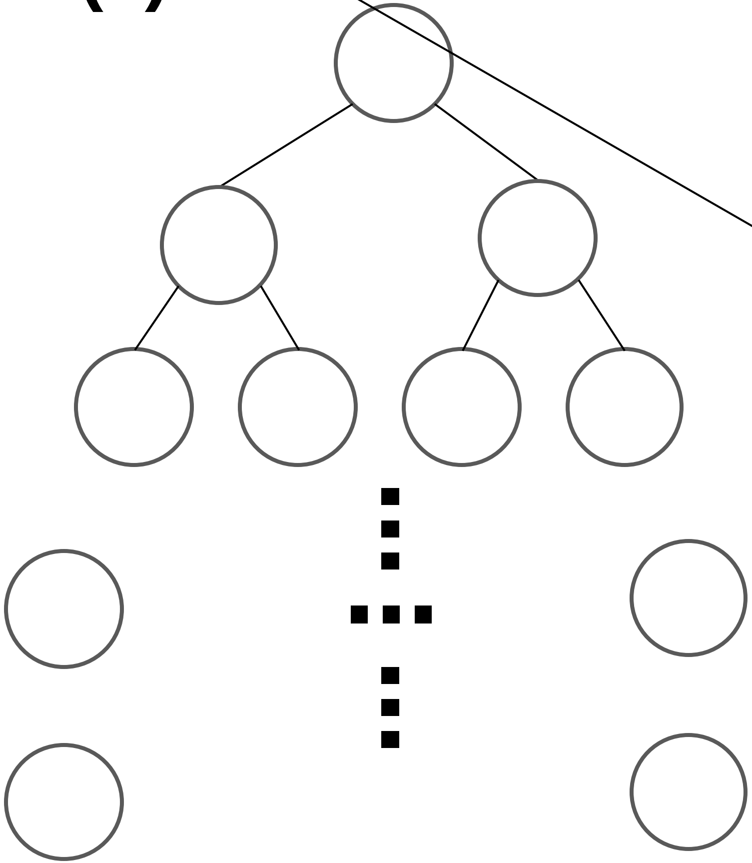
$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	cn
1	n/2	2	cn/2
2	n/4	4	cn/4
j	$n/2^j$	2^j	$cn/2^j$
$\log_2 n$	1	n	???

$T(n) = 2T(n/2) + cn$

$T(1) = d$



Depth	**Problem Size**	**# Nodes Per Level**	**Local Work per Node**
0	n	1	cn
1	n/2	2	cn/2
2	n/4	4	cn/4
j	$n/2^j$	2^j	$cn/2^j$
$\log_2 n$	1	n	**d**

$T(n) = 2T(n/2) + cn$

Depth	Problem Size	# Nodes Per Level	Local Work per Node	Local Work per Level
0	n	1	cn	1 x cn
1	n/2	2	cn/2	2 x cn/2
2	n/4	4	cn/4	4 x cn/4
j	$n/2^j$	2^j	$cn/2^j$	$2^j \times cn/2^j$
$\log_2 n$	1	n	d	n x d

Multiply across each level to get its work

$T(n) = 2T(n/2) + cn$

Depth	**Problem Size**	**# Nodes Per Level**	**Local Work per Node**	**Local Work per Level**
0	n	1	cn	cn
1	n/2	2	cn/2	cn
2	n/4	4	cn/4	cn
j	$n/2^j$	2^j	$cn/2^j$	cn
$\log_2 n$	1	n	d	dn

(Simplification isn't always this nice)

T(n) = 2T(n/2) + cn

Depth	Problem Size	# Nodes Per Level	Local Work per Node	Local Work per Level
0	n	1	cn	cn
1	n/2	2	cn/2	cn
2	n/4	4	cn/4	cn
j	$n/2^j$	2^j	$cn/2^j$	cn
$\log_2 n$	1	n	d	dn

$$dn + \sum_{j=0}^{\log_2 n - 1} cn$$

$T(n) = 2T(n/2) + cn$

Depth	Problem Size	# Nodes Per Level	Local Work per Node	Local Work per Level
0	n	1	cn	cn
1	$n/2$	2	$cn/2$	cn
2	$n/4$	4	$cn/4$	cn
j	$n/2^j$	2^j	$cn/2^j$	cn
$\log_2 n$	1	n	d	dn

$$dn + \sum_{j=0}^{\log_2 n - 1} cn$$

$= dn + cn \log_2 n$

$= \Theta(n \log n)$

Recursion Tree Methodology (Again)

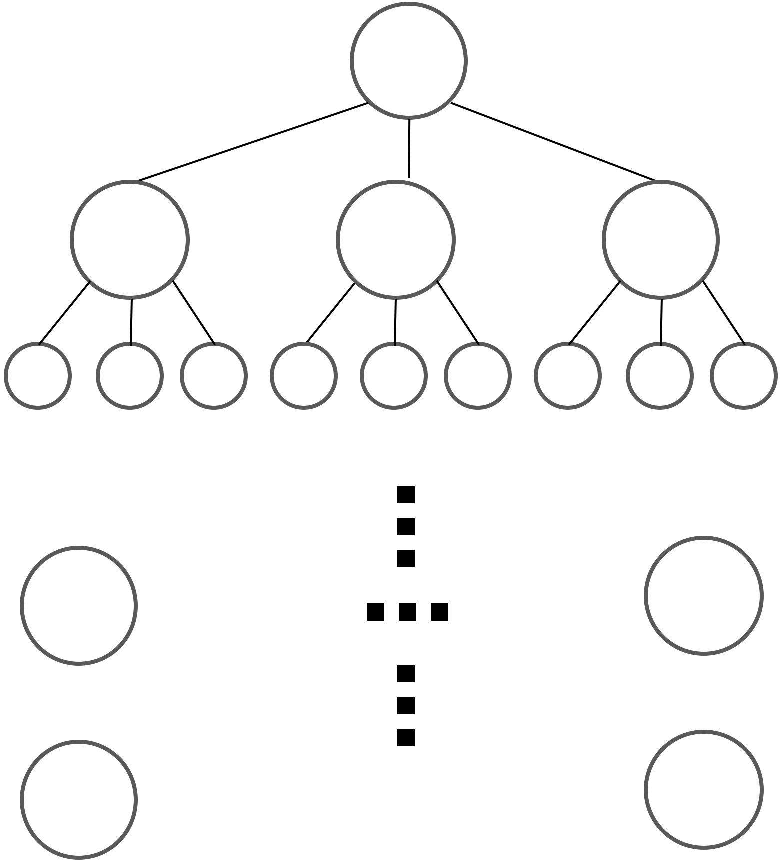
- Given recurrence…
- Sketch the tree (figure out its height!)
- Figure out problem size, # nodes, and local work/node at each level
- Sum local work at each level, then across levels

Example: $T(n) = 3T(n/4) + cn^2$ $[T(1) = d]$

- [This one is worked in your text as well – see p. 89]

Example: $T(n) = 3T(n/4) + cn^2$ $[T(1) = d]$

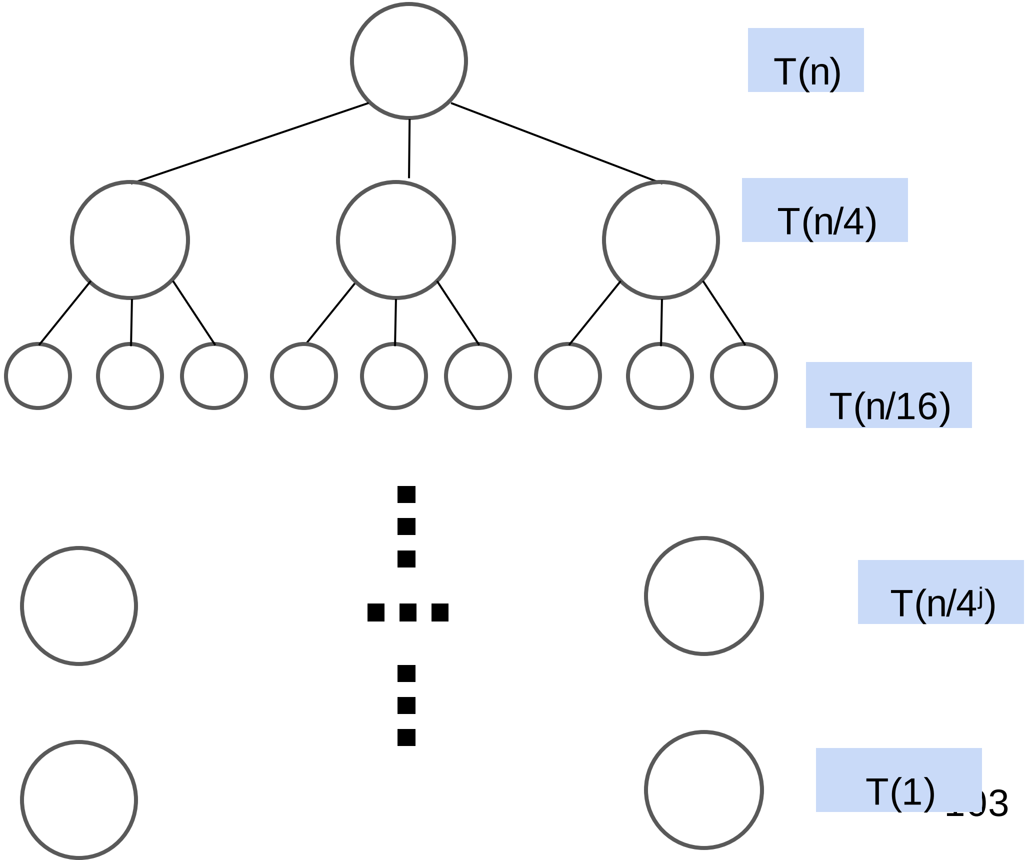
- This time, a = 3, so each node branches 3 ways!



Example: $T(n) = 3T(n/4) + cn^2$ $[T(1) = d]$

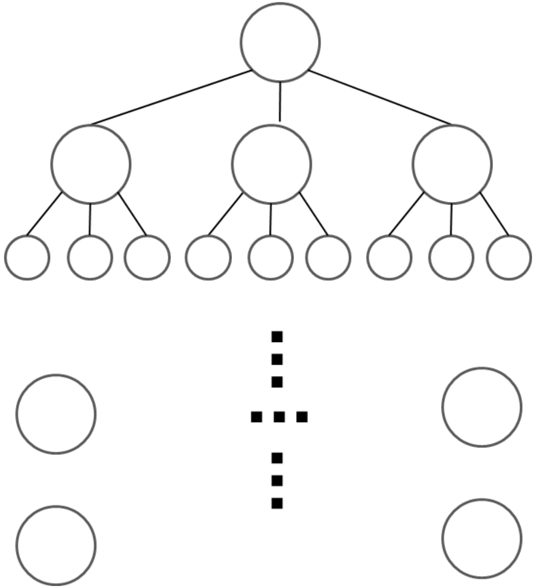
- This time, a = 3, so each node branches 3 ways!

- This time, b = 4, so problem size goes down by factor of 4 per level.



$T(n) = 3T(n/4) + cn^2$

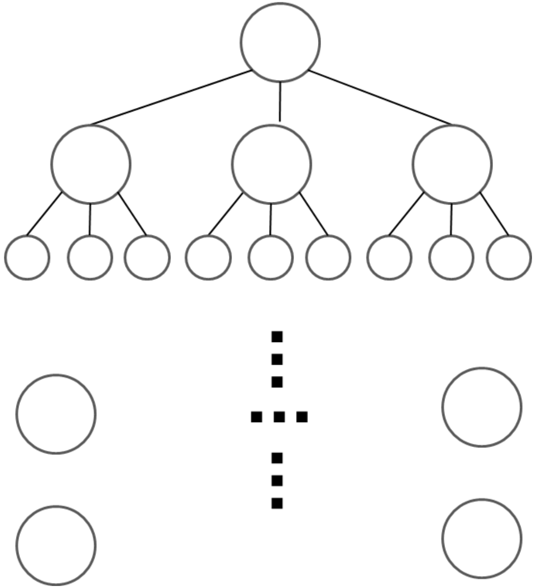
$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n		
1	n/4		
2	n/16		
j	$n/4^j$		
???	1		

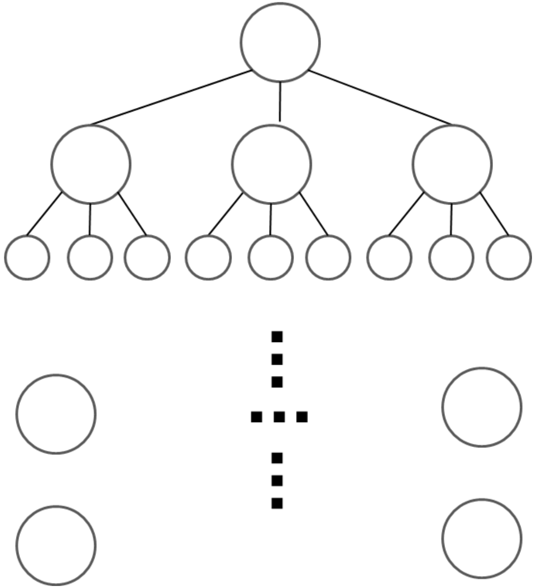
$T(n) = 3T(n/4) + cn^2$

$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	n/4	3	
2	n/16	???	
j	$n/4^j$		
$\log_4 n$	1		

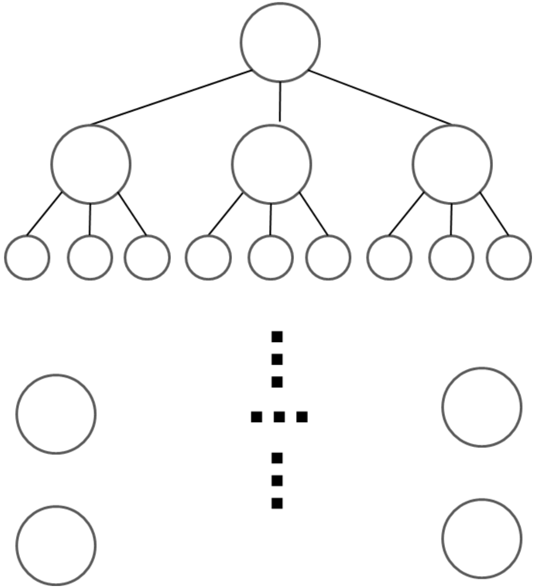
$$T(n) = 3T(n/4) + cn^2$$
$$T(1) = d$$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	$n/4$	3	
2	$n/16$	9	
j	$n/4^j$	???	
$\log_4 n$	1		

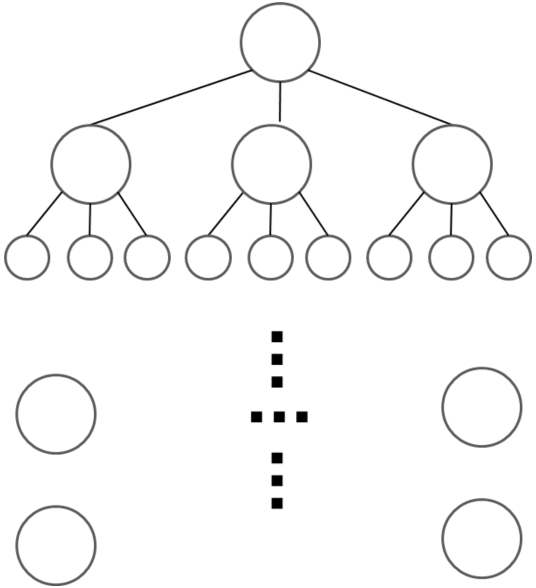
$T(n) = 3T(n/4) + cn^2$

$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	n/4	3	
2	n/16	9	
j	$n/4^j$	3^j	
$\log_4 n$	1	???	

$T(n) = 3T(n/4) + cn^2$
$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	n/4	3	
2	n/16	9	
j	$n/4^j$	3^j	
$\log_4 n$	1	$3^{\log_4 n}$	

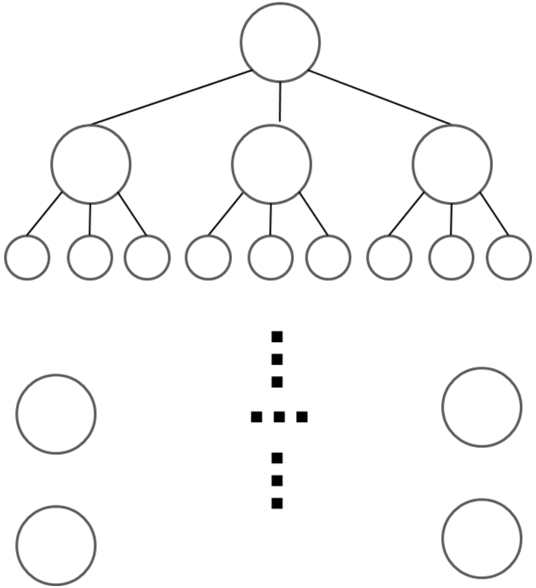
A Brief Diversion

$a^{\log_b n} = a^{\log_a n \log_b a}$ by change of base $\log_b n = \log_a n \log_b a$

$= n^{\log_b a}$

$$\boldsymbol{a^{\log_b n} = n^{\log_b a}}$$

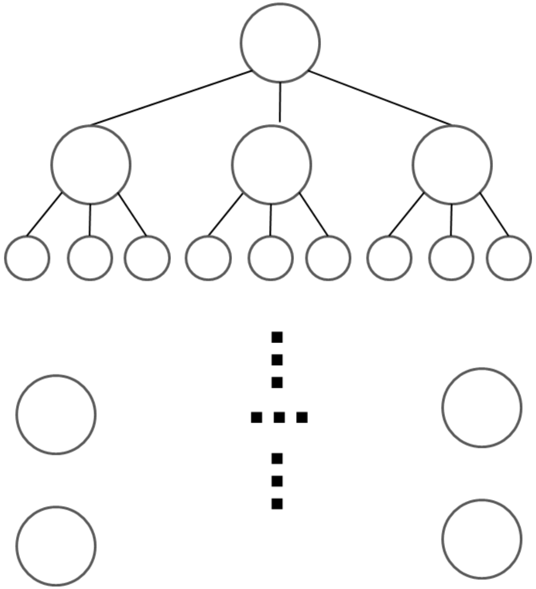
$T(n) = 3T(n/4) + cn^2$
$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	n/4	3	
2	n/16	9	
j	$n/4^j$	3^j	
$\log_4 n$	1	$n^{\log_4 3}$	

$$T(n) = 3T(n/4) + cn^2$$

$$T(1) = d$$

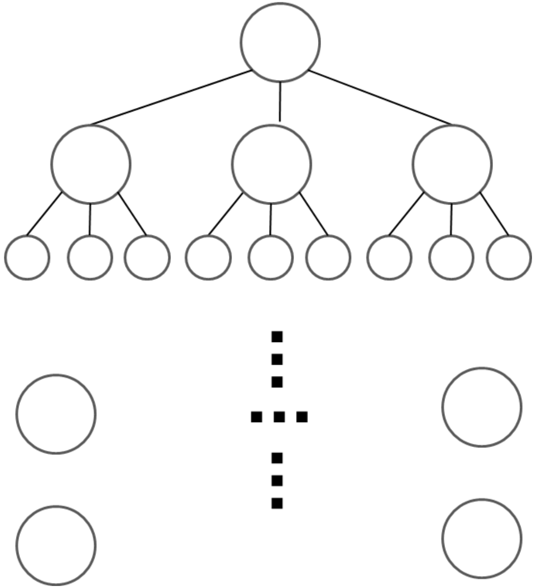


Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	
1	n/4	3	
2	n/16	9	
j	$n/4^j$	3^j	
		$n^{\log_4 3}$	

Please simplify to this form! →

$T(n) = 3T(n/4) + cn^2$

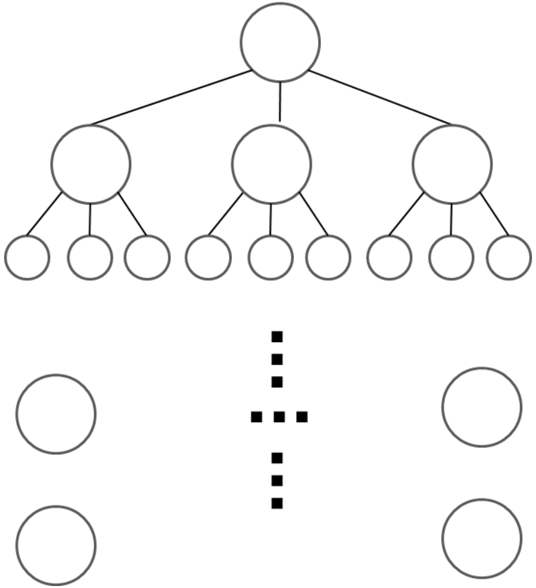
$T(1) = d$



Depth	**Problem Size**	**# Nodes Per Level**	**Local Work per Node**
0	n	1	cn^2
1	n/4	3	???
2	n/16	9	
j	$n/4^j$	3^j	
$\log_4 n$	1	$n^{\log_4 3}$	

$T(n) = 3T(n/4) + cn^2$

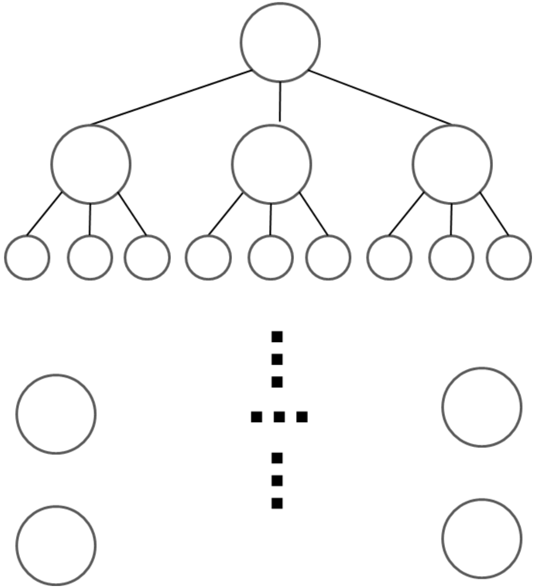
$T(1) = d$



Depth	**Problem Size**	**# Nodes Per Level**	**Local Work per Node**
0	n	1	cn^2
1	n/4	3	$c(n/4)^2$
2	n/16	9	
j	$n/4^j$	3^j	
$\log_4 n$	1	$n^{\log_4 3}$	

$T(n) = 3T(n/4) + cn^2$

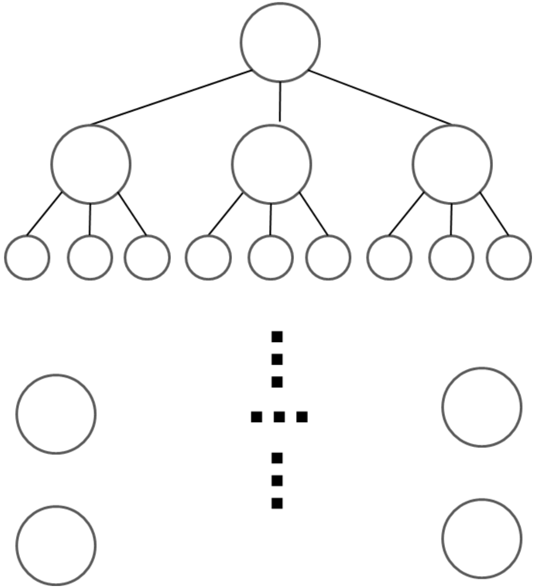
$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	cn^2
1	n/4	3	$c(n/4)^2$
2	n/16	9	$c(n/16)^2$
j	$n/4^j$	3^j	$c(n/4^j)^2$
$\log_4 n$	1	$n^{\log_4 3}$	???

$T(n) = 3T(n/4) + cn^2$

$T(1) = d$



Depth	Problem Size	# Nodes Per Level	Local Work per Node
0	n	1	cn^2
1	n/4	3	$c(n/4)^2$
2	n/16	9	$c(n/16)^2$
j	$n/4^j$	3^j	$c(n/4^j)^2$
$\log_4 n$	1	$n^{\log_4 3}$	d

$T(n) = 3T(n/4) + cn^2$

Depth	Problem Size	# Nodes Per Level	Local Work per Node	Local Work per Level
0	n	1	cn^2	
1	n/4	3	$c(n/4)^2$	
2	n/16	9	$c(n/16)^2$	
j	$n/4^j$	3^j	$c(n/4^j)^2$	
$\log_4 n$	1	$n^{\log_4 3}$	d	

$T(n) = 3T(n/4) + cn^2$

Depth	**Problem Size**	**# Nodes Per Level**	**Local Work per Node**	**Local Work per Level**
0	n	1	cn^2	$1 \times cn^2$
1	n/4	3	$c(n/4)^2$	$3 \times c(n/4)^2$
2	n/16	9	$c(n/16)^2$	$9 \times c(n/16)^2$
j	$n/4^j$	3^j	$c(n/4^j)^2$	$3^j \times c(n/4^j)^2$
$\log_4 n$	1	$n^{\log_4 3}$	d	$dn^{\log_4 3}$

$T(n) = 3T(n/4) + cn^2$

Depth	Problem Size	# Nodes Per Level	Local Work per Node	Local Work per Level
0	n	1	cn^2	cn^2
1	n/4	3	$c(n/4)^2$	$3c(n/4)^2$
2	n/16	9	$c(n/16)^2$	$9c(n/16)^2$
j	$n/4^j$	3^j	$c(n/4^j)^2$	$3^j c(n/4^j)^2$
$\log_4 n$	1	$n^{\log_4 3}$	d	$dn^{\log_4 3}$

$T(n) = 3T(n/4) + cn^2$

Depth	Problem Size	# Nodes Per Level	Local Work per Node	Local Work per Level
0				
1				
2				
j				
$\log_4 n$	1	$n^{\log_4 3}$	d	$dn^{\log_4 3}$

$$T(n) = dn^{\log_4 3} + \sum_{j=0}^{\log_4 n - 1} 3^j c\left(n/4^j\right)^2$$

Switch to separate PDF for algebraic resolution of this formula into an asymptotic complexity

End of Lecture 4