Lecture 7: Solving Recurrences

CSE 373: Data Structures and Algorithms

Warm Up – Writing Recurrences

Administriva

HW 2 Part 1 due Friday

- git runners will get overloaded on Friday, plan accordingly

No Kasey office hours Friday

Solving Recurrences

Review: Modeling Recursion

Write a mathematical model of the following code

```
public int factorial(int n) {
    if (n == 0 || n == 1) {  +3
        return 1;   +1
    } else {
        return n * factorial(n-1);
    }               +2
}
```

$$T(n) = \begin{cases} 4 \; when \; n = 0,1 \\ 2 + \; T(n-1) \; otherwise \end{cases}$$

What is the Big O?

Solving Recurrences

How do we go from code model to Big O?

1. Explore the recursive pattern by tracing through the a few levels of recursion
2. Write a new model of the runtime or "work done" for the pattern in terms of the level of recursion "i"
3. Use algebra (and likely a summation) to simplify the T recursive call out of your new model
4. Use algebra to simplify down to the "closed form" so you can easily identify the Big O

Unrolling Method

Walk through function definition until you see a pattern

$$T(n) = \begin{cases} 4 & when\ n = 0,1 \\ 2 + T(n-1) & otherwise \end{cases}$$

$$T(n) = \boxed{\boxed{}} \Rightarrow T(n) = \begin{cases} 4\ when\ n = 0,1 \\ 2 + T(n-1)\ otherwise \end{cases} \Rightarrow T(n) = 2 + \boxed{2 + T(\boxed{n-1} - 1)}$$ *i = 1*

$$T(n) = 2 + 2 + \boxed{T(\boxed{n-2})} \Rightarrow T(n) = \begin{cases} 4\ when\ n = 0,1 \\ 1 + T(n-1)\ otherwise \end{cases} \Rightarrow T(n) = 2 + 2 + \boxed{2 + T(\boxed{n-2} - 1)}$$ *i = 2*

$$T(n) = 2 + 2 + 2 + \boxed{T(\boxed{n-3})} \Rightarrow T(n) = \begin{cases} 4\ when\ n = 0,1 \\ 1 + T(n-1)\ otherwise \end{cases} \Rightarrow T(n) = 2 + 2 + 2 + \boxed{2 + T(\boxed{n-3} - 1)}$$ *i = 3*

$$T(n) = 2 + 2 + 2 + 2 + \cdots + T(1) = \underbrace{2 + 2 + 2 + 2 +}_{\text{n-1 recursive cases}} \ldots + \underbrace{4}_{\text{1 base case}}$$ *T(n-i) = T(1) when i= n-1*

$$T(n) = 4 + \sum_{i=1}^{n-1} 2 \Rightarrow$$

Summation of a constant

$$\sum_{i=1}^{n} c = cn$$

$$\Rightarrow \boxed{T(n) = 4 + 2(n-1)}$$

$$T(4) = 2 + T(4-1) = 2 + 2 + T(3-1) = 2 + 2 + 2 + T(2-1) = 2 + 2 + 2 + 4 = 3 * 2 + 4$$

5 Minutes

Unrolling Method

Walk through function definition until you see a pattern

$$T(n) = \begin{cases} 1 \; when \; n = 0 \\ 2T(n-1) + 1 \; otherwise \end{cases}$$

i = 1 $= 2T(n-1) + 1$

i = 2 $= 2(2T(n-2)+1)+1 = 2^2T(n-2) + 2 + 1$

i = 3 $= 2^2(2T(n-3)+1)+2+1 = 2^3T(n-3) + 2^2 + 2^1 + 2^0$

i = 4 $= 2^3(2T(n-4)+1) + 2^2 + 2^1 + 2^0 = 2^4T(n-4) + 2^3 + 2^2 + 2^1 + 2^0$

i = n-i $= 2^iT(n-i) + 2^{i-1} + 2^{i-2} + 2^{i-3} + \cdots + 2^0 = 2^iT(n-i) + \sum_{j=0}^{i-1} 2^j = 2^n\,T(n-n) + \sum_{j=0}^{n-1} 2^j = 2^n(1) + \sum_{j=0}^{n-1} 2^j$

Finite Geometric Series

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

$$= 2^n + \frac{2^n - 1}{2-1} = 2^n + 2^n - 1 = \boxed{2^{n+1} - 1}$$

Solving Recurrences

How do we go from code model to Big O?

1. Explore the recursive pattern
2. Write a new model in terms of “i”
3. Use algebra simplify the T away
4. Use algebra to find the “closed form”

Using unrolling method

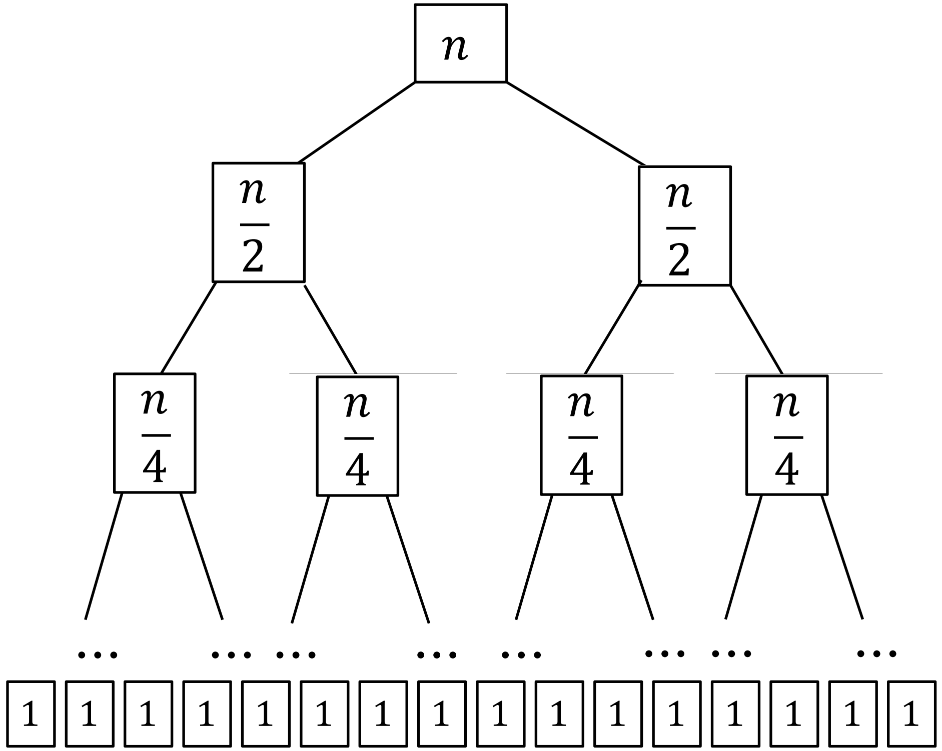
1. Plug definition into itself to write out first few levels of recursion
2. Simplify away parenthesis but leave separate terms to help identify pattern in terms of i
3. Plug in a value of i to solve for base case, write summation representing recursive work
4. Using summation identities as appropriate reduced to “closed form”

Tree Method

Draw out call stack, how much work does each call do?

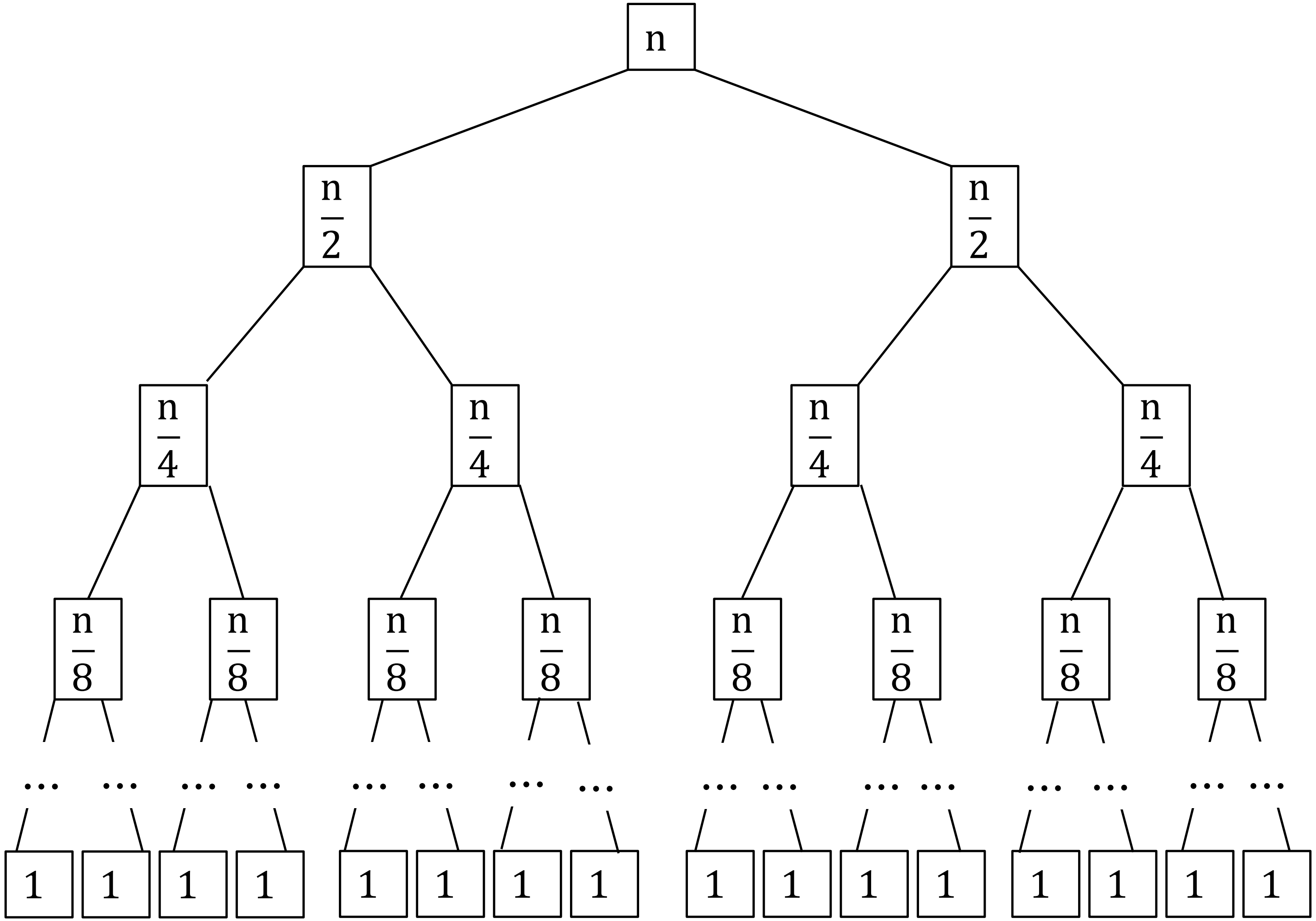
$$T(n) = \begin{cases} 1 \; when \; n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n \; otherwise \end{cases}$$

1. Draw an overall root representing the start of your family of recursive calls
2. How much work is done by the top recursive level?
3. How much of that work is delegated to downstream recursive calls?
4. How much work is done by each of those child recursive calls?
5. How much of that work is delegated to downstream recursive calls?
6. …
7. What does the last row of the tree look like?
8. Sum up all the work!



Tree Method

$$T(n) = \begin{cases} 1 \text{ when } n \le 1 \\ 2T\left(\frac{n}{2}\right) + n \text{ otherwise} \end{cases}$$



How many pieces of work at each level?	How much work done by each piece?	How much work across each level?
1	n	n
2	$\frac{n}{2}$	n
4	$\frac{n}{2^2}$	n
8	$\frac{n}{2^3}$	n
n	1	n

Tree Method Formulas

$$T(n) = \begin{cases} 1 \text{ when } n \le 1 \\ 2T\left(\frac{n}{2}\right) + n \text{ otherwise} \end{cases}$$

How much work is done by recursive levels (branch nodes)?

1. How many recursive calls are on the i-th level of the tree?
   - i = 0 is overall root level

numberNodesPerLevel(i) = 2^i

2. At each level i, how many inputs does a single node process?

inputsPerRecursiveCall(i) = $(n/ 2^i)$

3. How many recursive levels are there?
   - Based on the pattern of how we get down to base case

branchCount = $\log_2 n - 1$

$$Recursive\ work = \sum_{i=0}^{branchCount} branchNum(i) branchWork(i)$$

$$T(n > 1) = \sum_{i=0}^{\log_2 n - 1} 2^i \left(\frac{n}{2^i}\right)$$

How much work is done by the base case level (leaf nodes)?

1. How much work is done by a single leaf node?

leafWork = 1

2. How many leaf nodes are there?

leafCount = $2^{\log_2 n} = n$

$$NonRecursive\ work = leafWork \times leafCount = leafWork \times branchNum^{numLevels}$$

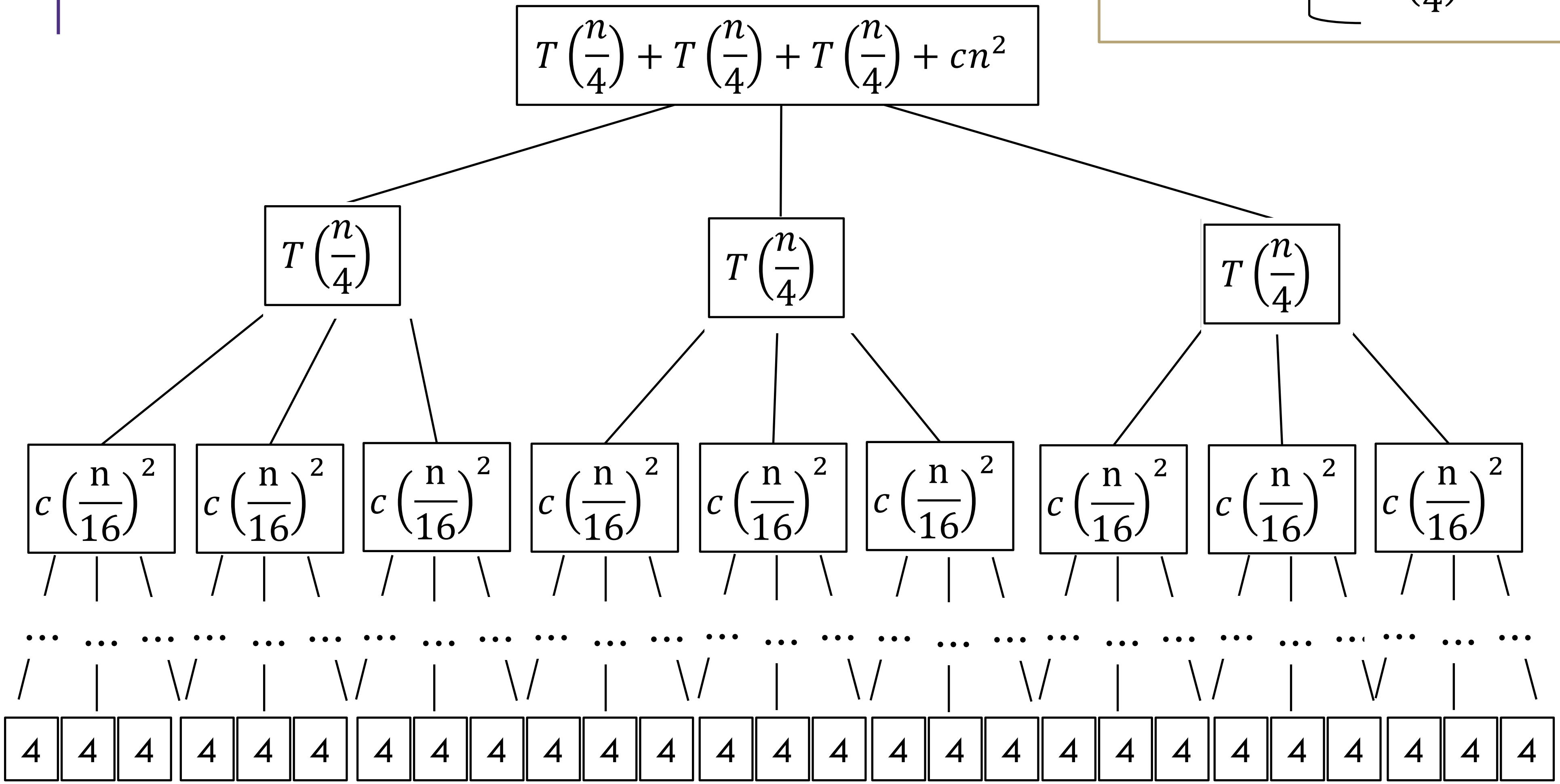
$$T(n \le 1) = 1\left(2^{\log_2 n}\right) = n$$

$$total\ work = recursive\ work + nonrecursive\ work =$$

$$T(n) = \sum_{i=0}^{\log_2 n - 1} 2^i \left(\frac{n}{2^i}\right) + n = n \log_2 n + n$$

Tree Method Practice

$$T(n) = \begin{cases} 4 \; when \; n \leq 1 \\ 3T\left(\frac{n}{4}\right) + cn^2 \; otherwise \end{cases}$$



Answer the following questions:

1. How many nodes on each branch level?
2. How much work for each branch node?
3. How much work per branch level?
4. How many branch levels?
5. How much work for each leaf node?
6. How many leaf nodes?

Tree Method Practice

5 Minutes

$$T(n) = \begin{cases} 4 \text{ when } n \le 1 \\ 3T\left(\frac{n}{4}\right) + cn^2 \text{ otherwise} \end{cases}$$

1. How many nodes on each branch level? 3^i
2. How much work for each branch node? $c\left(\frac{n}{4^i}\right)^2$
3. How much work per branch level? $3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$
4. How many branch levels? $\log_4 n - 1$
5. How much work for each leaf node? 4
6. How many leaf nodes? $3^{\log_4 n}$ power of a log $x^{\log_b y} = y^{\log_b x}$ $n^{\log_4 3}$

Level (i)	Number of Nodes	Work per Node	Work per Level
0	1	cn^2	cn^2
1	3	$c\left(\frac{n}{4}\right)^2$	$\frac{3}{16}cn^2$
2	9	$c\left(\frac{n}{16}\right)^2$	$\frac{9}{256}cn^2$
base	$3^{\log_4 n}$	4	$12^{\log_4 n}$

Combining it all together…

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + 4n^{\log_4 3}$$

Tree Method Practice

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + 4n^{\log_4 3}$$

factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c\sum_{i=a}^{b} f(i)$$

$$T(n) = cn^2 \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i + 4n^{\log_4 3}$$

finite geometric series

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

Closed form:

$$T(n) = cn^2 \left(\frac{\frac{3}{16}^{\log_4 n} - 1}{\frac{3}{16} - 1}\right) + 4n^{\log_4 3}$$

If we're trying to prove upper bound…

$$T(n) = cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + 4n^{\log_4 3}$$

infinite geometric series

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x}$$

when -1 < x < 1

$$T(n) = cn^2 \left(\frac{1}{1 - \frac{3}{16}}\right) + 4n^{\log_4 3}$$

$$T(n) \in O(n^2)$$

Solving Recurrences

How do we go from code model to Big O?

1. Explore the recursive pattern
2. Write a new model in terms of "i"
3. Use algebra simplify the T away
4. Use algebra to find the "closed form"

Using unrolling method

1. Plug definition into itself to write out first few levels of recursion
2. Simplify away parenthesis but leave separate terms to help identify pattern in terms of i
3. Plug in a value of i to solve for base case, write summation representing recursive work
4. Using summation identities as appropriate reduced to "closed form"

Using tree method

1. Plug definition into itself to draw out first few levels of tree
2. Answer questions about nature of tree to identify work done by recursive levels and base case in terms of i
3. Combine answers to questions to complete model in terms of i
4. Using summation identities as appropriate reduced to "closed form"

Is there an easier way?

What if you do want an exact closed form?

Sorry, no

If we want to find a big Θ

Sometimes, yes!

Master Theorem

Given a recurrence of the following form:

$$T(n) = \begin{cases} d \text{ } when \text{ } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \text{ } otherwise \end{cases}$$

Then thanks to magical math brilliance we can know the following:

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

Apply Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \; when \; n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \; otherwise \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

$$T(n) = \begin{cases} 1 \; when \; n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n \; otherwise \end{cases}$$

a = 2
b = 2
c = 1
d = 1

$$\log_b a = c \Rightarrow \log_2 2 = 1$$

$$T(n) \in \Theta(n^c \log_2 n) \Rightarrow \Theta(n^1 \log_2 n)$$

Reflecting on Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \text{ } when \text{ } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \text{ } otherwise \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

$height \approx \log_b a$

$branchWork \approx n^c \log_b a$

$leafWork \approx d\left(n^{\log_b a}\right)$

The $\log_b a < c$ case

- Recursive case conquers work more quickly than it divides work
- Most work happens near “top” of tree
- Non recursive work in recursive case dominates growth, n^c term

The $\log_b a = c$ case

- Work is equally distributed across call stack (throughout the “tree”)
- Overall work is approximately work at top level x height

The $\log_b a > c$ case

- Recursive case divides work faster than it conquers work
- Most work happens near “bottom” of tree
- Leaf work dominates branch work