Lecture 7: Solving Recurrences

CSE 373: Data Structures and Algorithms

5 Minutes

Thought Experiment

Discuss with your neighbors: Imagine you are writing an implementation of the List interface that stores integers in an Array. What are some ways you can assess your program's correctness in the following cases:

Expected Behavior
- Create a new list
- Add some amount of items to it
- Remove a couple of them

Forbidden Input
- Add a negative number
- Add duplicates
- Add extra large numbers

Empty/Null
- Call remove on an empty list
- Add to a null list
- Call size on an null list

Boundary/Edge Cases
- Add 1 item to an empty list
- Set an item at the front of the list
- Set an item at the back of the list

Scale
- Add 1000 items to the list
- Remove 100 items in a row
- Set the value of the same item 50 times

Extra Credit:
Go to PollEv.com/champk
Text CHAMPK to 22333 to join session, text "1" or "2" to select your answer

Administriva

Solving Recurrences

1 Minute

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);
    }              +1          +??????
}                          recurrence!
```

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

Writing a Recurrence

If the function runs recursively, our formula for the running time should probably be recursive as well.

- *Such a formula is called a **recurrence**.*

$$T(n) = \begin{cases} T(n-1) + 2 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

What does this say?

- The input to T is the size of the input to the Length.
- If the input to T() is large, the running time depends on the recusive call.
- If not we can just use the base case.

Another example

```
public int Mystery(int n){
    if(n == 1) { +1
        return 1; +1
    } else {
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                System.out.println(“hi!”); +1      n      n
            }
        }
        return Mystery(n/2)
    }
}
```

$$T(n) = \begin{cases} C \;\; when\; n = 1 \\ T(n/2) + n^2 \text{ if n} > 1 \end{cases}$$

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"

Three Methods:

1. Tree Method – draw out the branching nature of recursion to find pattern
2. Unrolling – plug function into itself to find pattern
3. Master Theorem – plug and chug!

Master Theorem

Given a recurrence of the following form:

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

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 $T(n) \in \Theta\left(n^{\log_b a}\right)$

Review: Logarithms

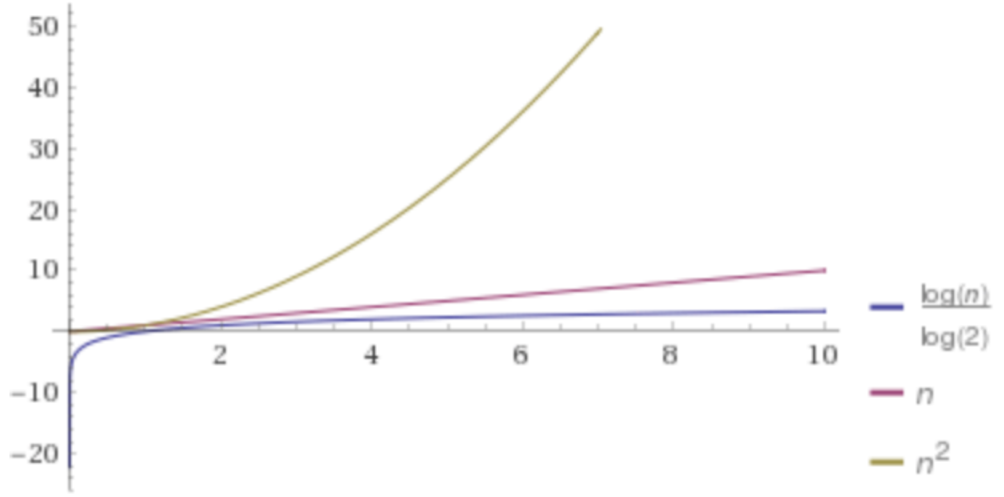
Logarithm – inverse of exponentials

$if\ b^x = n\ then\ x = \log_b n$

Examples:

$2^2 = 4 \Rightarrow 2 = \log_2 4$

$3^2 = 9 \Rightarrow 2 = \log_3 9$



Apply Master Theorem

Given a recurrence of the form:

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

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

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

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

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

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

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

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

Step 1: Code -> Recurrence

```
public static int mystery(int arr[], int min, int max, int val) {
   if (max < 1) {
      return -1;
   } else {
      int mid = min + (max - 1) / 2;
      if (arr[mid] == val) {
         return mid;
      }
      if (arr[mid] > val) {
         return binarySearch(arr, min, mid - 1, val);
      } else {
         return binarySearch(arr, mid + 1, max, val);
      }
   }
}
```

Reflecting on Master Theorem

Given a recurrence of the form:

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

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 $T(n) \in \Theta(n^{\log_b a})$

$height \approx \log_b a$

$branchWork \approx n^c \log_b a$

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

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

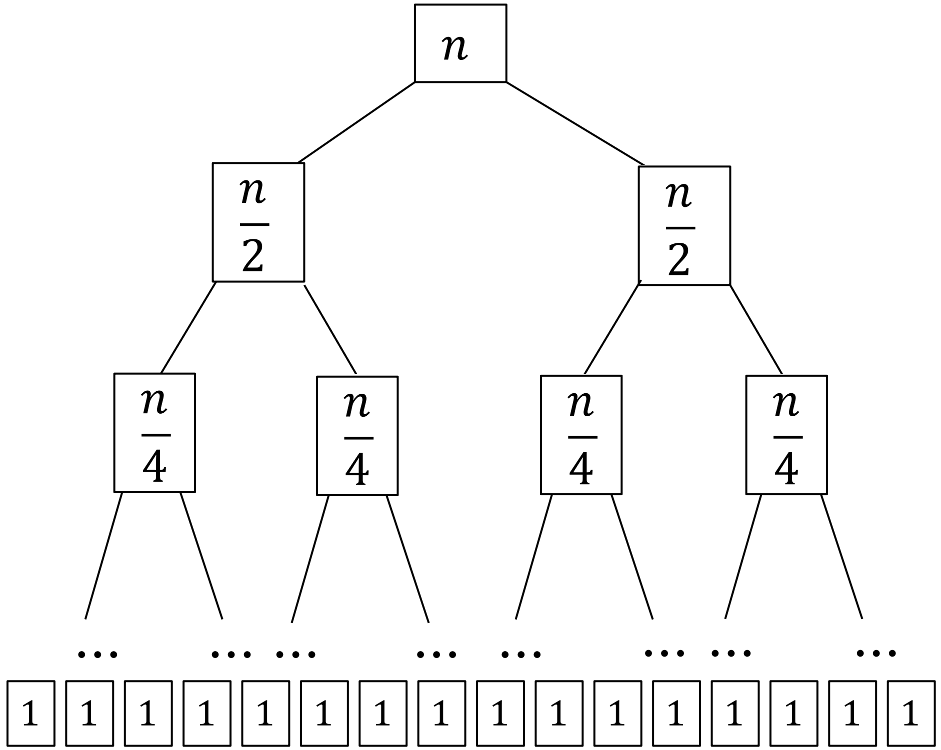
- Recursive case divides work faster than it conquers work
- Most work happens near “bottom” of tree
- Leaf work dominates branch work

Tree Method

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

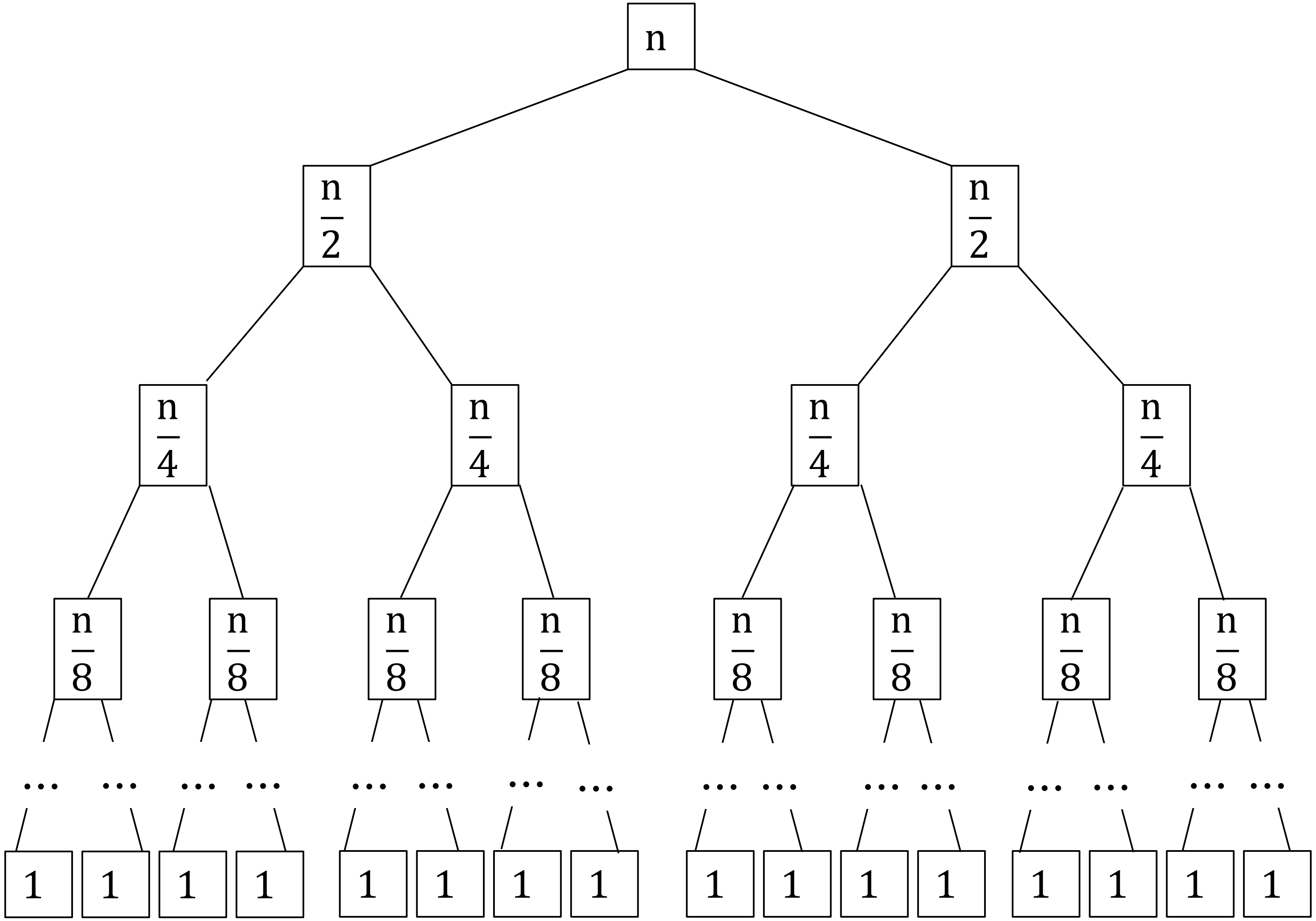
$$T(n) = \begin{cases} 1 \; when \; n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n \; otherwise \end{cases}$$

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

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



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

numRecursiveLevels = $\log_2 n - 1$

$$Recursive\ work = \sum_{i=0}^{numRecursiveLevels} numberNodesPerLevel(i) branchWork(i)$$

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

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

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

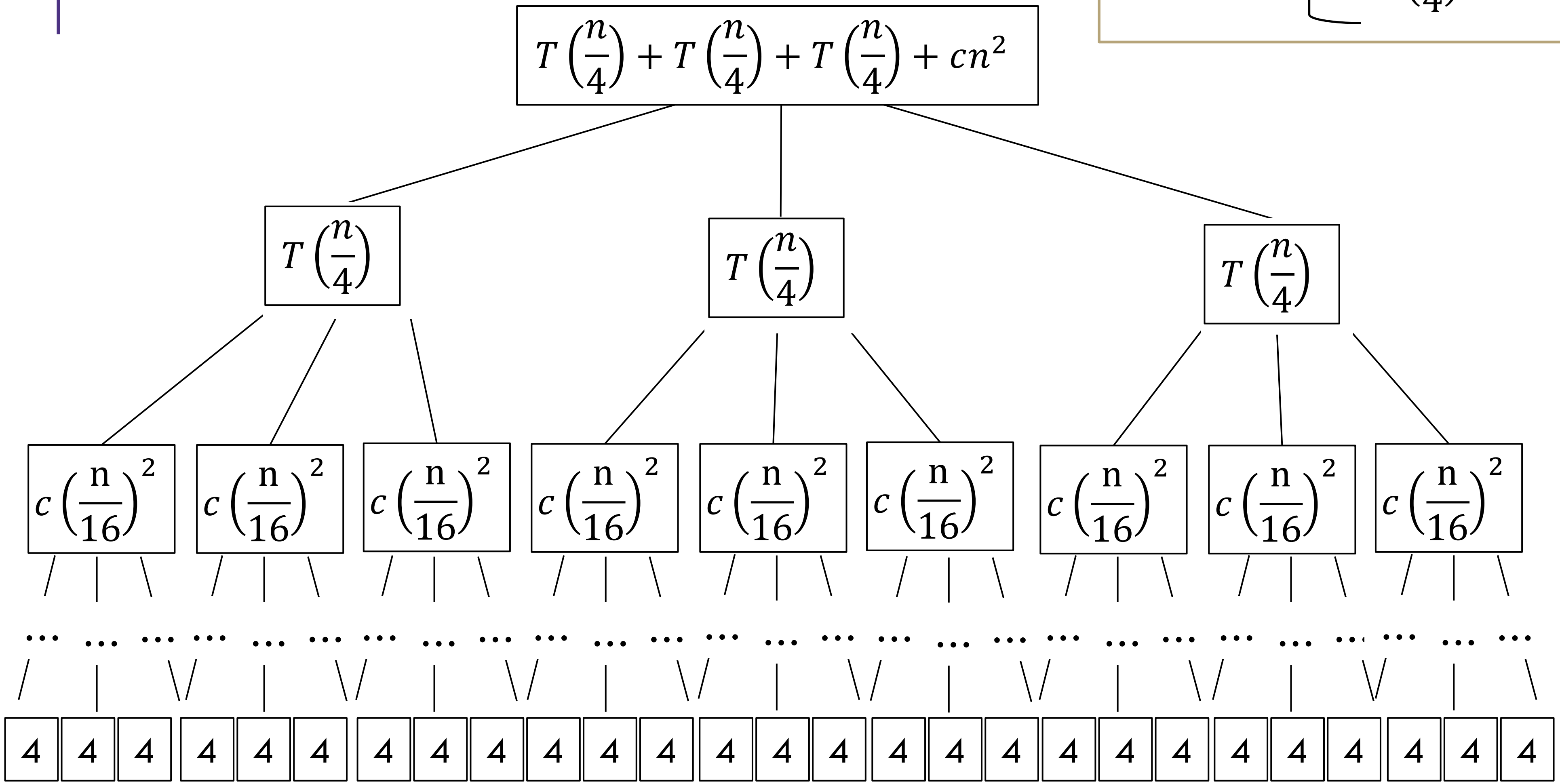
$$base\ case\ work = leafWork \times leafCount = leafWork \times numberNodesPerLevel^{numRecursiveLevels+1}$$

$$total\ work = recursive\ work + base\ case\ work =$$

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

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?

EXAMPLE PROVIDED BY CS 161 – JESSICA SU
HTTPS://WEB.STANFORD.EDU/CLASS/ARCHIVE/CS/CS161/CS161.1168/LECTURE3.PDF

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? $3^{\log_4 n}$ $n^{\log_4 3}$

power of a log
$x^{\log_b y} = y^{\log_b x}$

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

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

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