Lecture 3: Asymptotic Analysis + Recurrences

Data Structures and Algorithms

Warmup – Write a model and find Big-O

```
for (int i = 0; i < n; i++) {       runs n times
    for (int j = 0; j < i; j++) {   runs sum(0 + 1 + 2 + … + n-1) times
        System.out.println("Hello!");
    }
}
```

Summation

$1 + 2 + 3 + 4 + \ldots + n = \sum_{i=1}^{n} i$

Definition: Summation

$\sum_{i=a}^{b} f(i)$ = f(a) + f(a + 1) + f(a + 2) + … + f(b-2) + f(b-1) + f(b)

$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} c$

Simplifying Summations

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!");
    }
}
```

where this inner part takes c operations

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{i-1} c = \sum_{i=0}^{n-1} ci$$ Summation of a constant

$$= c\sum_{i=0}^{n-1} i$$ Factoring out a constant

$$= c\frac{n(n-1)}{2}$$ Gauss's Identity

$$= \frac{c}{2}n^2 - \frac{c}{2}n$$ $O(n^2)$

Function Modeling: Recursion

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

Function Modeling: Recursion

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

$+C_1$

$+T(n-1)$

$+C_2$

$$T(n) = \begin{cases} C_1 & \text{when } n = 0 \text{ or } 1 \\ C_2 + T(n-1) & \text{otherwise} \end{cases}$$

Definition: Recurrence
Mathematical equation that recursively defines a sequence The notation above is like an if / else statement

Unfolding Method

$$T(n) = \begin{cases} C_1 & \text{when } n = 0 \text{ or } 1 \\ C_2 + T(n-1) & \text{otherwise} \end{cases}$$

$$T(3) = C_2 + T(3-1) = C_2 + (C_2 + T(2-1)) = C_2 + (C_2 + (C_1)) = 2C_2 + C_1$$

$$T(n) = C_1 + \sum_{i=0}^{n-1} C_2$$

Summation of a constant

$$T(n) = C_1 + (n-1)C_2$$

Announcements

- Course background survey due by Friday
- HW 1 is Due Friday
- HW 2 Assigned on Friday – Partner selection forms due by 11:59pm **Thursday**

https://goo.gl/forms/rVrVUkFDdsqI8pkD2

A Detour on Style

- Checkstyle for project
  - No packages for HW1
  - Braces for blocks
- Good style is easy to read
  - Javadoc on public methods (not needed if interface has Javadoc)
  - Comment non-obvious code
    - Self-Documenting code is better than commented code
      - Good variable and method names go a long way towards this
  - No magic numbers (numbers larger than 2 or 3 should probably be class constants unless there's a really good reason)
  - No code duplication
  - Use Idioms!

    ex. for (int I = 0; I < 10; i++) instead of for (int I = 0; I == 9; i = i + 1)

    naming: CONSTANTS_USE_CAPS, ClassName, methodName

Any special numbers > 2 or 3 should be made into global constants if they are repeatedly used (DEFAULT_CAPACITY)

Don’t reuse code…

comment any possibly confusing/complex code.
comment each method (Javadocs style); all public methods need JavaDocs, private methods can just use in-line comments

example of javaDocs style

```
    public void add(T item);

    /**
     * Removes an item from the front of the queue.
     * @param item
                      item to be added to queue
     * @return The item at the front of the queue.
     * @throws NoSuchElementException
     *             If the queue is empty.
     */
```

Try to make code obvious:
i.e use is.Empty() rather than size == 0

Tree Method

Idea:

- Since we're making recursive calls, let's just draw out a tree, with one node for each recursive call.
- Each of those nodes will do some work, and (if they make more recursive calls) have children.
- If we can just add up all the work, we can find a big-Θ bound.

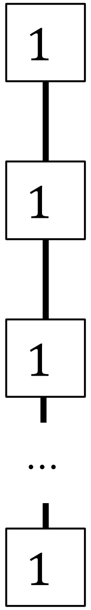
Solving Recurrences I: Binary Search

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

0. Draw the tree.
1. What is the input size at level i?
2. What is the number of nodes at level i?
3. What is the work done at recursive level i?
4. What is the last level of the tree?
5. What is the work done at the base case?
6. Sum over all levels (using 3,5).
7. Simplify

Solving Recurrences I: Binary Search

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



0. Draw the tree.
1. What is the input size at level i?
2. What is the number of nodes at level i?
3. What is the work done at recursive level i?
4. What is the last level of the tree?
5. What is the work done at the base case?
6. Sum over all levels (using 3,5).
7. Simplify

Solving Recurrences I: Binary Search

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

Level	Input Size	Work/call	Work/level
0	n	1	1
1	$n/2$	1	1
2	$n/2^2$	1	1
i	$n/2^i$	1	1
$\log_2 n$	1	1	1

0. Draw the tree.
1. What is the input size at level i?
2. What is the number of nodes at level i?
3. What is the work done at recursive level i?
4. What is the last level of the tree?
5. What is the work done at the base case?
6. Sum over all levels (using 3,5).
7. Simplify

Text

work per level

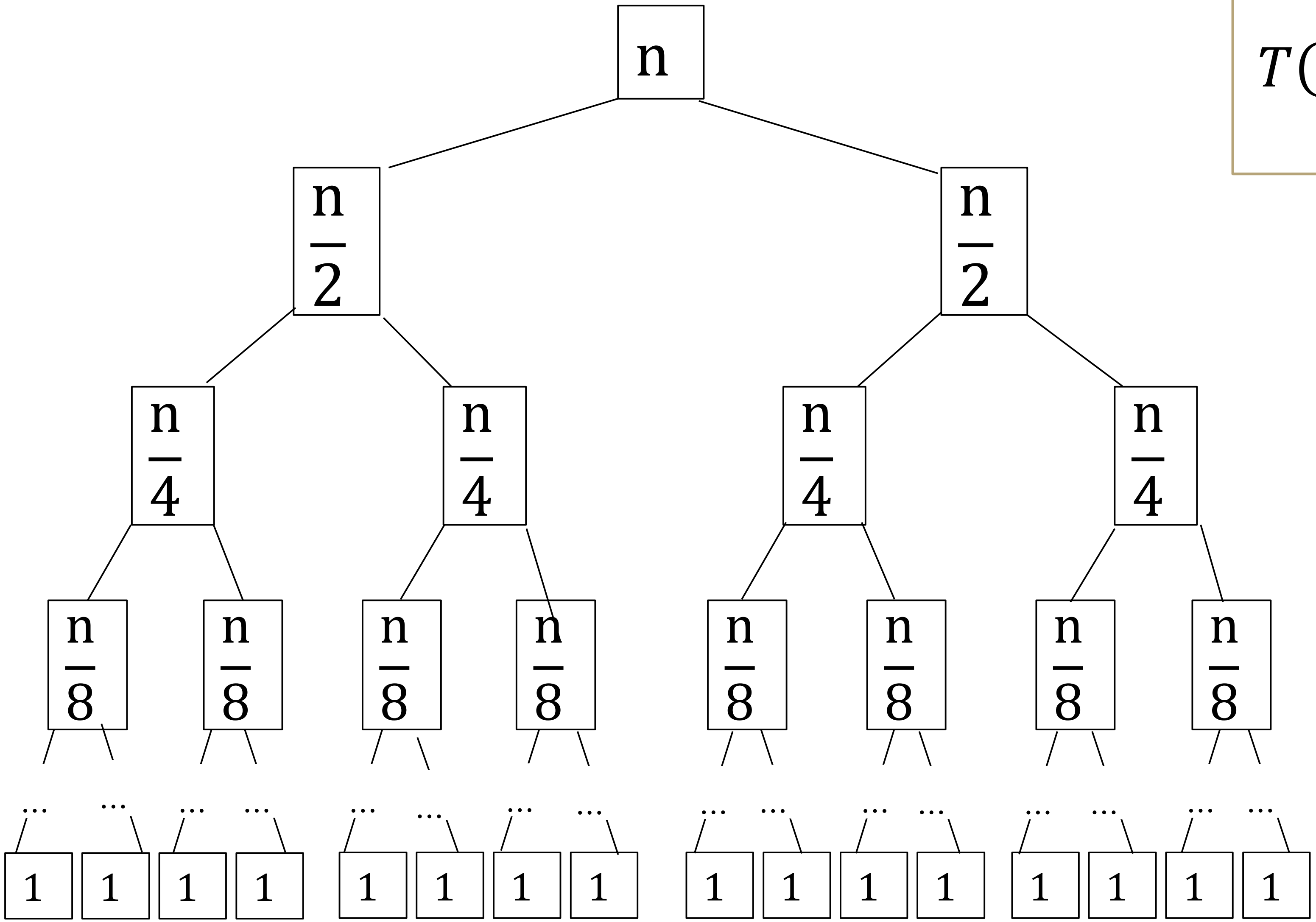
$$\sum_{i=0}^{\log_2 n - 1} 1 + 1 = \log_2 n$$

+ 1 = log2N

work done for recursive case

work done for base case

Solving Recurrences II: MERGE sort



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

Two recursive calls per level: i.e.

```
public int sum(IntTreeNode root) {
    if(root == null) {
        return 0;
    } else {
        return root.data + sum(root.left) + sum(root.right);
    }
}
```

Tree Method Formulas

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

work at each node is proportional to size

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

1. What is the input size at level i?
   - $i = 0$ is overall root level.

   $(n/2^i)$

   work at level i = plug size at level i into

2. At each level i, how many calls are there?

   2^i

3. At each level i, how much work is done??

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

   (# nodes at level i) * (work at level i per node)

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

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

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

4. What is the last level of the tree? $(n/2^i) = 1 \rightarrow 2^i = n \rightarrow i = \log_2 n$
5. What is the work done at the last level?

$$NonRecursive\ work = WorkPerBaseCase \times numberCalls$$

$$1 \cdot 2^{\log_2 n} = n$$

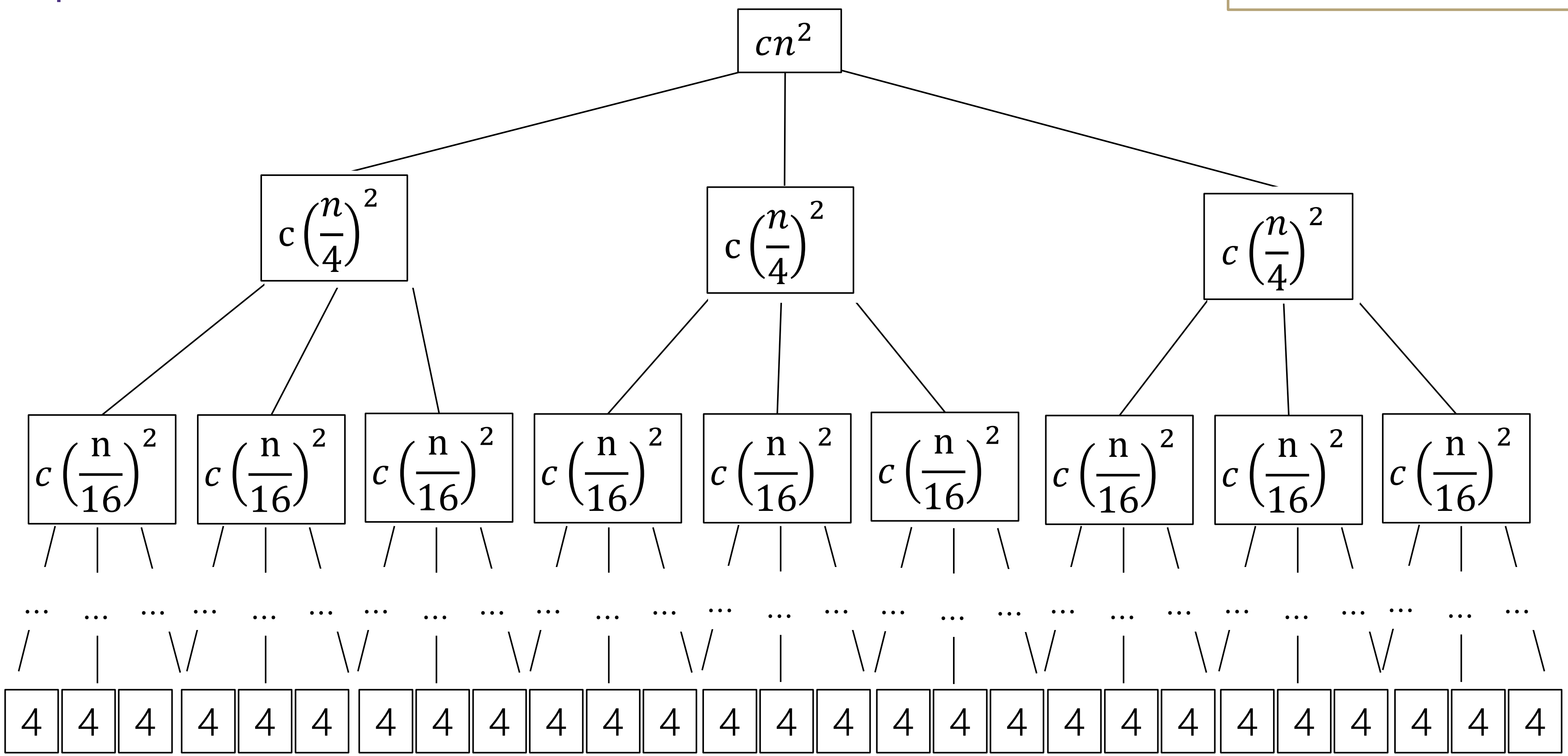
6. Combine and Simplify

$$T(n) = \underbrace{\sum_{i=0}^{\log_2 n - 1} 2^i \left(\frac{n}{2^i}\right)}_{\text{recursive work}} + \underbrace{n}_{\text{base case work}} = n \log_2 n + n$$

log2n -1 NON base case levels

Solving Recurrences III

$$T(n) = \begin{cases} 5 \text{ when } n \le 4 \\ 3T\left(\frac{n}{4}\right) + cn^2 \text{ otherwise} \end{cases}$$



Answer the following questions:
1. What is input size on level i?
2. Number of nodes at level i?
3. Work done at recursive level i?
4. Last level of tree?
5. Work done at base case?
6. What is sum over all levels?

Solving Recurrences III

$$T(n) = \begin{cases} 5 \text{ when } n \le 4 \\ 3T\left(\frac{n}{4}\right) + cn^2 \text{ otherwise} \end{cases}$$

1. Input size on level i? $\frac{n}{4^i}$

2. How many calls on level i? 3^i

3. How much work on level i? $3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$

4. What is the last level? When $\frac{n}{4^i} = 4 \rightarrow \log_4 n - 1$

5. A. How much work for each leaf node? 5

B. How many base case calls? $3^{\log_4 n - 1} = \frac{3^{\log_4 n}}{3}$ power of a log $x^{\log_b y} = y^{\log_b x}$ $= \frac{n^{\log_4 3}}{3}$

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	3^2	$c\left(\frac{n}{4^2}\right)^2$	$\left(\frac{3}{16}\right)^2 cn^2$
i	3^i	$c\left(\frac{n}{4^i}\right)^2$	$\left(\frac{3}{16}\right)^i cn^2$
Base = $\log_4 n - 1$	$3^{\log_4 n - 1}$	5	$\left(\frac{5}{3}\right) n^{\log_4 3}$

6. Combining it all together…

$$\boxed{T(n) = \sum_{i=0}^{\log_4 n - 2} \left(\frac{3}{16}\right)^i cn^2 + \left(\frac{5}{3}\right) n^{\log_4 3}}$$

Solving Recurrences III

7. Simplify…

$$T(n) = \sum_{i=0}^{\log_4 n - 2} \left(\frac{3}{16}\right)^i cn^2 + \left(\frac{5}{3}\right) n^{\log_4 3}$$

factoring out a constant

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

$$T(n) = cn^2 \sum_{i=0}^{\log_4 n - 2} \left(\frac{3}{16}\right)^i + \left(\frac{5}{3}\right) n^{\log_4 3}$$

finite geometric series

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

Closed form:

$$T(n) = cn^2 \left(\frac{\frac{3}{16}^{\log_4 n - 1} - 1}{\frac{3}{16} - 1} \right) + \left(\frac{5}{3}\right) n^{\log_4 3}$$

If we're trying to prove upper bound…

$$T(n) \le cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + \left(\frac{5}{3}\right) n^{\log_4 3}$$

infinite geometric series

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

when -1 < x < 1

$$T(n) \le cn^2 \left(\frac{1}{1 - \frac{3}{16}} \right) + \left(\frac{5}{3}\right) n^{\log_4 3}$$

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

Another Example

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ T(n-2) + 4 & \text{otherwise} \end{cases}$$

Is there an easier way?

We do all that effort to get an exact formula for the number of operations,

But we usually only care about the Θ bound.

There must be an easier way

Sometimes, there is!

Master Theorem

Given a recurrence of the following form:

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

Where a, b, c, and d are all constants.
The big-theta solution always follows this pattern:

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

If $\log_b a = c$ then $T(n)$ is $\Theta(n^c \log n)$

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

Apply Master Theorem

Given a recurrence of the form:

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

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

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

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

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

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

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

$$T(n) \text{ is } \Theta(n^c \log_2 n) \Rightarrow \Theta(n^1 \log_2 n)$$

Reflecting on Master Theorem

Given a recurrence of the form:

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

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

If $\log_b a = c$ then $T(n)$ is $\Theta(n^c \log n)$

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

$height \approx \log_b a$

$branchWork \approx n^c \log_b a$

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

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 levels of the tree
- Overall work is approximately work at any 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
- Work at base case dominates.

Benefits of Solving By Hand

If we had the Master Theorem why did we do all that math???

Not all recurrences fit the Master Theorem.

- Recurrences show up everywhere in computer science.
- And they're not always nice and neat.

It helps to understand exactly where you're spending time.

- Master Theorem gives you a very rough estimate. The Tree Method can give you a much more precise understanding.

Amortization

What's the worst case for inserting into an ArrayList?

- O(n). If the array is full.

Is O(n) a good description of the worst case behavior?

- If you're worried about a single insertion, maybe.
- If you're worried about doing, say, n insertions in a row. NO!

Amortized bounds let us study the behavior of a bunch of consecutive calls.

Amortization

The most common application of amortized bounds is for insertions/deletions and data structure resizing.

Let's see why we always do that doubling strategy.

How long in total does it take to do n insertions?

We might need to double a bunch, but the total resizing work is at most O(n)

And the regular insertions are at most $n \cdot O(1) = O(n)$

So n insertions take $O(n)$ work total

Or amortized $O(1)$ time.

Amortization

Why do we double? Why not increase the size by 10,000 each time we fill up?

How much work is done on resizing to get the size up to n?

Will need to do work on order of current size every 10,000 inserts

$\sum_{i=0}^{\frac{n}{10000}} 10000i \approx 10{,}000 \cdot \frac{n^2}{10{,}000^2} = O(n^2)$

The other inserts do $O(n)$ work total.

The amortized cost to insert is $O\left(\frac{n^2}{n}\right) = O(n)$.

Much worse than the $O(1)$ from doubling!