Lecture 3: Asymptotic Analysis + Recurrences

Data Structures and Algorithms
Warmup – Write a model and find Big-O

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

Summation
1 + 2 + 3 + 4 + ... + 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

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

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

\[
= c \sum_{i=0}^{n-1} i \quad \text{Factoring out a constant}
\]

\[
= c \frac{n(n-1)}{2} \quad \text{Gauss’s Identity}
\]

\[
= \frac{c}{2} n^2 - \frac{c}{2} n \quad \text{O}(n^2)
\]
public int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
Function Modeling: Recursion

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

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
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-$\Theta$ bound.
Solving Recurrences I: Binary Search

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

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

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

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

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

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

\[ \sum_{i=0}^{\log_2 n-1} 1 + 1 = \log_2 n \]
Solving Recurrences II:

\[ T(n) = \begin{cases} 1 & \text{when } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases} \]
Tree Method Formulas

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.
2. At each level $i$, how many calls are there?
3. At each level $i$, how much work is done??

\[
\text{Recursive work} = \sum_{i=0}^{\text{lastRecursiveLevel}} \text{branchNum}(i) \text{branchWork}(i)
\]

How much work is done by the base case level (leaf nodes)?
4. What is the last level of the tree?
   \[ \frac{n}{2^i} = 1 \rightarrow 2^i = n \rightarrow i = \log_2 n \]
5. What is the work done at the last level?
   \[ \text{NonRecursive work} = \text{WorkPerBaseCase} \times \text{numberCalls} \]

\[ 1 \cdot 2^{\log_2 n} = n \]

6. Combine and Simplify
\[
T(n) = \sum_{i=0}^{\log_2 n - 1} 2^i \left( \frac{n}{2^i} \right) + n = n \log_2 n + n
\]

\[ T(n) = \begin{cases} 
1 & \text{when } n \leq 1 \\
2T \left( \frac{n}{2} \right) + n & \text{otherwise}
\end{cases} \]
Solving Recurrences III

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?

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

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? $3^{\log_4 n - 1} = \frac{3^{\log_4 n}}{3}$

6. Combining it all together...

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

<table>
<thead>
<tr>
<th>Level ($i$)</th>
<th>Number of Nodes</th>
<th>Work per Node</th>
<th>Work per Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$cn^2$</td>
<td>$cn^2$</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>$c\left(\frac{n}{4}\right)^2$</td>
<td>$\frac{3}{16}cn^2$</td>
</tr>
<tr>
<td>2</td>
<td>$3^2$</td>
<td>$c\left(\frac{n}{4^2}\right)^2$</td>
<td>$\left(\frac{3}{16}\right)^2 cn^2$</td>
</tr>
<tr>
<td>$i$</td>
<td>$3^i$</td>
<td>$c\left(\frac{n}{4^i}\right)^2$</td>
<td>$\left(\frac{3}{16}\right)^i cn^2$</td>
</tr>
</tbody>
</table>

Base = $\log_4 n - 1$ $3^{\log_4 n - 1}$ 5 $\left(\frac{5}{3}\right)^{n\log_4 3}$
Solving Recurrences III

7. Simplify...

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

factoring out a constant

\[ \sum_{i=a}^{b} c f(i) = c \sum_{i=a}^{b} f(i) \]

finite geometric series

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

infinite geometric series

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

when \(-1 < x < 1\)

If we’re trying to prove upper bound...

\[ T(n) \leq c n^2 \sum_{i=0}^{\infty} \left( \frac{3}{16} \right)^i + \left( \frac{5}{3} \right) n^{\log_4 3} \]

Closed form:

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

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

\[ T(n) \leq c n^2 \left( \frac{1}{1 - \frac{3}{16}} \right) + \left( \frac{5}{3} \right) n^{\log_4 3} \]

\[ T(n) \in O(n^2) \]
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 $\Theta$ bound.
There must be an easier way
Sometimes, there is!
Master Theorem

Given a recurrence of the following form:

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

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 \(\Theta(n^{\log_b a})\)
Apply 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 \( \Theta(n^{\log_b a}) \)

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

\[ a = 2 \quad b = 2 \quad c = 1 \quad d = 1 \]

\[ \log_b a = c \Rightarrow \log_2 2 = 1 \]

\[ T(n) \text{ is } \Theta(n^c \log_2 n) \Rightarrow \Theta(n^1 \log_2 n) \]
Reflecting on Master Theorem

Given a recurrence of the form:

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

1. If \( \log_b a < c \) then \( T(n) \) is \( \Theta(n^c) \)
2. If \( \log_b a = c \) then \( T(n) \) is \( \Theta(n^c \log n) \)
3. If \( \log_b a > c \) then \( T(n) \) is \( \Theta(n^{\log_b a}) \)

- **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 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

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

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!