Recurrences

- Divide and Conquer
- What is a Recurrence?
- The Recursion-Tree Method
- The Substitution Method
- The Master Method
Divide and Conquer

- Divide and Conquer is an algorithm strategy
  - Divide the input data into multiple pieces
  - Recurse into each piece
  - Join the answers together
Divide and Conquer

findMax(array):

    part1 = lower half of array
    part2 = upper half of array
    max1 = findMax(part1)
    max2 = findMax(part2)
    return max(max1, max2)

(We're temporarily overlooking the base case...)
Divide and Conquer

\[
\text{findMax}(array):
\]

\[
\begin{align*}
\text{part1} &= \text{lower half of array} \\
\text{part2} &= \text{upper half of array} \\
\text{max1} &= \text{findMax}(\text{part1}) \\
\text{max2} &= \text{findMax}(\text{part2}) \\
\text{return max}(\text{max1}, \text{max2})
\end{align*}
\]
Divide and Conquer

\[
\text{findMax}(\text{array}): \\
\text{part1} = \text{lower half of array} \\
\text{part2} = \text{upper half of array} \\
\text{max1} = \text{findMax}(\text{part1}) \\
\text{max2} = \text{findMax}(\text{part2}) \\
\text{return} \ \text{max}(\text{max1}, \text{max2})
\]
Divide and Conquer

\[
\text{findMax(array)}: \\
\text{part1} = \text{lower half of array} \\
\text{part2} = \text{upper half of array} \\
\text{max1} = \text{findMax(part1)} \\
\text{max2} = \text{findMax(part2)} \\
\text{return } \max(\text{max1},\text{max2})
\]

Join the answers together.
Divide and Conquer

```python
findMax(array):
    part1 = lower half of array
    part2 = upper half of array
    max1 = findMax(part1)
    max2 = findMax(part2)
    return max(max1, max2)
```

Determining the asymptotic cost of this algorithm is **difficult**.

Because the algorithm is recursive, the total cost is the **sum of all calls**.

The call stack might be very deep.
Thinking About Recursive Calls

Let's draw a picture of a recursive algorithm.
Thinking About Recursive Calls

We split our original dataset into two pieces.

split/join
Thinking About Recursive Calls

Then the pieces themselves are split again...over and over.
Thinking About Recursive Calls

split/join
Thinking About Recursive Calls

Usually, this is not the actual order in which the operations happen.

But often, it's a good way to analyze the cost.
Divide and Conquer

findMax(array):

    part1 = lower half of array
    part2 = upper half of array
    max1 = findMax(part1)
    max2 = findMax(part2)
    return max(max1, max2)

• What is the split cost for this method?
• How many times does it recurse, and how large are the parts?
• What is the join cost?
**Divide and Conquer**

```python
findMax(array):
    part1 = lower half of array
    part2 = upper half of array
    max1 = findMax(part1)
    max2 = findMax(part2)
    return max(max1, max2)
```

Sometimes, a “split in half” algorithm takes $O(n)$ time to split the data.
Divide and Conquer

findMax(array):
    part1 = lower half of array
    part2 = upper half of array
    max1 = findMax(part1)
    max2 = findMax(part2)
    return \text{max}(\text{max1}, \text{max2})

Our join cost is $O(1)$. 
Divide and Conquer

findMax(array):

part1 = lower half of array
part2 = upper half of array
max1 = findMax(part1)
max2 = findMax(part2)
return max(max1,max2)

split+join is $\Theta(n) + \Theta(1) = \Theta(n)$. 
Divide and Conquer

\begin{verbatim}
findMax(array):
    part1 = lower half of array
    part2 = upper half of array
    max1 = findMax(part1)
    max2 = findMax(part2)
    return max(max1, max2)
\end{verbatim}

We recurse twice, and each of the pieces is half of the original size.
Figuring the Costs, Per Layer

- The total split cost per layer is:
  \[ \text{numSplits} \times \text{costPerSplit} \]

- For simple binary algorithms like this one, at layer \( L \):
  \[ \text{numSplits} = 2^L \]

  \[ \text{costPerSplit} = c_0 \frac{n}{2^L} \]

  where \( c_0 \) is the constant on the split cost.
Figuring the Costs, Per Layer

• Thus, the total cost **per layer** is:

\[ 2^L \frac{c_0 n}{2^L} = c_0 n = \Theta(n) \]

• So every layer has the same cost
  - But only if the **number of splits** equals **the divisor**
Thinking About Recursive Calls

\[ \Theta(n) \]

How many layers are there?

\[ \Theta(n) \]

\[ \Theta(n) \]

\[ \Theta(n) \]

How many layers are there?
Thinking About Recursive Calls

\[ \Theta(n) \]

\[ \Theta(n) \]

\[ \Theta(n) \]

\[ \Theta(\log n) \]
Thinking About Recursive Calls

Only one thing is missing: we have not yet dealt with the base case.

When does the recursion end, and what is the cost?
Divide and Conquer

findMax(array):

    if (array.length <= 8)
        return specialCase(array)

    part1 = lower half of array
    part2 = upper half of array
    max1 = findMax(part1)
    max2 = findMax(part2)
    return max(max1, max2)

The base case generally assumes:
- Happens at fixed size
- Takes \( \Theta(1) \) time
Thinking About Recursive Calls

\( \Theta(n) \)

\( \Theta(n) \)

\( \Theta(n) \)

\( \Theta(1) \) \( \Theta(1) \) \( \Theta(1) \) \( \Theta(1) \) \( \Theta(1) \) \( \Theta(1) \) \( \Theta(1) \) \( \Theta(1) \)
The Base Layer

- The base layer has $2^{\log n}$ nodes

- Thus, it has $\Theta(n)$ nodes

- Thus, the layer has total cost of $\Theta(n) \times \Theta(1) = \Theta(n)$
Thinking About Recursive Calls

Thus, each layer has the same cost – but only when splits=divisor!

Total cost in this example is: $\Theta(n \lg n)$
Group Exercise:

Write a recurrence to represent the cost of this algorithm.

doThing(array):

    // split takes $\Theta(n^2)$ time
    part1 = 2/3 of array
    part2 = 2/3 of array
    part3 = 2/3 of array
    ret1 = doThing(part1)
    ret2 = doThing(part2)
    ret3 = doThing(part3)
    // join() is $\Theta(n)$
    return join(ret1, ret2, ret3)
\[ T(n) = 3 T\left(\frac{2n}{3}\right) + \Theta(n^2) + \Theta(n) \]

\[ T(n) = 3 T\left(\frac{2n}{3}\right) + \Theta(n^2) \]

doThing(array):

// split takes \( \Theta(n^2) \) time
part1 = 2/3 of array
part2 = 2/3 of array
part3 = 2/3 of array
ret1 = doThing(part1)
ret2 = doThing(part2)
ret3 = doThing(part3)

// join() is \( \Theta(n) \)
return join(ret1, ret2, ret3)

This is non-trivial to solve!
Recurrences

- Divide and Conquer
- **What is a Recurrence?**
- The Repeated Substitution Method
- The Recursion-Tree Method
- The Substitution Method
- The Master Method
What is a Recurrence?

• A “recurrence” is:
  – An equality (or inequality) describing cost
  – Defined recursively
  – Piecewise definition to handle base case

\[
T(n) = \begin{cases} 
  \Theta(1) & \text{if } n = 1 \\
  2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1
\end{cases}
\]
What is a Recurrence?

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 
\end{cases} \]

- Two Recursive Calls
- Each Handles Half the Data
- Linear Cost to Split and/or Join the Data
What is a Recurrence?

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 
\end{cases} \]

- Constant time...
- ...for the base case(s)
Things We Generally Ignore

• Floor/Ceiling
  – Recursive calls are almost balanced, but maybe not exactly

• Imply the base case
  – Since it's going to be \( \Theta(1) \) anyway!
  – Note: Base case may apply to more than just \( n=1 \)

\[
T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)
\]
Generalizing

- Lots of different recurrences are possible

\[ T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^2) \]
\[ T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n) \]
\[ T(n) = 2T\left(\frac{9n}{10}\right) + \Theta(lg\ n) \]
\[ T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \]  \hspace{1cm} \text{(binary search)}
\[ T(n) = 2T(n-1) + \Theta(1) \]  \hspace{1cm} \text{(boolean satifiability)}
Recurrences

- Divide and Conquer
- What is a Recurrence?
- The Repeated Substitution Method
- The Recursion-Tree Method
- The Substitution Method
- The Master Method
Repeated Substitution

- We've already seen **repeated substitution** in previous slides:
  - Expand each recursive call in the recurrence
  - Look for a pattern, prove it correct

**Group Exercise:**

Write the recurrence for Binary Search; then solve it using repeated substitution.

Can you do it without your notes? (If not, check your notes to remind yourself.)
Repeated Substitution (review)

- **Split step:**
  - Constant time to “check the middle”
- **Recursion:**
  - Recurse (once) into half the data
- **Join step:**
  - Nothing to do!

\[
T(n) = T\left(\frac{n}{2}\right) + \Theta(1)
\]
Repeated Substitution (review)

\[
T(n) = T\left(\frac{n}{2}\right) + \Theta(1)
\]

\[
T(n) = T\left(\frac{n}{4}\right) + \Theta(1) + \Theta(1)
\]

\[
T(n) = T\left(\frac{n}{8}\right) + \Theta(1) + \Theta(1) + \Theta(1)
\]

\[
\vdots
\]

\[
T(n) = T\left(\frac{n}{2^k}\right) + k \cdot \Theta(1)
\]
Repeated Substitution (review)

\[ T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \]

\[ T(n) = T\left(\frac{n}{2^k}\right) + k \times \Theta(1) \]

\[ n = 2^k \]

\[ k = \lg n \]

\[ T(n) = T(1) + \lg n \times \Theta(1) \]

\[ T(n) = \Theta(\lg n) \]
Thus, by repeated substitution, we know that if

\[ T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \]

then

\[ T(n) = \Theta(lg \ n) \]
Recurrences

- Divide and Conquer
- What is a Recurrence?
- The Repeated Substitution Method
- The Recursion-Tree Method
- The Substitution Method
- The Master Method
The Recursion Tree Method

• To solve a recurrence using a recursion tree:
  – Show the cost of each recursive call at each layer
  – Sum each layer
  – Sum the layers
Building a Recursion Tree

\[ T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \]

Suppose that we want to solve the recursion at the right.
We are going to need to be specific about the scaling constant, so we rewrite the recurrence.

This conversion is \textit{approximate}, so our solution will also be an approximation.
Building a Recursion Tree

There are $n$ elements.
The overall cost of the entire tree is $T(n)$.

\[ T(n) = 2T\left(\frac{n}{2}\right) + cn \]
Assume that $n$ is large, and thus $T(n)$ will recurse.

\[ T(n) = 2T\left(\frac{n}{2}\right) + cn \]
Building a Recursion Tree

The per-level cost is $c \cdot n$

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n$$
Building a Recursion Tree

There are two recursive calls.

Each call has size \( \frac{n}{2} \)

\[
T(n) = 2T\left(\frac{n}{2}\right) + cn
\]
Building a Recursion Tree

We recurse to a third level, using the same logic.

\[ T(n) = 2T\left(\frac{n}{2}\right) + cn \]
Building a Recursion Tree

Recursion continues, down many levels.

\[ T(n) = 2T\left(\frac{n}{2}\right) + cn \]
Building a Recursion Tree

Total up each layer.

The algebra at this step can be quite painful, if we have a non-trivial recurrence!

\[ T(n) = 2T\left(\frac{n}{2}\right) + cn \]
Building a Recursion Tree

Now we need to consider the leaves of the tree!

Each leaf costs $O(1)$. But how many are there? (Don't assume $n$ — that only works in special cases!)

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
Building a Recursion Tree

Use a logarithm to figure out how many layers there are in the tree.

This is $\log_2$ because we divided by 2 at every layer. If we divided by 4, it would be $\log_4$. 

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
Building a Recursion Tree

Use an exponent to express how wide the tree gets...

Again, the base of the exponent is determined by the formula – we had two children at each node, so the formula is $2^x$.

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
Building a Recursion Tree

The number of leaves is \( n \) only in the special case where the constants are equal.

\[
T(n) = 2T\left(\frac{n}{2}\right) + cn
\]

\[
2^{\log_2 n} = n
\]
Building a Recursion Tree

Now, we sum the layers:

- Each of the upper layers
- Plus the leaves

\[ T(n) = 2T\left(\frac{n}{2}\right) + cn \]

Overall total: \( O(n \log n) \)
Building a Recursion Tree

• What's the point? We had already solve this recurrence...

• We can generalize to harder recurrences!
  - What is the solution to this recurrence?

\[
T(n) = 2T\left(\frac{9n}{10}\right) + \Theta(lg\ n)
\]
Building a Recursion Tree

\[ T(n) = 2T\left(\frac{9n}{10}\right) + c \log n \]
Building a Recursion Tree

\[ T(n) = 2T\left(\frac{9n}{10}\right) + c \lg n \]
Building a Recursion Tree

\[ T(n) = 2T\left(\frac{9n}{10}\right) + c \log n \]
Building a Recursion Tree

The algebra is nasty, but we can show that each node in the tree is \(\leq c \ lg \ n\).

\[
T(n) = 2T\left(\frac{9n}{10}\right) + c \ lg \ n
\]
Building a Recursion Tree

The base for this logarithm is $10/9$, **not** 2 – because in each layer, we divide $n$ by $10/9$.

$$T(n) = 2T\left(\frac{9n}{10}\right) + c \log n$$
Building a Recursion Tree

The base for the exponent is 2.

It turns out that there are approximately $n^{6.5}$ leaf nodes.

This cost completely dominates the entire tree.

\[ T(n) = 2T\left(\frac{9n}{10}\right) + c \log n \]

\[ 2^{\log_{10/9} n} = n^{\log_{10/9} 2} \approx n^{6.5} \]
Building a Recursion Tree

\[ T(n) = 2T\left(\frac{9n}{10}\right) + c \log n \]

\[ T(n) = \Theta\left(n^{\log_{10/9} 2}\right) \]
Is a Recursion Tree Enough?

- We used an approximation in our recursion tree
- Do we trust our result?

- We could solve it formally
Topic 04: Recurrences

- Divide and Conquer
- What is a Recurrence?
- The Recursion-Tree Method
- The Substitution Method
- The Master Method
The Substitution Method

- To solve a recurrence using the substitution method:
  - Suggest an inequality as the solution for the recurrence
  - Prove it valid with induction
The Substitution Method

The Recurrence:

\[ T(n) = 2T\left(\frac{n}{2}\right) + n \]

The Conjecture:

\[ T(n) \leq c \times n \lg n, \quad n \geq 2 \]

This is where our slides STOP on this topic. These proofs are long and painful, so we'll skip them in this class.

You can read CLRS if you would like to see examples of these proofs in practice.

In this class, we'll focus instead on the Master Method.
The Substitution Method

The Recurrence:

\[ T(n) = 2T\left(\frac{n}{2}\right) + n \]

The Conjecture:

\[ T(n) \leq c \cdot n \log n, n \geq 2 \]

Can anybody see why we have \( n \geq 2 \) in our conjecture?
The Substitution Method

Base Case (n=2):

When \( n=2 \), we split the array into two parts, each of size \( 1 \), and then recurse into each. Each of these calls is the base case, and takes \( \Theta(1) \) time.

Thus, there exists some constant \( c \) such that the total cost (split, 2x recurse, 2x base, join) is

\[
T(n) \leq c \cdot n \lg n = 2c
\]
The Substitution Method

Inductive Case:

Assume that the conjecture holds for some constant $c$, for all $n \leq k$. We will attempt to prove that it holds for all $n \leq 2k$, for the same constant $c$.

By the definition of the recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(2k) = 2T(k) + 2k$$
The Substitution Method

Inductive Case (cont):

By the I.H.:

\[ T(2k) \leq 2\left(c \cdot k \log k\right) + 2k \]

\[ = c \cdot 2^k \left(\frac{1}{c} + \log k\right) \]

Assuming \( c \geq 1 \):

\[ \leq c \cdot 2^k \left(1 + \log k\right) \]

\[ \leq c \cdot 2^k \log 2^k \]
The Substitution Method

Inductive Case (cont):
In the previous slide, we had to assume that
\[ c \geq 1. \]
It is safe to assume this because there are no other assumptions which force \( c \) to be small.

Thus, the inductive step holds.
Thus, the conjecture holds for \( n \geq 2 \).
The Substitution Method

Substitution gives us a way to formally prove the solution to recurrences

But it's hard to use!

Is there an easier way?
Recurrences

• Divide and Conquer
• What is a Recurrence?
• The Recursion-Tree Method
• The Substitution Method
• The Master Method
The Master Method

Recursion trees and induction is hard!

Isn't there some easier way?

Welcome to the Master Method!
The Master Method

- Master Method is a theorem which gives mechanical answers for many common recurrences
- Requires that the recurrence be in a standard form

\[ T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \quad a \geq 1, \quad b > 1 \]

- Has three cases, to handle common scenarios
The Master Method

\[ T(n) = a \, T\left(\frac{n}{b}\right) + f(n) \]

- Step 1: Calculate \( \log_b a \)
- Step 2: Compare \( f(n) \) to \( n^{\log_b a} \)
  - Hopefully, they are equal – or different by a polynomial factor.
- Step 3: Find the appropriate case
The Intuition Behind the Master Method

This is the general recursion tree for $T(n)$.

The value of $n$ goes down by powers of $b$. 
The Intuition Behind the Master Method

The width of the tree grows as powers of $a$. 

\[
\begin{array}{cccccc}
O(1) & O(1) & O(1) & O(1) & O(1) & O(1) & O(1)
\end{array}
\]
The Intuition Behind the Master Method

That familiar polynomial of the Master Method is the number of leaf nodes!

\[ f(n) \]

\[ f\left(\frac{n}{b}\right) \]

\[ f\left(\frac{n}{b^2}\right) \]

\[ a^{\log_b n} = n^{\log_b a} \]

\[ O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \]
The Intuition Behind the Master Method

If there are few leaves and/or \( f(n) \) is large...

...then the \( f(n) \) values in the upper nodes dominate the recurrence.

\[
\begin{align*}
\underbrace{f(n)} & \quad \underbrace{f(n)} \\
\underbrace{f(n/b^2)} & \quad \underbrace{f(n/b^2)} \\
\underbrace{f(n/b^2)} & \quad \underbrace{f(n/b^2)}
\end{align*}
\]

\[
a \log_b n = n \log_b a
\]
The Intuition Behind the Master Method

If there are many leaves and/or $f(n)$ is small...

...then the leaves dominate the recurrence.

If $f(n)$ is small compared to $n^{log_b a}$...

...then the leaves dominate the recurrence.

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

$O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1)$
The Intuition Behind the Master Method

If there is balance... ...then all layers contribute equally.

\[ f(n) \]

\[ f\left(\frac{n}{b}\right) \quad f\left(\frac{n}{b}\right) \]

\[ f\left(\frac{n}{b^2}\right) \quad f\left(\frac{n}{b^2}\right) \quad f\left(\frac{n}{b^2}\right) \quad f\left(\frac{n}{b^2}\right) \]

\[ \alpha = \log_b n = n \log_b \alpha \]

\[ O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \quad O(1) \]
The Master Method, Case 1

\[ T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \]

- Case 1: \( f(n) = O\left(n^{\log_b a - \epsilon}\right), \epsilon > 0 \)

- Solution: \( T(n) = \Theta\left( n^{\log_b a} \right) \)

\( f(n) \) is smaller than the target function, by at least a polynomial factor.

Thus, the leaves dominate.
The Master Method, Case 2

\[ T(n) = a \, T\left(\frac{n}{b}\right) + f(n) \]

- **Case 2:** \[ f(n) = \Theta(n^{\log_b a}) \]

- **Solution:** \[ T(n) = \Theta\left(n^{\log_b a \, \lg n}\right) \]

The leaves and upper nodes are balanced, and thus contribute equally.

The cost is the cost of each layer, times the number of layers.
The Master Method, Case 3

\[ T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \]

- **Case 3:** \( f(n) = \Omega\left(n^{\log_b a + \epsilon}\right), \epsilon > 0 \)
  \[ a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n), c < 1, \text{ for all large } n \]

- **Solution:** \( T(n) = \Theta\left(f(n)\right) \)

Thus, \( f(n) \) dominates the recurrence.

\( f(n) \) is **larger** than the target function, by **at least** a polynomial factor.

\( f(n) \) gets smaller as you reduce \( n \).
Master Method Examples

\[ T(n) = 2T\left(\frac{n}{2}\right) + n^2 \]

\[ T(n) = 4T\left(\frac{n}{4}\right) + n \lg n \]

\[ T(n) = 3T\left(\frac{n}{4}\right) + n \]

\[ T(n) = 2T\left(\frac{9n}{10}\right) + \lg n \]

\[ T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \]

\[ T(n) = 2T(n-1) + \Theta(1) \]

Let's work on these in class.

Not all of these can be solved by the Master Method!
Master Method Examples

\[ T(n) = 2T\left(\frac{n}{2}\right) + n^2 \]

\[ a = ? \]
\[ b = ? \]
\[ \log_b a = ? \]
\[ n^{\log_b a} = ? \]

Which case?

? 

\[ T(n) = ? \]
Master Method Examples

\[ T(n) = 2T\left(\frac{n}{2}\right) + n^2 \]

\[ a = 2 \]
\[ b = 2 \]
\[ \log_b a = 1 \]
\[ n^{\log_b a} = n \]

Case 3

\[ f(n) = \Omega(n^{1+\epsilon}), \epsilon > 0 \]

\[ T(n) = \Theta(n^2) \]
Master Method Examples

\[ T(n) = 4T\left(\frac{n}{4}\right) + n \log n \]

\[ a = ? \]
\[ b = ? \]
\[ \log_b a = ? \]
\[ n^{\log_b a} = ? \]

Which case?

? 

\[ T(n) = ? \]
Master Method Examples

\[ T(n) = 4T\left(\frac{n}{4}\right) + n \log n \]

- \( a = 4 \)
- \( b = 4 \)
- \( \log_b a = 1 \)
- \( n^{\log_b a} = n \)

Not polynomially different
Master Method Examples

\[ T(n) = 3T\left(\frac{n}{4}\right) + n \]

\[ a = ? \]
\[ b = ? \]
\[ \log_b a = ? \]
\[ n^{\log_b a} = ? \]

Which case?

? 

\[ T(n) = ? \]
Master Method Examples

\[ T(n) = 3T\left(\frac{n}{4}\right) + n \]

\[ a = 3 \]
\[ b = 4 \]
\[ \log_b a = \log_4 3 \approx 0.79 \]
\[ n^{\log_b a} \approx n^{0.79} \]

Case 3

\[ f(n) = \Omega(n^{0.79 + \epsilon}), \epsilon > 0 \]

\[ T(n) = \Theta(n) \]
Master Method Examples

\[ T(n) = 2T\left(\frac{9n}{10}\right) + \lg n \]

\[ a = ? \]
\[ b = ? \]
\[ \log_b a = ? \]
\[ n^{\log_b a} = ? \]

Which case?

? 

\[ T(n) = ? \]
Master Method Examples

\[ T(n) = 2T\left(\frac{9n}{10}\right) + \lg n \]

\[ a = 2 \]
\[ b = 10/9 \]
\[ \log_b a = \log_{10/9} 2 \approx 6.5 \]
\[ n^{\log_b a} \approx n^{6.5} \]

Case 1

\[ f(n) = O\left(n^{6.5-\epsilon}\right), \epsilon > 0 \]

\[ T(n) \approx \Theta\left(n^{6.5}\right) \]
Master Method Examples

\[ T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \]

\[ a = ? \]
\[ b = ? \]
\[ \log_b a = ? \]
\[ n^{\log_b a} = ? \]

Which case?

? 

\[ T(n) = ? \]
Master Method Examples

\[ T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \]

- \( a = 1 \)
- \( b = 2 \)
- \( \log_b a = 0 \)
- \( n^{\log_b a} = n^0 = 1 \)

Case 2

\[ f(n) = \Theta(n^0) \]

\[ T(n) = \Theta(n^0 \log n) = \Theta(\log n) \]
Master Method Examples

\[ T(n) = 2T(n-1) + \Theta(1) \]

\[ a = ? \]
\[ b = ? \]
\[ \log_b a = ? \]
\[ n^{\log_b a} = ? \]

Which case?

? 

\[ T(n) = ? \]
The recurrence is not in the standard form, and so the Master Method doesn't apply!

Can we still solve it?
Master Method Examples

\[ T(n) = 2T(n-1) + \Theta(1) \]

- \[ T(n) \leq 2T(n-1) + cn \]
- \[ T(n) \leq 2^2 T(n-2) + 3cn \]
- \[ T(n) \leq 2^3 T(n-3) + 7cn \]
- \[ T(n) \leq 2^4 T(n-4) + 15cn \]

\[ T(n) \leq 2^k (T(n-k) + cn) - cn \]
- \[ T(n) \leq 2^n (cn) - cn \]
- \[ T(n) = \Theta(2^n) \]
Recurrences

• Divide and Conquer
• What is a Recurrence?
• The Repeated Substitution Method
• The Recursion-Tree Method
• The Substitution Method
• The Master Method

Summary