Recurrences and Inequalities

- Often easier to prove that a recurrence is no more than some quantity than to prove that it equals something
- Consider: $f(n) = f(n - 1) + f(n - 2)$, $f(1) = f(2) = 1$
- "Guess" that $f(n) \leq 2^n$

Inequalities (II)

Goal: Prove by induction that for $f(n) = f(n - 1) + f(n - 2)$, $f(1) = f(2) = 1$, $f(n) \leq 2^n$

- Base case: $f(1) = 1 \leq 2^1$, $f(2) = 1 \leq 2^2$
- Inductive hypothesis: For all $j < n$, $f(j) \leq 2^j$
- Inductive step:
  \[
  f(n) = f(n - 1) + f(n - 2) \leq 2^{n-1} + 2^{n-2} < 2 \cdot 2^{n-1} = 2^n
  \]
Recursion-tree method

- Each node represents the cost of a single subproblem in a recursive call
- First, we sum the costs of the nodes in each level of the tree
- Then, we sum the costs of all of the levels

Example 1

- Consider the recurrence for the running time of Mergesort:
  \[ T(n) = 2T(n/2) + n, \quad T(1) = O(1) \]

- We can see that each level of the tree sums to \( n \)
- Further the depth of the tree is \( \log n \) (\( n/2^d = 1 \) implies that \( d = \log n \)).
- Thus there are \( \log n + 1 \) levels each of which sums to \( n \)
- Hence \( T(n) = \Theta(n \log n) \)
• Let's solve the recurrence $T(n) = 3T(n/4) + n^2$
• Note: For simplicity, from now on, we'll assume that $T(i) = \Theta(1)$ for all small constants $i$. This will save us from writing the base cases each time.

$\log_4 n \sum_{i=0}^{\log_4 n} (3/16)^i n^2$ (5)

< $n^2 \sum_{i=0}^{\infty} (3/16)^i$ (6)

= $\frac{1}{1 - (3/16)} n^2$ (7)

= $O(n^2)$ (8)

• We can see that the $i$-th level of the tree sums to $(3/16)^i n^2$.
• Further the depth of the tree is $\log_4 n$ ($n/4^d = 1$ implies that $d = \log_4 n$)
• So we can see that $T(n) = \sum_{i=0}^{\log_4 n} (3/16)^i n^2$

$T(n) = aT(n/b) + f(n)$ (9)

• Divide and conquer algorithms often give us running-time recurrences of the form

Where $a$ and $b$ are constants and $f(n)$ is some other function.

• The so-called “Master Method” gives us a general method for solving such recurrences when $f(n)$ is a simple polynomial.
Master Theorem

- Unfortunately, the Master Theorem doesn’t work for all functions $f(n)$
- Further many useful recurrences don’t look like $T(n)$
- However, the theorem allows for very fast solution of recurrences when it applies

Master Theorem

- Master Theorem is just a special case of the use of recursion trees
- Consider equation $T(n) = aT(n/b) + f(n)$
- We start by drawing a recursion tree

The Recursion Tree

- The root contains the value $f(n)$
- It has $a$ children, each of which contains the value $f(n/b)$
- Each of these nodes has $a$ children, containing the value $f(n/b^2)$
- In general, level $i$ contains $a^i$ nodes with values $f(n/b^i)$
- Hence the sum of the nodes at the $i$-th level is $a^i f(n/b^i)$

Details

- The tree stops when we get to the base case for the recurrence
- We’ll assume $T(1) = f(1) = \Theta(1)$ is the base case
- Thus the depth of the tree is $\log_b n$ and there are $\log_b n + 1$ levels
Recursion Tree

- Let $T(n)$ be the sum of all values stored in all levels of the tree:
  
  $$T(n) = f(n) + a f(n/b) + a^2 f(n/b^2) + \cdots + a^L f(n/b^L)$$
  
  - Where $L = \log_b n$ is the depth of the tree
  - Since $f(1) = \Theta(1)$, the last term of this summation is $\Theta(a^L) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$

A "Log Fact" Aside

- It’s not hard to see that $a^{\log_b n} = n^{\log_b a}$
  
  $$a^{\log_b n} = n^{\log_b a}$$  (10)
  
  $$a^{\log_b n} = a^{\log_a n \cdot \log_b a}$$  (11)
  
  $$\log_b n = \log_a n \cdot \log_b a$$  (12)
  
  - We get to the last eqn by taking $\log_a$ of both sides
  - The last eqn is true by our third basic log fact

Master Theorem

- We can now state the Master Theorem
- We will state it in a way slightly different from the book
- Note: The Master Method is just a “short cut” for the recursion tree method. It is less powerful than recursion trees.

Master Method

The recurrence $T(n) = aT(n/b) + f(n)$ can be solved as follows:

- If $af(n/b) \leq Kf(n)$ for some constant $K < 1$, then $T(n) = \Theta(f(n))$.
- If $af(n/b) \geq Kf(n)$ for some constant $K > 1$, then $T(n) = \Theta(n^{\log_b a})$.
- If $af(n/b) = f(n)$, then $T(n) = \Theta(f(n) \log_b n)$. 
Proof

• If \( f(n) \) is a constant factor larger than \( a f(n/b) \), then the sum is a descending geometric series. The sum of any geometric series is a constant times its largest term. In this case, the largest term is the first term \( f(n) \).

• If \( f(n) \) is a constant factor smaller than \( a f(n/b) \), then the sum is an ascending geometric series. The sum of any geometric series is a constant times its largest term. In this case, this is the last term, which by our earlier argument is \( \Theta(n \log_b a) \).

• Finally, if \( a f(n/b) = f(n) \), then each of the \( L + 1 \) terms in the summation is equal to \( f(n) \).

Example

• Karatsuba’s multiplication algorithm: \( T(n) = 3T(n/2) + n \)
  • If we write this as \( T(n) = aT(n/b) + f(n) \), then \( a = 3, b = 2, f(n) = n \)
  • Here \( a f(n/b) = 3n/2 \) is bigger than \( f(n) = n \) by a factor of \( 3/2 \), so \( T(n) = \Theta(n \log_2 3) \)

Example

• Mergesort: \( T(n) = 2T(n/2) + n \)
  • If we write this as \( T(n) = aT(n/b) + f(n) \), then \( a = 2, b = 2, f(n) = n \)
  • Here \( a f(n/b) = f(n) \), so \( T(n) = \Theta(n \log n) \)
**Example**

- \( T(n) = T(n/2) + n \log n \)
- If we write this as \( T(n) = aT(n/b) + f(n) \), then \( a = 1, b = 2, f(n) = n \log n \)
- Here \( a f(n/b) = n/2 \log n/2 \) is smaller than \( f(n) = n \log n \) by a constant factor, so \( T(n) = \Theta(n \log n) \)

**In-Class Exercise**

- Consider the recurrence: \( T(n) = 4T(n/2) + n \log n \)
- Q: What is \( f(n) \) and \( a f(n/b) \)?
- Q: Which of the three cases does the recurrence fall under (when \( n \) is large)?
- Q: What is the solution to this recurrence?

**In-Class Exercise**

- Consider the recurrence: \( T(n) = 2T(n/4) + n \log n \)
- Q: What is \( f(n) \) and \( a f(n/b) \)?
- Q: Which of the three cases does the recurrence fall under (when \( n \) is large)?
- Q: What is the solution to this recurrence?

**Take Away**

- Recursion tree and Master method are good tools for solving many recurrences
- However these methods are limited (they can’t help us get guesses for recurrences like \( f(n) = f(n - 1) + f(n - 2) \))
- For info on how to solve these other more difficult recurrences, review the notes on annihilators on the class web page.
Intro to Annihilators

• Suppose we are given a sequence of numbers \( A = \langle a_0, a_1, a_2, \cdots \rangle \)
• This might be a sequence like the Fibonacci numbers
• I.e. \( A = \langle a_0, a_1, a_2, \cdots \rangle = (T(1), T(2), T(3), \cdots) \)

Annihilator Operators

We define three basic operations we can perform on this sequence:

1. Multiply the sequence by a constant: \( cA = \langle ca_0, ca_1, ca_2, \cdots \rangle \)
2. Shift the sequence to the left: \( LA = \langle a_1, a_2, a_3, \cdots \rangle \)
3. Add two sequences: if \( A = \langle a_0, a_1, a_2, \cdots \rangle \) and \( B = \langle b_0, b_1, b_2, \cdots \rangle \), then \( A + B = \langle a_0 + b_0, a_1 + b_1, a_2 + b_2, \cdots \rangle \)

Annihilator Description

• We first express our recurrence as a sequence \( T \)
• We use these three operators to “annihilate” \( T \), i.e. make it all 0’s
• Key rule: can’t multiply by the constant 0
• We can then determine the solution to the recurrence from the sequence of operations performed to annihilate \( T \)

Example

• Consider the recurrence \( T(n) = 2T(n - 1), T(0) = 1 \)
• If we solve for the first few terms of this sequence, we can see they are \( \langle 2^0, 2^1, 2^2, 2^3, \cdots \rangle \)
• Thus this recurrence becomes the sequence:
\[ T = \langle 2^0, 2^1, 2^2, 2^3, \cdots \rangle \]
Let's annihilate $T = \langle 2^0, 2^1, 2^2, 2^3, \ldots \rangle$

- Multiplying by a constant $c = 2$ gets: $2T = \langle 2 \cdot 2^0, 2 \cdot 2^1, 2 \cdot 2^2, 2 \cdot 2^3, \ldots \rangle = \langle 2^1, 2^2, 2^3, 2^4, \ldots \rangle$
- Shifting one place to the left gets $LT = \langle 2^1, 2^2, 2^3, 2^4, \ldots \rangle$
- Adding the sequence $LT$ and $-2T$ gives: $LT - 2T = \langle 2^1 - 2^1, 2^2 - 2^2, 2^3 - 2^3, \ldots \rangle = \langle 0, 0, 0, \ldots \rangle$
- The annihilator of $T$ is thus $L - 2$

$0$, the “Forbidden Annihilator”

- Multiplication by 0 will annihilate any sequence
- Thus we disallow multiplication by 0 as an operation
- In particular, we disallow $(c - c) = 0$ for any $c$ as an annihilator
- Must always have at least one $L$ operator in any annihilator!

An annihilator annihilates exactly one type of sequence
- In general, the annihilator $L - c$ annihilates any sequence of the form $\langle a_0 c^n \rangle$
- If we find the annihilator, we can find the type of sequence, and thus solve the recurrence
- We will need to use the base case for the recurrence to solve for the constant $a_0$
Example

If we apply operator \((L - 3)\) to sequence \(T\) above, it fails to annihilate \(T\):
\[
(L - 3)T = LT + (-3)T
= \langle 2^1, 2^2, 2^3, \cdots \rangle + (-3 \times 2^0, -3 \times 2^1, -3 \times 2^2, \cdots)
= \langle (2 - 3) \times 2^0, (2 - 3) \times 2^1, (2 - 3) \times 2^2, \cdots \rangle
= (2 - 3)T = -T
\]

Example (II)

What does \((L - c)\) do to other sequences \(A = \langle a_0d^n \rangle\) when \(d \neq c\)?:
\[
(L - c)A = (L - c)(a_0, a_0d, a_0d^2, a_0d^3, \cdots)
= L(a_0, a_0d, a_0d^2, a_0d^3, \cdots) - c(a_0, a_0d, a_0d^2, a_0d^3, \cdots)
= \langle a_0d, a_0d^2, a_0d^3, \cdots \rangle - \langle ca_0, ca_0d, ca_0d^2, ca_0d^3, \cdots \rangle
= \langle a_0d - ca_0, a_0d^2 - ca_0d, a_0d^3 - ca_0d^2, \cdots \rangle
= ((d - c)a_0, (d - c)a_0d, (d - c)a_0d^2, \cdots)
= (d - c)(a_0, a_0d, a_0d^2, \cdots)
= (d - c)A
\]

Uniqueness

- The last example implies that an annihilator annihilates one type of sequence, but does not annihilate other types of sequences.
- Thus Annihilators can help us classify sequences, and thereby solve recurrences.

Lookup Table

- The annihilator \(L - a\) annihilates any sequence of the form \(\langle c_1a^n \rangle\).
First calculate the annihilator:

- Recurrence: \( T(n) = 4 \times T(n - 1), \ T(0) = 2 \)
- Sequence: \( T = \langle 2, 2 \times 4, 2 \times 4^2, 2 \times 4^3, \ldots \rangle \)
- Calculate the annihilator:
  - \( LT = \langle 2 \times 4, 2 \times 4^2, 2 \times 4^3, 2 \times 4^4, \ldots \rangle \)
  - \( 4T = \langle 2 \times 4, 2 \times 4^2, 2 \times 4^3, 2 \times 4^4, \ldots \rangle \)
  - Thus \( LT - 4T = \langle 0, 0, 0, \ldots \rangle \)
  - And so \( L - 4 \) is the annihilator

Now use the annihilator to solve the recurrence

- Look up the annihilator in the “Lookup Table”
- It says: “The annihilator \( L - 4 \) annihilates any sequence of the form \( \langle c_1 4^n \rangle \)”
- Thus \( T(n) = c_1 4^n \), but what is \( c_1 \)?
- We know \( T(0) = 2 \), so \( T(0) = c_1 4^0 = 2 \) and so \( c_1 = 2 \)
- Thus \( T(n) = 2 \times 4^n \)

Consider the recurrence \( T(n) = 3 \times T(n - 1), \ T(0) = 3 \)

- Q1: Calculate \( T(0), T(1), T(2) \) and \( T(3) \) and write out the sequence \( T \)
- Q2: Calculate \( LT \), and use it to compute the annihilator of \( T \)
- Q3: Look up this annihilator in the lookup table to get the general solution of the recurrence for \( T(n) \)
- Q4: Now use the base case \( T(0) = 3 \) to solve for the constants in the general solution

HW 1