CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

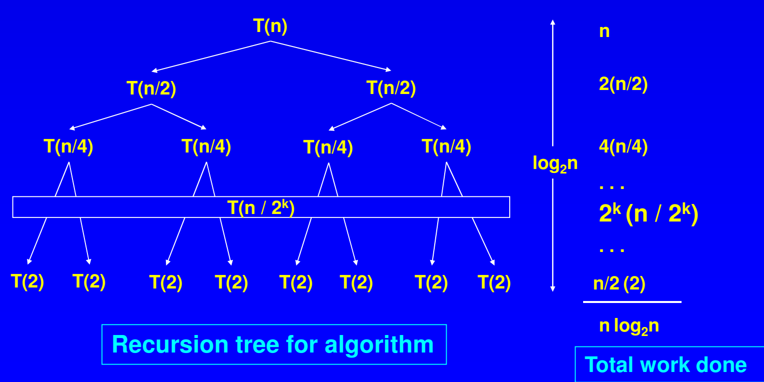
Business

- SA4 out, due Monday, Nov 21
  - Possibility of *very small* SA5 out today too! More on that if it happens
- Expect PS4-Lookahead out soon
  - Likely due after Thanksgiving break
- Project 3 out
  - Deadline extended: Due end of day Nov. 21
  - *Note*: Assume you have constructors and accessors for the four types of PLEs (not, and, or, implies)—similar to IBTs and LLists
    - Document your function names in your project and use them as usual, as we did for IBTs and LLists
- Project 2 Grading update:
  - In progress, but will be slow (catching up from illness may take a while…)
  - Please *meet with me* if you'd like prompt feedback on any part of Project 2!

Recursion-Tree Method

- An example:
  Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Recursion Tree Exercises

- Use the recursion-tree method to solve the following recurrences for n ≥ 1
  - T(n) = 3T(n/3) + n if n ≥ 3; 1 if n < 3
  - T(n) = 4T(n/2) + n if n ≥ 2; 1 if n = 1
  - T(n) = T(n/2) + n if n ≥ 2; 1 if n = 1
  - T(n) = 2T(n/2) + n if n ≥ 2; 1 if n = 1

Note: These three cases are important—we'll come back to them on a later slide!

- Those last three examples illustrate three different cases:
  1. The amount of work per level increases, with the most work done at the leaves of the tree
  2. The amount of work per level decreases, with the most work done at the root
  3. The amount of work per level is constant—and there are (lg n + 1) levels in the tree

There are (lg n + 1) levels in all three cases, really, but it's of particular importance here

A Recurring Observation

- In general, many divide-and-conquer algorithm runtimes may be expressed as recurrences of the form:

$$T(n) = \begin{cases} \theta(1) & \text{if } n \le k \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

- Where:
  - a = number of sub-problems
  - n/b = size of a sub-problem
  - $D(n)$ = time to divide the problem into sub-problems
  - $C(n)$ = time to re-combine the sub-problem solutions

$D(n) + C(n)$ might be represented as a single function $f(n)$, i.e., work done at each node in a recursion tree

See Section 4.5 of CLRS

Master Method

- In many common cases, there is a "cookbook" solution available, using the Master Theorem
- Master Theorem:
  - Let $a \geq 1$, $b > 1$ be constants, $f(n)$ be a function (asymp. positive),…
  - and $T(n)$ be defined by **$T(n) = a*T(n/b) + f(n)$** (on non-neg. integers)
  - Then, $T(n)$ can be bounded asymptotically as follows:

1. $T(n) = \theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a} \lg n)$ if $f(n) = \theta(n^{\log_b a})$
3. $T(n) = \theta(f(n))$ if $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$ and if $a*f(n/b) \leq c*f(n)$ for some constant $0 < c < 1$ and all sufficiently large n.

Master Terminology

- Master Theorem (slightly abridged / elided):
  - Let $a \geq 1$, $b > 1$, $f(n)$ be a function, $T(n) = a*T(n/b) + f(n)$; then

1. $T(n) = \theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a} \lg n)$ if $f(n) = \theta(n^{\log_b a})$
3. $T(n) = \theta(f(n))$ if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and if $a*f(n/b) \leq c*f(n)$ for some constant $0 < c < 1$ and all sufficiently large n.

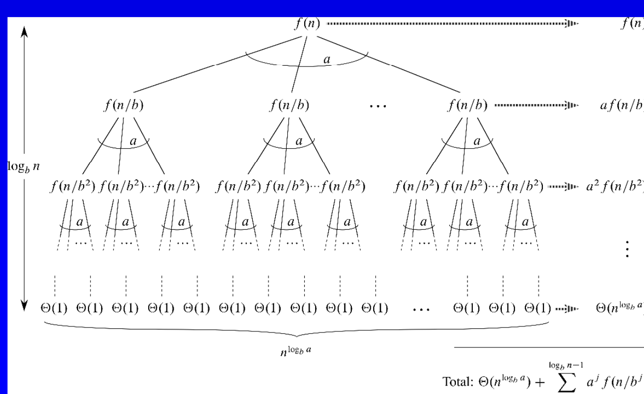
- Note: Comparison of $f(n)$ with $n^{\log_b a}$ (or $\theta(n^{\log_b a})$)
  - case 1: $f(n)$ is *polynomially smaller than* $n^{\log_b a}$
  - case 2: $f(n)$ is *asymptotically equal to* $n^{\log_b a}$
  - case 3: $f(n)$ is *polynomially larger than* $n^{\log_b a}$
- Also case 3: *regularity condition*: $a*f(n/b) \leq c*f(n)$ for $0 < c < 1$ (etc.)
  - Intuition: Amount of work goes down with each recursive call

i.e., number of times dividing *a* by *b* to get O(1)

Master Key

See Section 4.6 for a proof

- Some intuition behind the Master Theorem



This isn't obvious just by looking at it!

Please talk with me outside of class if you'd like to understand this in detail!

Incomplete Mastery

- Master Theorem (slightly abridged / elided):
  - Let $a \geq 1$, $b > 1$, $f(n)$ be a function, $T(n) = a*T(n/b) + f(n)$; then

1. $T(n) = \theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a}\lg n)$ if $f(n) = \theta(n^{\log_b a})$
3. $T(n) = \theta(f(n))$ if $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$ and if $a*f(n/b) \leq c*f(n)$ for some constant $0 < c < 1$ and all sufficiently large n.

- Note: The three cases are not exhaustive! E.g.,
  - $f(n)$ may be smaller than $n^{\log_b a}$, but not polynomially smaller (see cases 1,2)
  - $f(n)$ may be larger than $n^{\log_b a}$, but not polynomially larger (see cases 2,3)
- If the function falls into one of these gaps, or if the regularity condition can't be shown to hold, the Master Method can't be used

An Example

Master Theorem (slightly abridged / elided):

1. $T(n) = \theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a}\lg n)$ if $f(n) = \theta(n^{\log_b a})$
3. $T(n) = \theta(f(n))$ if $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$ and if $a*f(n/b) \leq c*f(n)$ for some constant $0 < c < 1$ and all sufficiently large n.

- Example: $T(n) = 9T(n/3) + n$
  - $a = 9$, $b = 3$, $f(n) = n$, $n^{\log_b a} = n^{\log_3 9} = n^2$
  - So, compare $f(n) = n$ with n^2: $n = O(n^{2-\varepsilon})$
  - … Thus, case 1 applies: $T(n) = \theta(n^2)$

i.e., f(n) is polynomially smaller than $n^{\log_b a}$

- Example: $T(n) = T(2n/3) + 1$

More Examples

Master Theorem (slightly abridged / elided):

1. $T(n) = \theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$
2. $T(n) = \theta(n^{\log_b a} \lg n)$ if $f(n) = \theta(n^{\log_b a})$
3. $T(n) = \theta(f(n))$ if $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$ and if $a*f(n/b) \leq c*f(n)$ for some constant $0 < c < 1$ and all sufficiently large n.

- Examples:
  - $T(n) = 3T(n/4) + n \lg n$
  - $T(n) = 2T(n/2) + n \lg n$
  - $T(n) = 2T(n/2) + \theta(n)$
  - $T(n) = 8T(n/2) + \theta(n^2)$
  - $T(n) = 7T(n/2) + \theta(n^2)$

Design Paradigm	**Analysis**	
	Complexity (Efficiency)	**Correctness**
Iterative	Counting (Exact count of operations / space used)	*Loop invariants*
Recursive	Solving *recurrences*	*Induction*

The green-shaded ones are examples of *polynomial time* classes—upper bounded by n^k for some constant *k*. Problems solvable in polynomial time are considered *tractable*. (More about this later in the semester!)

Time Complexity Classes Illustrated!

Complexity Class	*What we call it*	*Example algorithms / objects*
O(1)	Constant	Print "Hello, World!"; stack operations [and much, much more—be careful!]
O(lg n)	Log time	Binary search
O(n)	Linear	Exhaustive search of an array (linear search); Merge (as used in Mergesort)
O(n lg n)	n lg n	Mergesort; Heapsort [**Recall: sorting can be done in θ(n lg n)**]
O(n^2)	n-squared; quadratic	Insertion / selection / bubble sort; several graph algos
O(n^3)	n-cubed; cubic	*My favorite algorithm*! (a graph algo)
O(2^n)	Exponential	Number of *subsets* of a set of size n
O(n!)	Factorial	Number of orderings / permutations of elements of a list of length n