CSCE5150
Analysis of Computer Algorithms

Dr. Xuan Guo

1

Content

- Asymptotic notation
  - Big Oh
  - Big Omega
  - Big Theta
  - little Oh
  - little Omega
- Asymptotic Dominance
- Recursive algorithms
  - Merge Sort
- Resolving Recurrences
  - Substitution method
  - Recursion trees
  - Master method

2

Problem Solving: Main Steps

1. Problem definition
2. Algorithm design / Algorithm specification
3. Algorithm analysis
4. Implementation
5. Testing
6. [Maintenance]

3

Algorithm Analysis

- Goal
  - Predict the resources that an algorithm requires
- What kind of resources?
  - Memory
  - Communication bandwidth
  - Hardware requirements
  - Running time
- Two approaches
  - Empirical tests
  - Mathematical analysis

4

Algorithm Analysis – Empirical Tests

- Steps:
  - Implement algorithm in a given programming language
  - Measure runtime with several inputs
  - Infer running time for any input
- Pros:
  - No math, straightforward method
- Cons:
  - Not reliable, heavily dependent on
    - the sample inputs
    - programming language and environment
- We want to analyze algorithms to decide whether they are worth implementing

5

Algorithm Analysis – Mathematical Analysis

- Use math to estimate the running time of an algorithm
  - almost always dependent on the size of the input
  - Algorithm1 running time is n^2 for an input of size n
  - Algorithm2 running time is $n \times \log(n)$ for an input of size n
- Pros:
  - formal, rigorous
  - no need to implement algorithms
  - machine-independent
- Cons:
  - math knowledge

6

Algorithm Analysis

- **Best case analysis**
  - shortest running time for any input of size n
  - often meaningless, one can easily cheat
- **Worst case analysis**
  - longest running time for any input of size n
  - it guarantees that the algorithm will not take any longer
  - provides an upper bound on the running time
  - worst case occurs often search for an item that does not exist
- **Average case analysis**
  - running time averaged for all possible inputs
  - it is hard to estimate the probability of all possible inputs

7

Random-Access Machine (RAM)

In order to predict running time, we need a (simple) computational model: the Random-Access Machine (RAM)

- Instructions are executed sequentially
  - No concurrent operations
- Each basic instruction takes a constant amount of time
  - arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling, shift left/shift right
  - data movement: load, store, copy
  - control: conditional/unconditional branch, subroutine call and return
  - Loops and subroutine calls are not simple operations. They depend upon the size of the data and the contents of a subroutine. "Sort" is not a single step operation.
- Each memory access takes exactly 1 step.

We measure the run time of an algorithm by counting the number of steps.

RAM model is useful and accurate in the same sense as the flat-earth model (which is useful)!

8

The RAM Model of Computation

- The worst-case (time) complexity of an algorithm is the function defined by the maximum number of steps taken on any instance of size n.
- The best-case complexity of an algorithm is the function defined by the minimum number of steps taken on any instance of size n.
- The average-case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size n.

- Each of these complexities defines a numerical function: time vs. size!

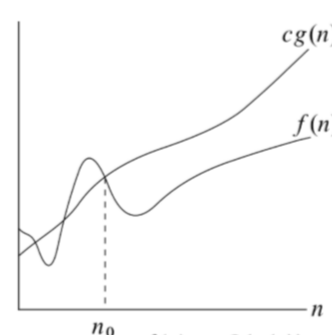
9

Asymptotic Notation – Big Oh, O

$$O(g(n)) = \{f(n) : \exists c > 0, \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)\}$$

- In plain English: $O(g(n))$ are all functions $f(n)$ for which there exists two positive constants c and n_0 such that for all $n \geq n_0$, $0 \leq f(n) \leq cg(n)$
  - $g(n)$ is an asymptotic upper bound for $f(n)$
- Intuitively, you can think of O as "≤" for functions
- If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$

- The definition implies a constant n_0 beyond which they are satisfied. We do not care about small value of n.



$f(n) = O(g(n))$

10

Examples

- Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

11

Examples

- Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

$2^{n+1} = O(2^n)$, but $2^{2n} \neq O(2^n)$.

To show that $2^{n+1} = O(2^n)$, we must find constants $c, n_0 > 0$ such that

$0 \le 2^{n+1} \le c \cdot 2^n$ for all $n \ge n_0$.

Since $2^{n+1} = 2 \cdot 2^n$ for all n, we can satisfy the definition with $c = 2$ and $n_0 = 1$.

To show that $2^{2n} \neq O(2^n)$, assume there exist constants $c, n_0 > 0$ such that

$0 \le 2^{2n} \le c \cdot 2^n$ for all $n \ge n_0$.

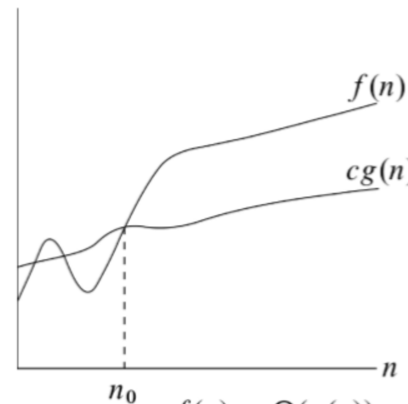
Then $2^{2n} = 2^n \cdot 2^n \le c \cdot 2^n \Rightarrow 2^n \le c$. But no constant is greater than all 2^n, and so the assumption leads to a contradiction.

12

Asymptotic Notation – Big Omega, $\mathbf{\Omega}$

$\Omega(g(n)) = \{f(n) : \exists c > 0, \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)\}$

- In plain English: $\Omega(g(n))$ are all function $f(n)$ for which there exists two positive constants c and n_0 such that for all $n \geq n_0$, $0 \leq cg(n) \leq f(n)$
  - $g(n)$ is an asymptotic lower bound for $f(n)$
- Intuitively, you can think of Ω as "$\geq$" for functions



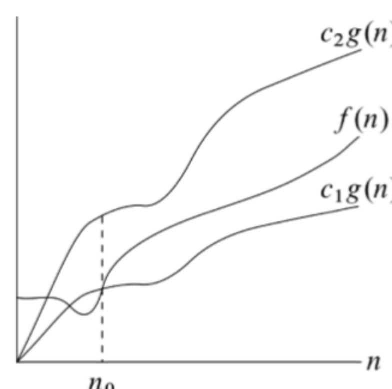
$f(n) = \Omega(g(n))$

13

Asymptotic Notation – Theta, $\mathbf{\Theta}$

$\Theta(g(n)) = \{f(n) : \exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

- In plain English: $\Theta(g(n))$ are all function $f(n)$ for which there exists three positive constants c_1, c_2, and n_0 such that for all $n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
- In other words, all functions that grow at the same rate as $g(n)$
  - $g(n)$ is an asymptotically tight bound for $f(n)$
- Intuitively, you can think of Θ as "$=$" for functions



$f(n) = \Theta(g(n))$

14

Asymptotic Notation – Theta, Θ

Theorem

$f(n) = \Theta(g(n))$ iff $f = O(g(n))$ and $f = \Omega(g(n))$

15

Asymptotic notation in equations and inequalities

- On the right-hand side alone of an equation (or inequality) ≡ a set of functions
  - ex., $n = O(n^2) \leftrightarrow n \in O(n^2)$
- In general, in a formula, stands for some anonymous function that we do not care to name
  - ex., $2n^2 + 3n + 1 = 2n^2 + \Theta(n) \leftrightarrow 2n^2 + 3n + 1 = 2n^2 + f(n)$, where $f(n) = \Theta(n)$
  - help eliminate inessential detail and clutter in a formula
  - ex., $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

16

Asymptotic Notation – little oh, o

$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0 : 0 \leq f(n) < cg(n)\}$

- In plain English: $o(g(n))$ are all function $f(n)$ for which for all constant $c > 0$, there exists a constant $n_0 > 0$ such that for all $n \geq n_0$, $0 \leq f(n) < cg(n)$
  - $o(g(n))$ is an asymptotic upper bound for $f(n)$, but not tight
  - $f(n)$ becomes insignificantly relative to $g(n)$ as n grows
  - $f(n)$ grows asymptotically slower than $g(n)$
- Similar to $O(g(n))$, intuitively, you can think of o as "<" for functions

17

Asymptotic Notation – little oh, o

Examples

- $n^{1.999} = o(n^2)$
- $\frac{n^2}{\log(n)} = o(n^2)$
- $n^2 \neq o(n^2)$
- $\frac{n^2}{1000} \neq o(n^2)$

18

Asymptotic Notation – little Omega, ω

$$\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0 : 0 \leq cg(n) < f(n)\}$$

- In plain English: $\omega(g(n))$ are all function $f(n)$ for which for all constant $c > 0$, there exists a constant $n_0 > 0$ such that for all $n \geq n_0$, $0 \leq cg(n) < f(n)$
  - $\omega(g(n))$ is an asymptotic lower bound for $f(n)$, but not tight
  - $f(n)$ becomes arbitrarily large relative to $g(n)$ as n grows
  - $f(n)$ grows asymptotically faster than $g(n)$
  - Similar to $\Omega(g(n))$

- Intuitively, you can think of ω as "$>$" for functions

19

Asymptotic Notation – little Omega, ω

Examples:

- $n^{2.0001} = \omega(n^2)$
- $n^2 \log(n) = \omega(n^2)$
- $n^2 \neq \omega(n^2)$

20

Asymptotic Notation

- Asymptotic notation is a way to compare functions
  - $O \approx \leq$
  - $\Omega \approx \geq$
  - $\Theta \approx =$
  - $o \approx <$
  - $\omega \approx >$

- When using asymptotic notations, sometimes,
  - drop lower-order terms
  - ignore constant coefficient in the leading term

21

Asymptotic Notation Multiplication by Constant

- Multiplication by a constant does not change the asymptotic:

$$O(c \cdot f(n)) \rightarrow O(f(n))$$
$$\Omega(c \cdot f(n)) \rightarrow \Omega(f(n))$$
$$\Theta(c \cdot f(n)) \rightarrow \Theta(f(n))$$

- The "old constant" C from the Big Oh becomes $c \cdot C$.

22

Asymptotic Notation Multiplication by Function

- But when both functions in a product are increasing, both are important:

$$O(f(n)) \cdot O(g(n)) \rightarrow O(f(n) \cdot g(n))$$
$$\Omega(f(n)) \cdot \Omega(g(n)) \rightarrow \Omega(f(n) \cdot g(n))$$
$$\Theta(f(n)) \cdot \Theta(g(n)) \rightarrow \Theta(f(n) \cdot g(n))$$

This is why the running time of two nested loops is $O(n^2)$.

Testing Dominance

- $f(n)$ dominates $g(n)$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$, which is the same as saying $g(n) = o(f(n))$.
- Note the little-oh – it means “grows strictly slower than”.

Dominance Rankings

- You must come to accept the dominance ranking of the basic functions:

- $n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$

25

Advanced Dominance Rankings

- Additional functions arise in more sophisticated analysis:

- $n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg (\log n)^2 \gg \log n \gg {}^{\log n}/_{\log \log n} \gg \log \log n \gg 1$

26

Logarithms

- It is important to understand deep in your bones what logarithms are and where they come from.
- A logarithm is simply an inverse exponential function.
- Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.
- Logarithms reflect how many times we can double something until we get to n or halve something until we get to 1.

27

The Base is not Asymptotically Important

- Recall the definition, $c^{\log_c x} = x$ and that
- $\log_b a = \frac{\log_c a}{\log_c b}$
- Thus, $\log_2 n = (1/\log_{100} 2) \times \log_{100} n$. Since $\frac{1}{\log_{100} 2} = 6.643$ is just a constant, it does not matter in the Big Oh.

28

Analyzing Merge Sort

Input: $S = \{a_1, a_2, \ldots, a_n\}$
Output: A sorted permutation of the input sequence

if $n \leq 1$ **then**
  return S;
else
  $m = \frac{n}{2}$; // divide input into two
  left $= S[1..m]$; *right* $= S[m+1..n]$;
  merge_sort(*left*); // conquer subproblem$_1$
  merge_sort(*right*); // conquer subproblem$_2$
  merge(*left*, *right*); // combine subproblems
end
return S

29

Analyzing Merge Sort – Merge Procedure

Input: S_1, S_2, two sorted sequences
Output: S, a sorted sequence containing S_1 and S_2

sequence S;
while *length*$(S_1) > 0$ *and length*$(S_2) > 0$ **do**
  // merge in order elements of S_1 and S_2
  if *first*$(S_1) \leq$ *first*(S_2) **then**
    append(*first*(S_1), S); *discard*(*first*(S_1));
  else
    append(*first*(S_2), S); *discard*(*first*(S_2));
  end
end
while *length*$(S_1) > 0$ **do**
  append(*first*(S_1), S); *discard*(*first*(S_1)); // add whatever is left in S_1
end
while *length*$(S_2) > 0$ **do**
  append(*first*(S_2), S); *discard*(*first*(S_2)); // add whatever is left in S_2
end
return S

30

Analyzing Merge Sort

Input: {10,5,7,6,1,4,8,3}
Output: {1,3,4,5,6,7,9,10}

Input: $S = \{a_1, a_2, \ldots, a_n\}$
Output: A sorted permutation of the input sequence

```
if n ≤ 1 then
    return S;
else
    m = n/2;                          // divide input into two
    left = S[1..m]; right = S[m+1..n];
    merge_sort(left);                 // conquer subproblem1
    merge_sort(right);                // conquer subproblem2
    merge(left, right);               // combine subproblems
end
return S
```


Analyzing Algorithms – recursive algorithms

- Divide-and-conquer paradigm
  - solve a problem for a given input of size n
  - You don't know how to solve it for any n, but
    - you can **divide** the problem into smaller subproblems (input sizes $n' < n$)
    - you can **conquer** (solve) the subproblems recursively – you need a base case: for a small enough input solving the problem is straightforward
    - you can **combine** the solutions to the subproblems to solve the original problem with input size n
- Recursive algorithms call themselves with a different input
  - we cannot just count the number of instructions

Analyzing Merge Sort – running time

- In plain English, the cost of merge sort, i.e., $T(n)$ is
  - a constant if $n <= 1$; $\Theta(1)$
  - the cost of dividing, solving the subproblems and combining the subproblems, if $n > 1$
    - dividing: $\Theta(n)$
    - conquering (solving) subproblems: $2 \times T(n/2)$
    - combining subproblems: $\Theta(n)$

```
Input: S = {a_1, a_2, ..., a_n}
Output: A sorted permutation of the input sequence
if n ≤ 1 then
    return S;
else
    m = n/2;                                  // divide input into two
    left = S[1..m]; right = S[m+1..n];
    merge_sort(left);                         // conquer subproblem_1
    merge_sort(right);                        // conquer subproblem_2
    merge(left, right);                       // combine subproblems
end
return S
```


Analyzing Merge Sort – running time

- Formally

$$T(n) = \begin{cases} \Theta(1), \text{if } n = 1 \\ 2\mathrm{T}(n/2) + \Theta(n), \text{if } n > 1 \end{cases}$$

How do we solve a recurrence?
(**Solving** a **recurrence** relation **means** obtaining a closed-form **solution**: a non-**recursive** function of n.)

Resolving Recurrences

- Substitution method
  - Guess a solution and check if it is correct
    - Pros: rigorous, formal solution
    - Cons: requires induction, how do you guess?
- Recursion trees
  - Build a tree representing all recursive calls, infer the solution
    - Pros: intuitive, visual, can be used to guess
    - Cons: not too formal
- Master method
  - Check the cookbook and apply the appropriate case
    - Pros: formal, useful for most recurrences
    - Cons: have to memorize, useful for most recurrence

35

Resolving Recurrences – Substitution Method

- Steps
  - Guess the solution
  - Use induction to show that the solution works
- What is induction?
  - A method to prove that a given statement is true for all-natural numbers k
  - Two steps
    - Base case: show that the statement is true for the smallest value of n (typically 0)
    - Induction step: assuming that the statement is true for any k, show that it must also hold for $k + 1$

36

Resolving Recurrences – Substitution Method, induction

- A more serious example
- $T(n) = \begin{cases} 1 & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + n & if\ n > 1 \end{cases}$
- guess $T(n) = n\log n + n$
- $T(n) = \Theta(n \log n)$

37

$$T(n) = \begin{cases} 1 & \text{if } n = 1\ , \\ 2T(n/2) + n & \text{if } n > 1\ . \end{cases}$$

1. *Guess:* $T(n) = n \lg n + n$. *[Here, we have a recurrence with an exact function, rather than asymptotic notation, and the solution is also exact rather than asymptotic. We'll have to check boundary conditions and the base case.]*
2. *Induction:*

   Basis: $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$

   Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$. We'll use this inductive hypothesis for $T(n/2)$.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(\frac{n}{2}\lg\frac{n}{2} + \frac{n}{2}\right) + n \qquad \text{(by inductive hypothesis)} \\ &= n\lg\frac{n}{2} + n + n \\ &= n(\lg n - \lg 2) + n + n \\ &= n\lg n - n + n + n \\ &= n\lg n + n\ . \qquad \blacksquare \end{aligned}$$

38

Resolving Recurrences – Substitution Method, induction

- When we want an asymptotic solution to a recurrence, we don't worry about the base cases in our proof.
  - Since we are ultimately interested in an asymptotic solution to a recurrence, it will always be possible to choose base cases that work.

- When we want an exact solution, then we have to deal with base case.

39

Resolving Recurrences – Substitution Method, induction

- When solving recurrences with asymptotic notation
  - assume $T(n) = \Theta(1)$ for small enough n, don't worry about base case
  - show upper and lower bounds separately (O, Ω)
- Example
  - $T(n) = 2T(n/2) + \Theta(n)$

40

Resolving Recurrences – Recursion Trees

- Draw a "recursion tree" for the recurrence
  - Nodes are the cost of a single subproblem
  - Nodes have as their children the subproblems they are decomposed into
- Example
  - $T(n) = 2T(n/2) + n$

41

Example -- Recursion Tree Method

- Example
  - $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + \Theta(n^2)$
- For upper bound, rewrite as $T(n) \leq T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + cn^2$
- For lower bound, as $T(n) = \Omega(n^2)$

42

Asymptotic Behavior of a Geometric Series

- Sum a geometric series
- $S = n^2 + \frac{3}{4}n^2 + \left(\frac{3}{4}\right)^2 n^2 + \cdots + \left(\frac{3}{4}\right)^k n^2$
- $\sum_{i=1}^{n} a_i = a\left(\frac{1-r^n}{1-r}\right)$, $\sum_{i=1}^{\infty} a_i = a\left(\frac{1}{1-r}\right)$ if $|r| < 1$
- A decreasing geometric series behaves asymptotically just like its 1st term: $S = \Theta(n^2)$
- By symmetry, if S were increasing, it would behave asymptotically like its final term: $S = n^2 + 2n^2 + 2^2n^2 + \cdots + 2^k n^2 = \Theta\left(2^k n^2\right)$

43

Resolving Recurrences – Master Method

- "Cookbook" for many divide-and-conquer recurrences of the form $T(n) = aT(n/b) + f(n)$, where
  - $a \geq 1$
  - $b \geq 1$
  - $f(n)$ is an asymptotically positive function
- What are a, b, and $f(n)$?
  - a is a constant corresponding to ...
  - b is a constant corresponding to ...
  - $f(n)$ is a function corresponding to ...

44

Resolving Recurrences – Master Method

- Problem: You want to solve a recurrence of the form $T(n) = aT(n/b) + f(n)$, where
  - $a \geq 1$, $b \geq 1$ and $f(n)$ is asymptotically positive function
- Solution:
  - Compare $n^{\log_b a}$ vs. $f(n)$, and apply the Master Theorem

45

Resolving Recurrences – Master Method

- Intuition: the larger of the two functions determines the solution

- The three cases compare $f(n)$ with $n^{\log_b(a)}$
  - Case 1: $n^{\log_b(a)}$ is larger – O definition
  - Case 2: $f(n)$ and $n^{\log_b(a)}$ grow at the same rate – Θ definition
  - Case 3: $f(n)$ is larger – Ω definition

- The three cases do not cover all possibilities
  - but they cover most cases we are interested in

46

The Master Theorem

- Case 1: If $f(n) = O\left(n^{(\log_b a)-\epsilon}\right)$ for some $\epsilon > 0$
then $T(n) = \Theta\left(n^{\log_b(a)}\right)$

- Case 2: If $f(n) = \Theta\left(n^{\log_b(a)}\right)$
then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$

- Case 3: If $f(n) = \Omega\left(n^{(\log_b a)+\epsilon}\right)$ for some $\epsilon > 0$
and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n (regularity condition),
then $T(n) = \Theta\left(f(n)\right)$

A function $f(n)$ is polynomially bounded if $f(n) = O\left(n^k\right)$ for some constant k.

You need to memorize these rules and be able to use them

47

The Master Theorem

Case 2:

If $f(n) = \Theta\left(n^{\log_b(a)}(\log n)^k\right)$ for some constant $k \geq 0$
then $T(n) = \Theta\left(n^{\log_b(a)}(\log n)^{k+1}\right)$

48

Resolving Recurrences – Master Method, examples

$T(n) = aT(n/b) + f(n)$,
$n^{\log_b a}$ vs. $f(n)$

- Examples:
- $T(n) = 4T\left(\frac{n}{2}\right) + \mathrm{n}$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

49

Resolving Recurrences – Master Method, examples

- Examples:
- $T(n) = 5T\left(\frac{n}{2}\right) + \Theta(n^3)$

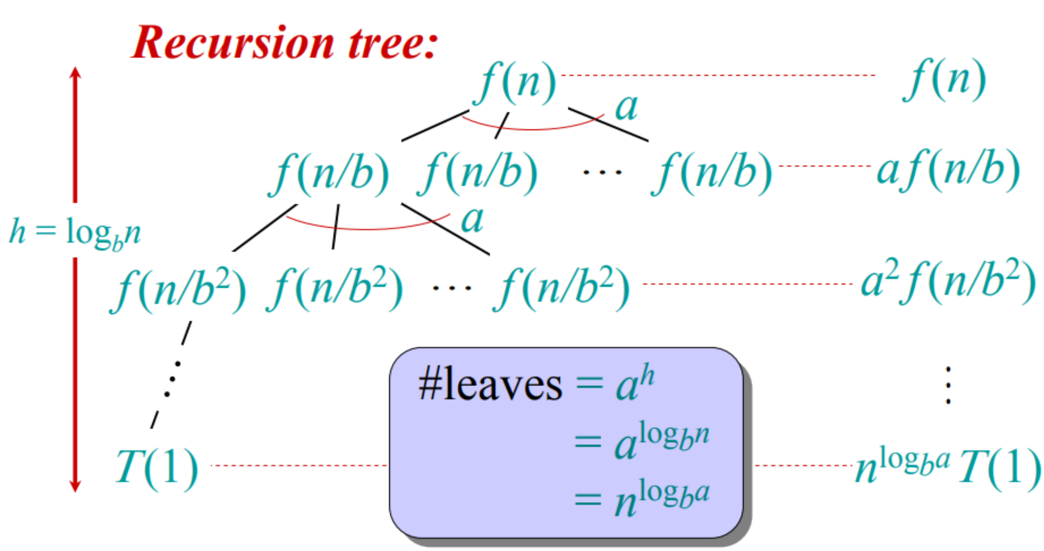
50

Resolving Recurrences – Master Method, examples

- Examples:
- $T(n) = 27T\left(\frac{n}{3}\right) + \Theta\left(n^3/\log n\right)$

51

Idea of master theorem



52

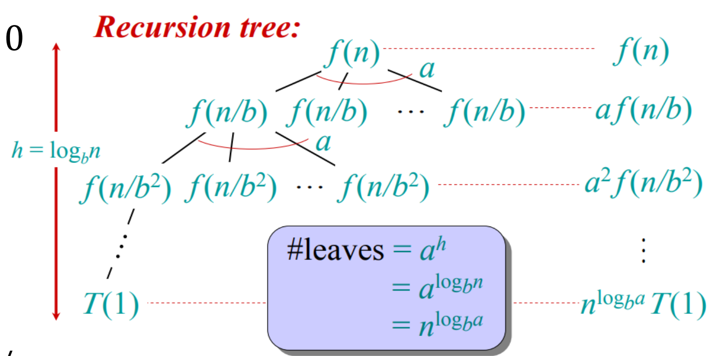
Resolving Recurrences – Master Method

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

then $T(n) = \Theta\left(n^{\log_b a}\right)$

$f(n)$ is polynomially smaller than $n^{\log_b a}$

Intuition: cost is dominated by the leaves



53

Resolving Recurrences – Master Method

- Case 3: If $f(n) = \Omega\left(n^{(\log_b a)+\epsilon}\right)$ for some $\epsilon > 0$

and if $af(n/b) \le cf(n)$ for some $c < 1$ and all sufficiently large n,

then $T(n) = \Theta\left(f(n)\right)$

The weight decreases geometrically from the root to the leaves.
The root holds a constant fraction of the total weight.

Intuition: cost is dominated by root, we can discard the rest

54

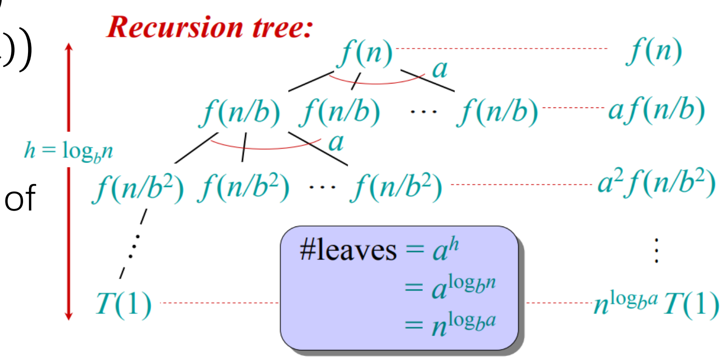
Resolving Recurrences – Master Method

- Case 2: If $f(n) = \Theta\left(n^{\log_b a}\right)$

then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$

$f(n)$ grows at the same rate of $n^{\log_b a}$

Intuition: cost $n^{\log_b a}$ at each level, there are $\log(n)$ levels



55

Resolving Recurrences – Master Method

Case 3: If $f(n) = \Omega\left(n^{(\log_b a)+\epsilon}\right)$ for some $\epsilon > 0$

and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n,

then $T(n) = \Theta\left(f(n)\right)$

What's with the Case 3 regularity condition?

Generally, not a problem. It always holds whenever $f(n) = n^k$ and $f(n) = \Omega\left(n^{(\log_b a)+\epsilon}\right)$ for constant $\epsilon > 0$. So, you don't need to check it when $f(n)$ is a polynomial.

56