Master Method

Motivation

Design and Analysis
of Algorithms I

Integer Multiplication Revisited

Motivation : potentially useful algorithmic ideas often need mathematical analysis to evaluate

Recall : grade-school multiplication algorithm uses $\theta(n^2)$ operation to multiply two n-digit numbers

A Recursive Algorithm

<u>Recursive approach</u>

Write $x = 10^{n/2}a + b \qquad y = 10^{n/2}c + d$

[where a,b,c,d are n/2 – digit numbers]

<u>So</u> :

$$x \cdot y = 10^n ac + 10^{n/2}(ad + bc) + bd \qquad (*)$$

<u>Algorithm#1</u> : recursively compute ac,ad,bc,bd, then compute (*) in the obvious way.

A Recursive Algorithm

T(n) = maximum number of operations this algorithm needs to multiply two n-digit numbers

<u>Recurrence</u> : express T(n) in terms of running time of recursive calls.

<u>Base Case</u> : T(1) <= a constant.

<u>For all n > 1</u> : $$T(n) \leq 4T(n/2) + O(n)$$

Work done here

Work done by recursive calls

A Better Recursive Algorithm

Algorithm #2 (Gauss) : recursively compute ac,$^{(1)}$ bd,$^{(2)}$ (a+b)(c+d)$^{(3)}$ [recall ad+bc = (3) – (1) – (2)]

New Recurrence :

Base Case : T(1) <= a constant

Ter
ver

Which recurrence best describes the running time of Gauss's algorithm for integer multiplication?

- ◯ $T(n) \leq 2T(n/2) + O(n^2)$
- ◯ $3T(n/2) + O(n)$
- ◯ $4T(n/2) + O(n)$
- ◯ $4T(n/2) + O(n^2)$

A Better Recursive Algorithm

Algorithm #2 (Gauss) : recursively compute ac,(1) bd,(2) (a+b)(c+d)(3) [recall ad+bc = (3) – (1) – (2)]

New Recurrence :

Base Case : T(1) <= a constant

Work done here

For all n>1 : $T(n) \leq 3T(n/2) + O(n)$

Work done by recursive calls

Design and Analysis of Algorithms I

Master Method

The Precise Statement

The Master Method

<u>Cool Feature</u> : a "black box" for solving recurrences.

<u>Assumption</u> : all subproblems have equal size.

Recurrence Format

1. <u>Base Case</u> : T(n) <= a constant for all sufficiently small n
2. For all larger n :

$$T(n) \leq aT(n/b) + O(n^d)$$

where

a = number of recursive calls (>= 1)

b = input size shrinkage factor (> 1)

d = exponent in running time of "combine step" (>=0)

[a,b,d <u>independent of n</u>]

The Master Method

Base doesn't matter (only changes leading constants)

•

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Base matters

Design and Analysis
of Algorithms I

Master Method

Examples

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Example #1

<u>Merge Sort</u>

$$\left.\begin{array}{l} a = 2 \\ b = 2 \\ d = 1 \end{array}\right\} \quad b^d = \boxed{a} => \ Case\ 1$$

$$T(n) = O(n^d \log n) = O(n \log n)$$

Where are the respective values of a, b, d for a binary search of a sorted array, and which case of the Master Method does this correspond to?

- 1, 2, 0 [Case 1] $\quad a = b^d \Rightarrow T(n) = O(n^d \log n) = O(\quad \log n)$
- 1, 2, 1 [Case 2]
- 2, 2, 0 [Case 3]
- 2, 2, 1 [Case 1]

Example #3

Integer Multiplication Algorithm # 1

a = 4
b = 2
d = 1

$$b^d = 2 < a \ \ (Case \ \ 3)$$

$$=> T(n) = O(n^{\log_b a}) = O(n^{\log_2 4})$$

$$= O(n^2)$$

Same as grade-school algorithm

Ter
ver

Where are the respective values of a, b, d for Gauss's recursive integer multiplication algorithm, and which case of the Master Method does this correspond to?

- ◯ 2, 2, 1 [Case 1]
- ◯ 3, 2, 1 [Case 1]
- ◯ 3, 2, 1 [Case 2]
- ◯ 3, 2, 1 [Case 3]

Better than the grade-school algorithm!!!

$$a = 3,\ b^d = 2\ a > b^d \quad (Case\ 3)$$

$$=> T(n) = O(n^{\log_2 3}) = O(n^{1.59})$$

Example #5

<u>Strassen's Matrix Multiplication Algorithm</u>

a = 7

b = 2

d = 2

$$\left.\begin{matrix} b = 2 \\ d = 2 \end{matrix}\right\} b^d = 4 < a \quad (Case\ 3)$$

$$=> T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

=> beats the naïve iterative algorithm !

Tim Roughgarden

Example #6

Fictitious Recurrence

$$T(n) \leq 2T(n/2) + O(n^2)$$

$\Rightarrow$ a = 2

$\left.\begin{array}{l}\Rightarrow \text{b = 2}\\ \Rightarrow \text{d = 2}\end{array}\right\}$ $b^d = 4 > a \quad (Case\ 2)$

$$=> T(n) = O(n^2)$$

Master Method

Proof (Part I)

Design and Analysis
of Algorithms I

The Master Method

- If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Preamble

Assume : recurrence is

I. $T(1) \leq c$

II. $T(n) \leq aT(n/b) + cn^d$

(For some constant c)

And n is a power of b.

(general case is similar, but more tedious)

Idea : generalize MergeSort analysis.

(i.e., use a recursion tree)

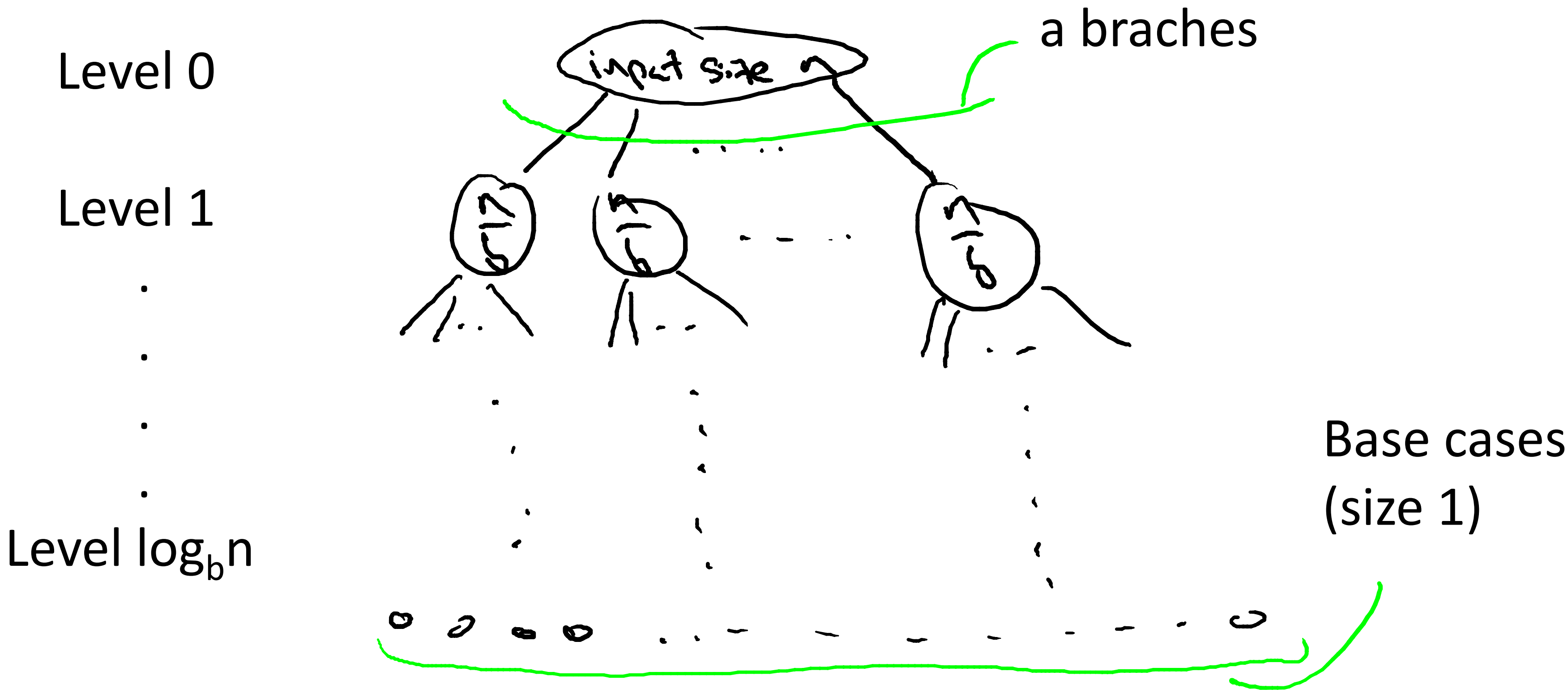
Ter
ver

What is the pattern ? Fill in the blanks in the following statement: at each level $j = 0,1,2,\ldots,\log_b n$, there are <blank> subproblems, each of size <blank>

# of times you can divide n by b before reaching 1

- ○ a^j and n/a^j, respectively.
- ○ a^j and n/b^j, respectively.
- ○ b^j and n/a^j, respectively.
- ○ b^j and n/b^j, respectively.

The Recursion Tree



Work at a Single Level

Total work at level j [ignoring work in recursive calls]

$$\le \underbrace{a^j}_{\text{\# of level-j subproblems}} \cdot c \cdot \left(\frac{n}{b^j}\right)^d = cn^d \cdot \left(\frac{a}{b^d}\right)^j$$

Work per level-j subproblem

Size of each level-j subproblem

Total Work

<u>Summing over all levels j = 0,1,2,..., $\log_b n$:</u>

$$\text{Total work} \leq cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \qquad (*)$$

Design and Analysis of Algorithms I

Master Method

Intuition for the 3 Cases

How To Think About (*)

- Our upper bound on the work at level j:

$$cn^d \times (\frac{a}{b^d})^j$$

<u>Interpretation</u>

a = rate of subproblem proliferation (RSP)

b^d = rate of work shrinkage (RWS)

(per subproblem)

Which of the following statements are true?
(Check all that apply.)

- ☐ If RSP < RWS, then the amount of work is decreasing with the recursion level j.
- ☐ If RSP > RWS, then the amount of work is increasing with the recursion level j.
- ☐ No conclusions can be drawn about how the amount of work varies with the recursion level j unless RSP and RWS are equal.
- ☐ If RSP and RWS are equal, then the amount of work is the same at every recursion level j.

Te
blo

Or
bu
13
24

Intuition for the 3 Cases

- Upper bound for level j: $cn^d \times (\frac{a}{b^d})^j$

1. RSP = RWS => Same amount of work each level (like Merge Sort) [expect $O(n^d \log(n))$]
2. RSP < RWS => less work each level => most work at the root [might expect $O(n^d)$]
3. RSP > RWS => more work each level => most work at the leaves [might expect O(# leaves)]

Master Method

Proof (Part II)

Design and Analysis
of Algorithms I

The Story So Far/Case 1

= 1 for all j

- Total work: $\leq cn^d \times \underbrace{\sum_{j=0}^{\log_b n} \boxed{\left(\frac{a}{b^d}\right)^j}}_{= (\log_b n + 1)}$ (*)

=1

$If \quad a = b^d, \; then$

$$(*) = cn^d(\log_b n + 1)$$

$$= O(n^d \log n)$$

[end Case 1]

Basic Sums Fact

For $r \neq 1$, we have

$$1 + r + r^2 + r^3 + ... + r^k = \frac{r^{k+1} - 1}{r - 1}$$

Proof : by induction (you check)

Upshot:

Independent of k

1. If r<1 is constant, RHS is <= $\dfrac{1}{1-r}$ = a constant
   I.e., 1st term of sum dominates
2. If r>1 is constant, RHS is <= $r^k \cdot (1 + \dfrac{1}{r-1})$
   I.e., last term of sum dominates

Case 2

- Total work: $\leq cn^d \times \sum_{j=0}^{\log_b n} (\frac{a}{b^d})^j$ (*)

:= r

<= a constant (independent of n)

[by basic sums fact]

$If \quad a < b^d \quad [RSP < RWS]$

$= O(n^d)$

[total work dominated by top level]

Case 3

:= r > 1

- Total work: $\leq cn^d \times \sum_{j=0}^{\log_b n} (\frac{a}{b^d})^j$ (*)

<= constant * largest term

$$If \quad a > b^d \quad [RSP > RWS]$$

$$(*) = O(n^d \cdot (\frac{a}{b^d})^{\log_b n})$$

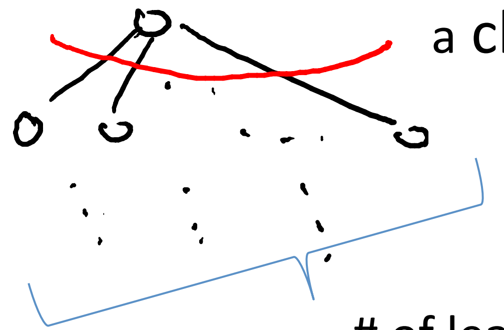
$$Note \quad : \quad b^{-d \log_b n} = (b^{log_b n})^{-d} = n^{-d}$$

$$So \quad : \quad (*) = O(a^{\log_b n})$$

Level 0

Level 1

Level $\log_b n$



a children

# of leaves = $a^{\log_b n}$

Ter

ver

Which of the following quantities is equal to $a^{\log_b n}$?

- The number of levels of the recursion tree.
- The number of nodes of the recursion tree.
- The number of edges of the recursion tree.
- The number of leaves of the recursion tree.

Case 3 continued

- Total work: $\leq cn^d \times \sum_{j=0}^{\log_b n} (\frac{a}{b^d})^j$ (*)

$$So: \quad (*) = O(a^{\log_b n}) = O(\#\ leaves)$$

$$Note: \quad a^{\log_b n} = n^{\log_b a}$$

More intuitive

Simpler to apply

$$[Since \quad (\log_b n)(\log_b a) = (\log_b a)(\log_b n)]$$

[End Case 3]

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$