



Design and Analysis
of Algorithms I

Master Method

Motivation

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

Recursive approach

Write $x = 10^{n/2}a + b$ $y = 10^{n/2}c + d$
[where a,b,c,d are n/2 – digit numbers]

So :

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

Algorithm#1 : 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

Recurrence : express $T(n)$ in terms of running time of recursive calls.

Base Case : $T(1) \leq$ a constant.

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

Work done
here

Work done by recursive calls

A Better Recursive Algorithm

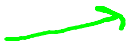
Algorithm #2 (Gauss): recursively compute $ac^{(1)}$, $bd^{(2)}$,
 $(a+b)(c+d)^{(3)}$ [recall $ad+bc = (3) - (1) - (2)$]

New Recurrence :

Base Case : $T(1) \leq \text{a constant}$

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 : $T(1) \leq$ a constant

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

Work done
here



Work done by recursive calls





Design and Analysis
of Algorithms I

Master Method

The Precise Statement

The Master Method

Cool Feature : a “black box” for solving recurrences.

Assumption : all subproblems have equal size.

Recurrence Format

1. Base Case : $T(n) \leq$ 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 independent of n]

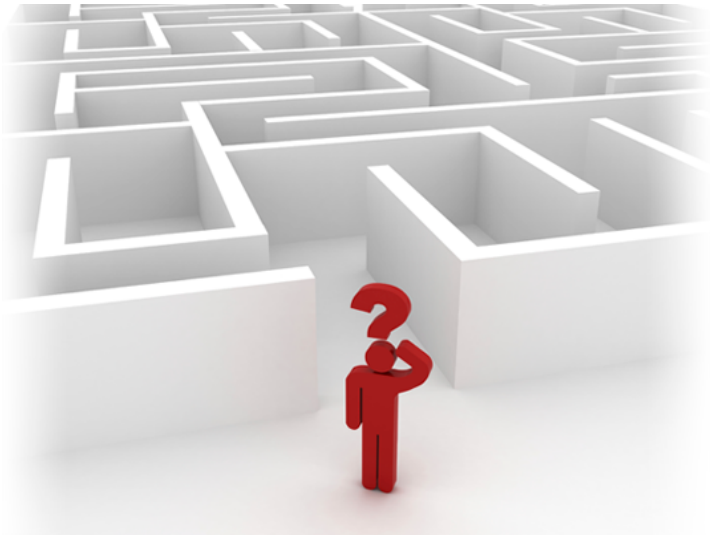
The Master Method

-

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

Base doesn't matter (only changes leading constants)

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

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

Example #1

Merge Sort

$$\left. \begin{array}{l} a = 2 \\ b = 2 \\ d = 1 \end{array} \right\} b^d = a \Rightarrow \textit{Case 1}$$

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

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]

☐ 1, 2, 1 [Case 2]

☐ 2, 2, 0 [Case 3]

☐ 2, 2, 1 [Case 1]

$$a = b^d \Rightarrow T(n) = O(n^d \log n) = O(\log n)$$

Example #3

Integer Multiplication Algorithm # 1

$$\left. \begin{array}{l} a = 4 \\ b = 2 \\ d = 1 \end{array} \right\} b^d = 2 < a \text{ (Case 3)}$$

$$\Rightarrow T(n) = O(n^{\log_b a}) = O(n^{\log_2 4})$$
$$= O(n^2)$$


Same as grade-school
algorithm

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!!!

$$a = 3, \quad b^d = 2 \quad a > b^d \quad (\text{Case 3})$$
$$\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.59})$$

Example #5

Strassen's Matrix Multiplication Algorithm

$$a = 7$$

$$b = 2$$

$$d = 2$$

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

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

\Rightarrow beats the naïve iterative algorithm !

Example #6

Fictitious Recurrence

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

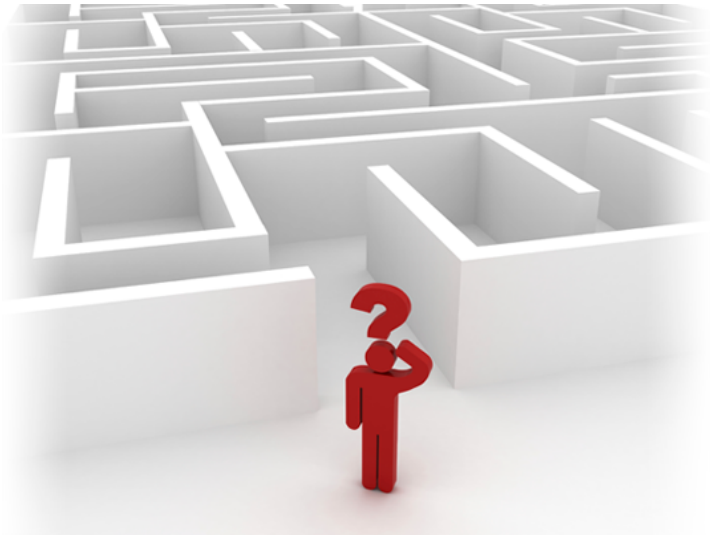
$$\Rightarrow a = 2$$

$$\Rightarrow b = 2$$

$$\Rightarrow d = 2$$

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

$$\Rightarrow T(n) = O(n^2)$$



Design and Analysis
of Algorithms I

Master Method Proof (Part I)

The Master Method

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

then

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

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)

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

of times you can divide n by b before reaching 1

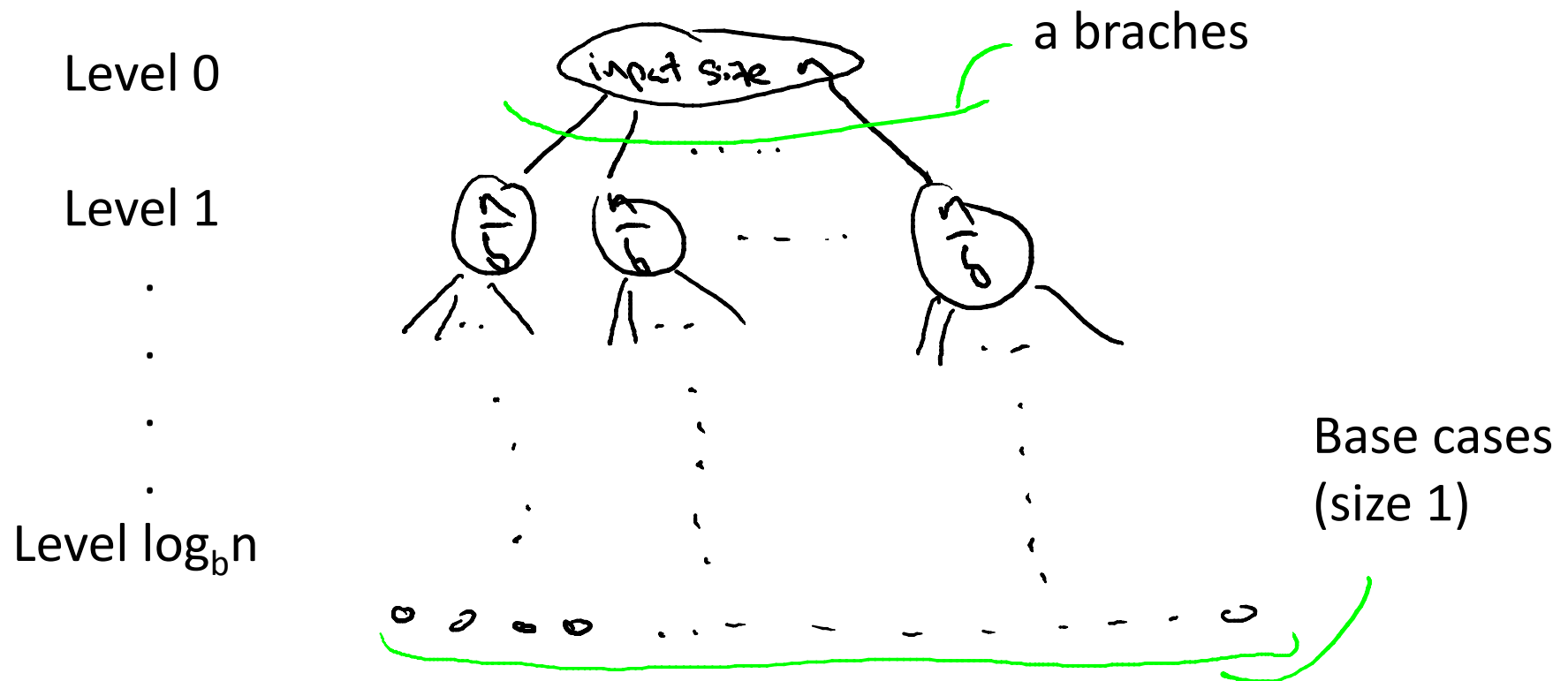
☐ a^j and n/a^j , respectively.

 ☒ 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]

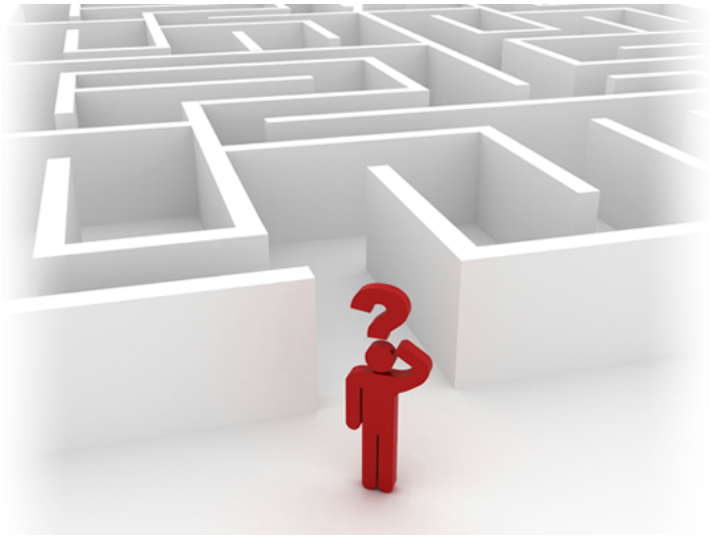
$$\leq \underbrace{a^j}_{\text{\# of level-}j \text{ subproblems}} \cdot c \cdot \underbrace{\left(\frac{n}{b^j}\right)^d}_{\text{Size of each level-}j \text{ subproblem}} = cn^d \cdot \left(\frac{a}{b^d}\right)^j$$

Work per level- j subproblem

Total Work

Summing over all levels $j = 0, 1, 2, \dots, \log_{\underline{b}} n$:

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



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:




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

Interpretation

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 .

Intuition for the 3 Cases

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

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



Design and Analysis
of Algorithms I

Master Method Proof (Part II)

The Story So Far/Case 1

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

If $a = b^d$, then

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

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

[end Case 1]

= 1 for
all j

= 1

= (log_b n + 1)

Basic Sums Fact

For $r \neq 1$, we have

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

Proof : by induction (you check)

Upshot:

1. If $r < 1$ is constant, RHS is $\leq \frac{1}{1 - r} = \text{a constant}$
i.e., 1st term of sum dominates

2. If $r > 1$ is constant, RHS is $\leq r^k \cdot \left(1 + \frac{1}{r - 1}\right)$
i.e., last term of sum dominates

Independent of k

Case 2

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

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

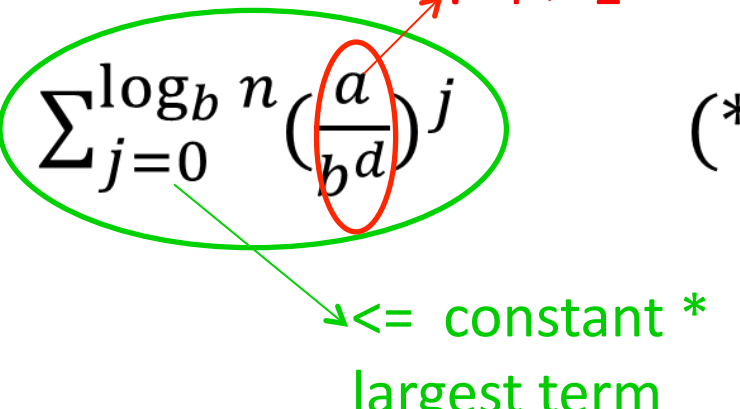
$$= O(n^d)$$

\leq a constant
(independent of n)
[by basic sums fact]

[total work dominated by top level]

Case 3

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



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

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

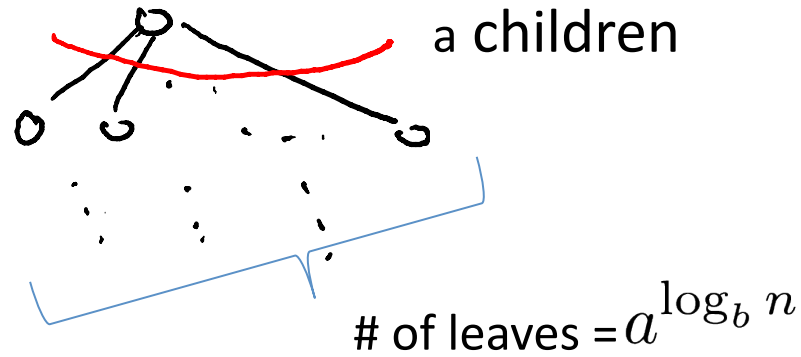
Note : $b^{-d \log_b n} = (b^{\log_b n})^{-d} = n^{-d}$

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

Level 0

Level 1

Level $\log_b n$



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

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

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

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

More intuitive
Simpler to apply

$$[\text{Since } (\log_b n)(\log_b a) = (\log_b a)(\log_b n)]$$

[End Case 3]

The Master Method

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

then

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