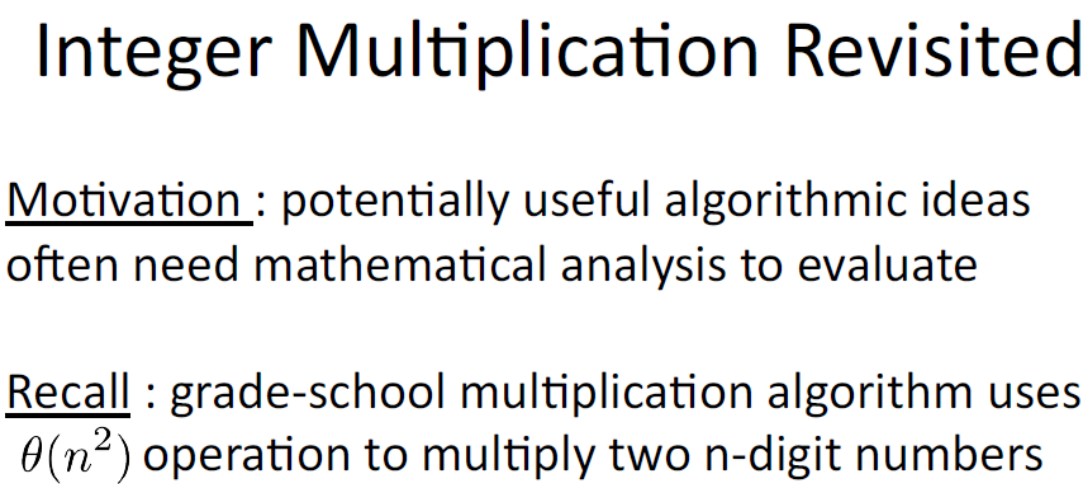
Part4-The Master Method

Completed on	@2020/03/17
Key videos	https://www.coursera.org/learn/algorithms-divide-conquer/lecture/HkcdO/formal-statement https://www.coursera.org/learn/algorithms-divide-conquer/lecture/Fkw1E/examples https://www.coursera.org/learn/algorithms-divide-conquer/lecture/nx5Nf/proof-i https://www.coursera.org/learn/algorithms-divide-conquer/lecture/39jDd/proof-ii
Note	Establishment, case study, and proofs of the Master Theorem.

Note

▼ Motivation

- 重新分析整数乘法



- 一般算法分析：分为四个子问题，加减都是O(n)复杂度！

A Recursive Algorithm

Recursive approach

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

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

So :

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

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

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) <= a constant.

Work done here

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

Work done by recursive calls

- 优化大数乘法：三次乘法，加上一下额外的加减O(n)复杂度

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

▼ Formal Statement

- 基本分析原理：假设所有子问题尺寸等价(all subproblems have equal size)
- 基本形式：

  除了跟输入大小n以外，有三个关键参数：递归子问题个数a；子问题缩减因子b；以及合并阶段指数d

Recurrence Format

1. Base Case : 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 independent of n]

Tim Roughgarden

- The Master Method

  三种情况的概括性分析

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

▼ Examples

- 基本公式

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

then

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

- ex1: MergeSort（归并排序）

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

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

- ex2: Binary Search（二值搜索）：一次调用a=1，大小减半b=2，一次比较d=0，常数复杂度

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]

$$\boxed{a = b^d => T(n) = O(n^d \log n) = O(\quad \log n)}$$

- ex3: Integer Multiplication：四次调用a=4，每次大小减半b=2，线性复杂度O(n)

<u>Integer Multiplication Algorithm # 1</u>

a = 4
b = 2
d = 1

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

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

$$= O(n^2)$$

- ex4: Gauss's recursive integer multiplication

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!!!

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

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

- Strassen's Matrix：7次调用，每次大小减半

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

a = 7

b = 2

d = 2

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

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

=> beats the naïve iterative algorithm !

- Fictitious Recurrence

<u>Fictitious Recurrence</u>

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

⇒ a = 2

⇒ b = 2

⇒ d = 2

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

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

▼ Proof I

- 前言：更为一般化/泛用的分析

Preamble

<u>Assume</u> : recurrence is

I. $T(1) \le c$

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

(For some constant c)

<u>And</u> n is a power of b.

(general case is similar, but more tedious)

<u>Idea :</u> generalize MergeSort analysis.

(i.e., use a recursion tree)

- 单层与总图分析

Work per level-j subproblem

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

Size of each level-j subproblem

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

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

▼ Interpretation of the 3 Cases

- 解读：a子问题增长率，b^d子问题工作量总体缩减率

Our upper bound on the work at level j:

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

Interpretation

a = rate of subproblem proliferation (RSP)

b^d = rate of work shrinkage (RWS)

(per subproblem)

分为三种情况进行分析，如果子问题越来越多，则复杂度大幅增加，反之，则每一层问题的总复杂度会不断缩减！

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.

- 总结：

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)]

▼ Proof II

分为三种情况进行分析：

- 子问题与复杂度相等

The Story So Far/Case 1

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

= 1 for all j

=1

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

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

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

[end Case 1]

- 另外两种不相等的情况

Case 2

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

:= 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

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

:= r > 1

<= constant * largest term

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

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

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

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

References

- Master Theorem

[主定理](https://zh.wikipedia.org/wiki/%E4%B8%BB%E5%AE%9A%E7%90%86)
https://zh.wikipedia.org/wiki/%E4%B8%BB%E5%AE%9A%E7%90%86

[Master theorem (analysis of algorithms)](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms))
https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

常用算法中的应用 [编辑]

算法	递回关系式	运算时间	备注
二分搜寻算法	$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$	$\Theta(\log n)$	情形二（$k = 0$）
二叉树遍历	$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$	$\Theta(n)$	情形一
最佳排序矩阵搜索(已排好序的二维矩阵)	$T(n) = 2T\left(\frac{n}{2}\right) + O(\log n)$	$\Theta(n)$	
合并排序	$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$	$\Theta(n \log n)$	情形二（$k = 0$）

- binary search

[二分搜尋演算法](https://zh.wikipedia.org/wiki/%E4%BA%8C%E5%88%86%E6%90%9C%E5%B0%8B%E6%BC%94%E7%AE%97%E6%B3%95)
https://zh.wikipedia.org/wiki/%E4%BA%8C%E5%88%86%E6%90%9C%E5%B0%8B%E6%BC%94%E7%AE%97%E6%B3%95

[Binary search algorithm](https://en.wikipedia.org/wiki/Binary_search_algorithm)
https://en.wikipedia.org/wiki/Binary_search_algorithm