Week 2. Lecture Notes

Topics: The Master Method
Divide and Conquer
Strassen's Algorithm
Quick Sort

The Master Method

The Master Method applies to recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ and f is asymptotically positive

Three Common Cases

Compare $f(n)$ with $n^{\log_b a}$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$,

$f(n)$ grows polynomially slower than ~~than~~ $n^{\log_b a}$ by a n^{ε} factor

Solution: $T(n) = \Theta(n^{\log_b a})$

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$

$f(n)$ and $n^{\log_b a}$ grows at similar rates

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$

$f(n)$ grows polynomially faster than $n^{\log_b a}$, by a factor n^ε factor)

and $f(n)$ satisfies the 'regularity condition' that of $af(n/b) \leq cf(n)$ for some constant $c < 1$

Solution: $T(n) = \Theta(f(n))$

Examples

1. $T(n) = 4T(n/2) + n$

here $a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n$

Case 1: $f(n) = O(n^{2-\varepsilon})$, for $\varepsilon = 1$

$\therefore T(n) = \Theta(n^2)$

2. $T(n) = 4T(n/2) + n^2$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$

Case 2: $f(n) = \Theta(n^2 \log^0 n)$, i.e. $k = 0$

$\therefore T(n) = \Theta(n^2 \log n)$

3. $T(n) = 4T(n/2) + n^3$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3$

Case 3: $f(n) = \Omega\left(n^{2+\varepsilon}\right)$ for $\varepsilon = 1$

and $4\left(cn/2\right)^2 \le cn^3$ (reg cond) for $c = 1/2$

$\therefore\ T(n) = \Theta(n^3)$

4. $T(n) = 4T(n/2) + \dfrac{n^2}{\lg n}$

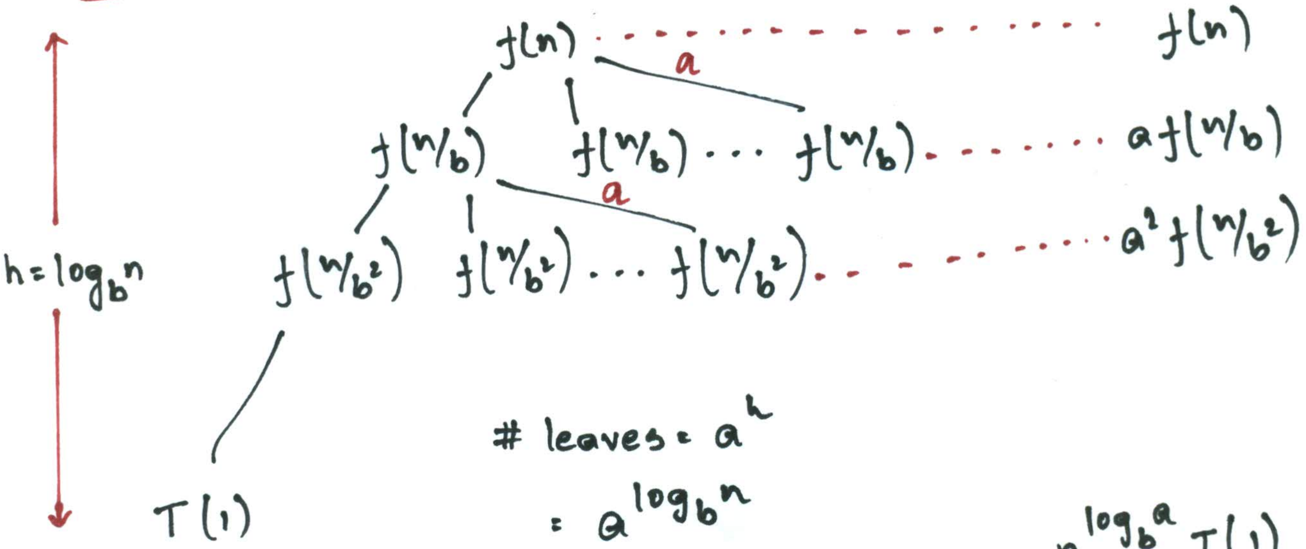
$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = \dfrac{n^2}{\lg n}$

Master method doesnot apply.

In particular for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\lg n)$

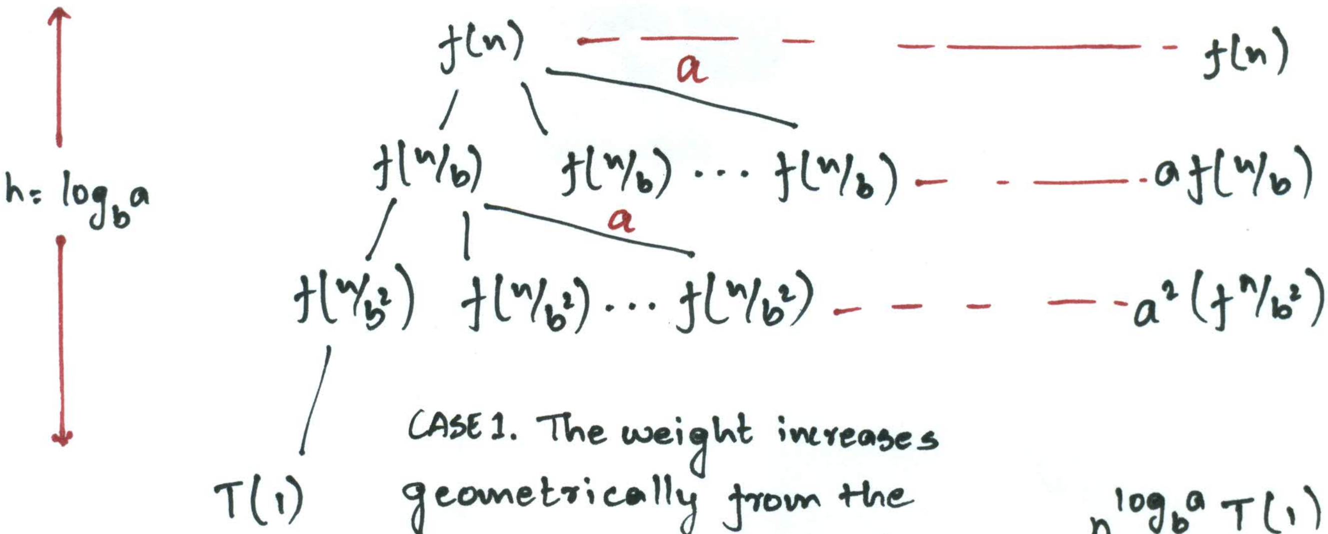
The Idea of Master Theorem

Recursion Tree:



$$\# \text{ leaves} = a^h = a^{\log_b n} = n^{\log_b a}$$

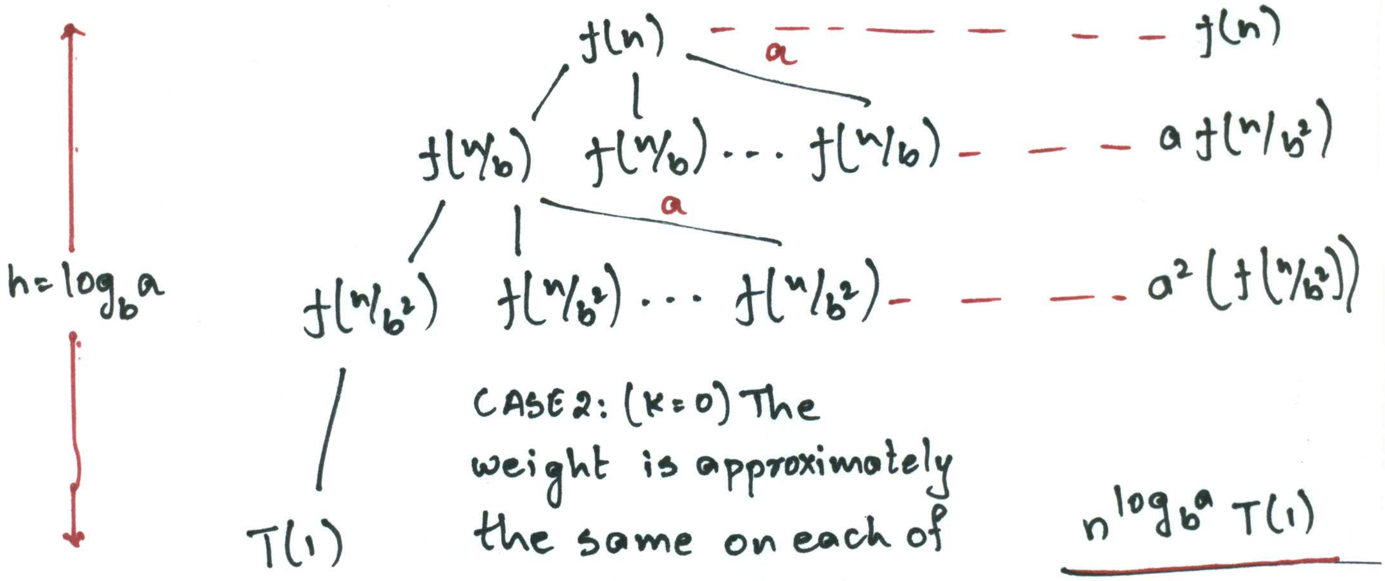
Recursion Tree



CASE 1. The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight

$$\frac{n^{\log_b a}\, T(1)}{\Theta\left(n^{\log_b a}\right)}$$

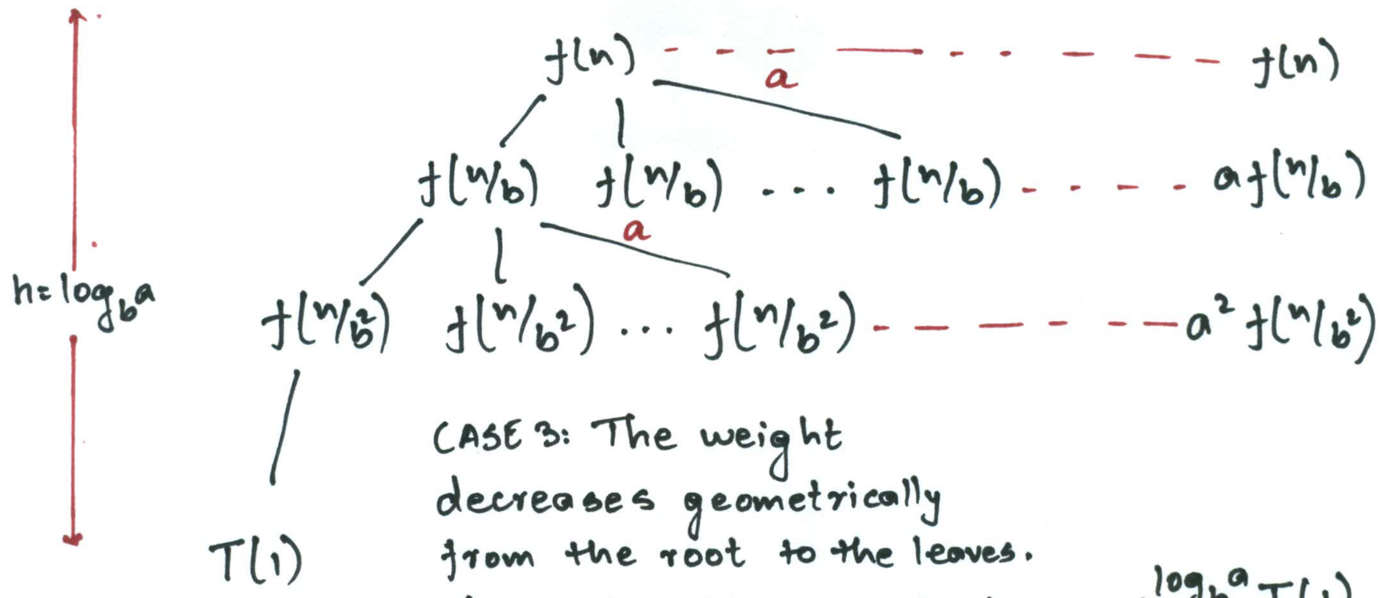
Recursion Tree



CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels

$$\frac{n^{\log_b a}\, T(1)}{\Theta\left(n^{\log_b a} \lg n\right)}$$

Recursion Tree



CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

$n^{\log_b a} T(1)$

$\Theta(f(n))$

The divide-and-conquer design paradigm

1. Divide the problem (instance) into subproblems
2. Conquer the subproblems by solving them recursively
3. Combine subproblem solutions

Example: Merge Sort

1. Divide: Trivial
2. Conquer: Recursively sort 2 subarrays
3. Combine: Linear-time merge

$$T(n) = 2\,T(n/2) + O(n)$$

\# subproblems

subproblem size

work dividing and combining

Binary search

Find an element in a sorted array

1. Divide - Check middleelement
2. Conquer - Recursively search 1 subarray
3. Combine - Trivial

Example:

Find 9 in

3 5 7 8 9 12 15

3 5 7 8 9 12 15

$3 \quad 5 \quad 7 \quad 8 \quad \boxed{9 \quad 12 \quad 15}$

3 5 7 8 9 12 15

3 5 7 8 9 12 15

$3 \quad 5 \quad 7 \quad 8 \quad \boxed{9} \quad 12 \quad 15$

Recurrence for binary search

$$T(n) = 1\,T(n/2) + \Theta(1)$$

$n^{\log_b a} = n^{\log_2 1} = n^{\log 2} = n^0 = 1 \Rightarrow$ CASE 2 $(k=0)$

$\Rightarrow T(n) = \Theta(\log n)$

Powering a Number

Problem: Compute a^n, $n \in \mathbb{N}$

Naive algorithm: $\Theta(n)$

Divide and conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1)$$

$$\Rightarrow T(n) = \Theta(\lg n)$$

Fibonacci Numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

0 1 1 2 3 5 8 13 21 ...

Naive recursive algorithm: $\Omega(\phi^n)$ (exponential time), where $\phi = (1+\sqrt{5})/2$ is the 'golden ratio.'

Computing Fibonacci Numbers

Naive recursive squaring:

$F_n = \phi^n/\sqrt{5}$, rounded to nearest integer

- Recursive squaring: $\Theta(\lg n)$ time
- This method is unreliable, since floating-point arithmetic is prone to round-off errors.

Bottom-up:

- Compute $F_0, F_1, F_2, \ldots, F_n$ in order, forming each number by summing the two previous.
- Running time: $\Theta(n)$

Recursive Squaring

Theorem: $$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Algorithm: Recursive squaring

Time $= \Theta(\lg n)$

Proof of theorem: (Induction on n)

For $n=1$: $\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$, which is true

Inductive step $(n \geq 2)$

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n}$$

$\blacksquare$

Matrix Multiplication

Input: $A = [a_{ij}]$, $B = [b_{ij}]$

Output: $C = [c_{ij}] = A \cdot B$, $i, j = 1, 2, \ldots, n$

$$c_{i,j} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \quad , \quad i, j = 1, 2, \ldots, n$$

Standard Algorithm

1. for $i \leftarrow 1$ to n
2. do for $j \leftarrow 1$ to n
3. do $c_{ij} \leftarrow 0$
4. for $k \leftarrow 1$ to n
5. do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{ik}$

Running time = $\Theta(n^3)$

Divide-and-Conquer Algorithm

IDEA:

$n \times n$ matrix = 2×2 matrices of $(n/2) \times (n/2)$ submatrices

$$\left[\begin{array}{c:c} r & s \\ \hdashline t & u \end{array}\right] = \left[\begin{array}{c:c} a & b \\ \hdashline c & d \end{array}\right] \left[\begin{array}{c:c} e & f \\ \hdashline g & h \end{array}\right]$$

$$C \quad = \quad A \quad \cdot \quad B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dh \\ u &= cf + dh \end{aligned}\right\}$$

8 mults of $(n/2) \times (n/2)$ submatrices

4 adds of $(n/2) \times (n/2)$ submatrices

Analysis of Divide and Conquer algorithm

$$T(n) = 8\,T(n/2) + \Theta(n^2)$$

#submatrices

submatrix size

work adding submatrices

$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \cdot \; T(n) = \Theta(n^3)$$

No better than the ordinary algorithm

Strassen's Idea

Multiply 2x2 matrices with only 7 recursive multiplications

$P_1 = a.(f-h)$

$P_2 = (a+b).h$

$P_3 = (c+d).e$

$P_4 = d.(g-e)$

$P_5 = (a+d).(e+h)$

$P_6 = (b-d).(g+h)$

$P_7 = (a-c).(e+f)$

$r = P_5 + P_4 - P_2 + P_6$

$s = P_1 + P_2$

$t = P_3 + P_4$

$u = P_5 + P_1 - P_3 - P_7$

7 mults, 18 adds/subs

Note: No reliance on commutativity of mults.

Here,

$$r = P_5 + P_4 - P_2 + P_6$$

$$= (a+d)(e+h) + d(g-e) - (a+b)h + (b-d)(g+h)$$

$$= ae + ah + de + dh + dg - de - ah - bh + bg + bh - dg - dh$$

$$= ae + bg$$

Similarly, others can be proved

Strassen's Algorithm

1. Divide: Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using + and -
2. Conquer: Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively
3. Combine: Form C using + and - on $(n/2) \times (n/2)$ submatrices.

$$T(n) = 7T(n/2) + \Theta(n^2)$$

Analysis of Strassen's Algorithm

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow \text{CASE 1: } T(n) = \Theta(n^{\log 7})$$

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 30$ or so.

Best upto date: $\Theta(n^{2.376\ldots})$

Quick Sort

- Proposed by C.A.R. Hoare in 1962
- Divide and Conquer Algorithm
- Sorts "in place" (like insertion sort)
- Very practical

Divide and Conquer

Quicksort an n-element array:

1. Divide: Partition the array into two subarrays around a pivot x such that elements in lower subarray $\leq x \leq$ elements in upper subarray

$x \leq$	x	$\geq x$

2. Conquer: Recursively sort the two subarrays

3. Combine: Trivial

Key: Linear-time partitioning subroutine

Partitioning Subroutine

```
PARTITION (A,p,q) ▷ A[p...q]
    x ← A[p] ▷ pivot = A[p]
    i ← p
    for j ← p+1 to q
        do if A[j] ≤ x
            then i ← i+1
                exchange A[i] ↔ A[j]
    exchange A[p] ↔ A[i]
    return i
```

Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11
			↑ i				