Analysis of Algorithms

COMP.4040, Spring 2019

Chapter 4: Divide and Conquer

By: Sirong Lin, PhD

University of
Massachusetts
Lowell

UMASS LOWELL

Learning with Purpose

Outline

Examples based on divide-and-conquer

Methods for solving recurrence

- substitution method
- recursion tree
- master method

Divide and Conquer Review

Divide and conquer approach:

- Divide: the problem into small subproblems that are smaller instances of the same problem
- Conquer: the subproblems by solving them *recursively*
  - base case: if the subproblems are small enough, just solve them
- Combine: the sub-solutions to solve the original problem

Recurrences Review

recurrence: an equation or inequality that describes a function in terms of its value on smaller inputs

solving the *recurrence* gives us *asymptotic* running time

so we often ignore boundary conditions, for sufficiently small n, e.g., $n = 1$

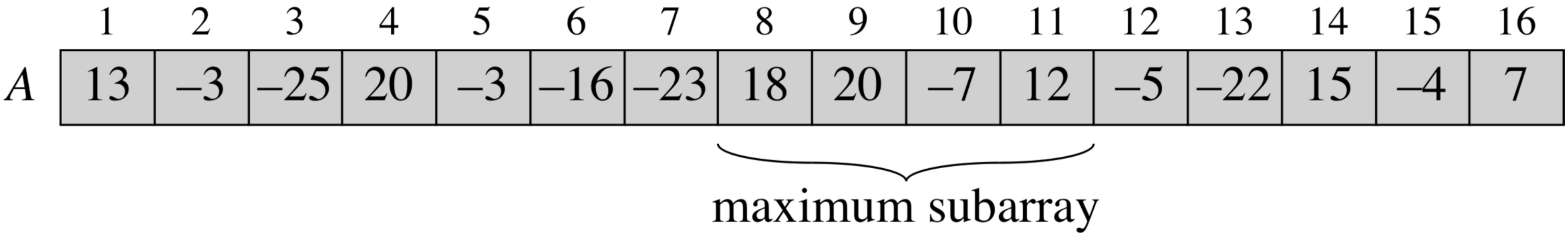
example of recurrence:

$T(n) = T(n-1) + 1$, if $n>1$

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

Example I — The Maximum-subarray problem

The Maximum-subarray problem



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

How can we get maximum sum of any non-empty contiguous subarray in A?

the array must include some negative numbers

The Maximum-subarray problem (Cont'd)

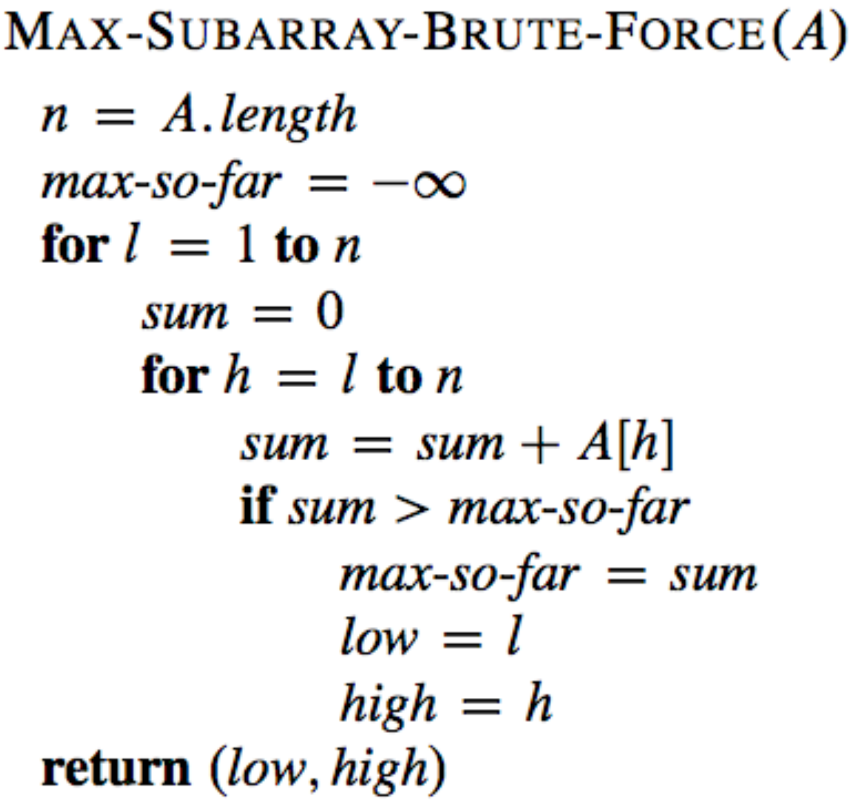
Brute-force Approach

a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved.

usually the one that is easiest to apply

The Maximum-subarray problem (Cont’d)

Brute-force Approach Algorithm (by exhaustion)



```
MAX-SUBARRAY-BRUTE-FORCE($A$)
 $n = A.length$
 $max\text{-}so\text{-}far = -\infty$
 for $l = 1$ to $n$
     $sum = 0$
     for $h = l$ to $n$
         $sum = sum + A[h]$
         if $sum > max\text{-}so\text{-}far$
             $max\text{-}so\text{-}far = sum$
             $low = l$
             $high = h$
 return ($low, high$)
```

look at the line that needs the most times of execution: $\Theta(n^2)$

Can we do better than this?

The Maximum-subarray problem — Divide & Conquer Approach

Divide A[*low*..*high*] into two subarrays: A[*low*..*mid*] and A[*mid+1*..*high*]

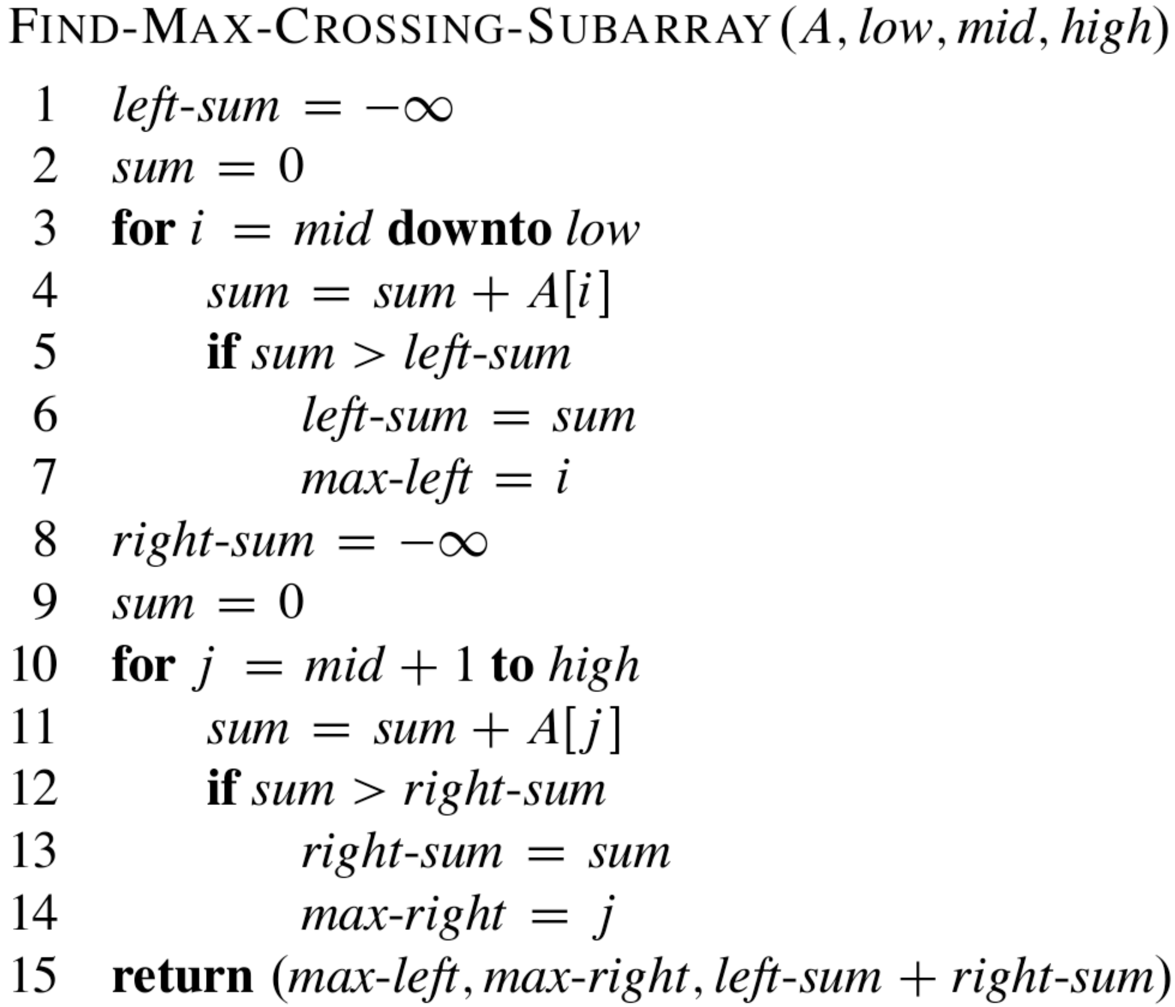
Conquer by finding a maximum subarrays of A[*low*..*mid*] and A[*mid+1*..*high*] (recursively)

Combine by finding a maximum subarray that crosses the midpoint, and using the best solution out of the three (the subarray crossing midpoint and the two solutions found in the conquer step)

Prof. Michael Shmos

The Maximum-subarray problem (Cont’d)

Algorithm to find the crossing subarray



```
FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$
 1  $left\text{-}sum = -\infty$
 2  $sum = 0$
 3  for $i = mid$ downto $low$
 4      $sum = sum + A[i]$
 5      if $sum > left\text{-}sum$
 6          $left\text{-}sum = sum$
 7          $max\text{-}left = i$
 8  $right\text{-}sum = -\infty$
 9  $sum = 0$
10  for $j = mid + 1$ to $high$
11      $sum = sum + A[j]$
12      if $sum > right\text{-}sum$
13          $right\text{-}sum = sum$
14          $max\text{-}right = j$
15  return $(max\text{-}left, max\text{-}right, left\text{-}sum + right\text{-}sum)$
```

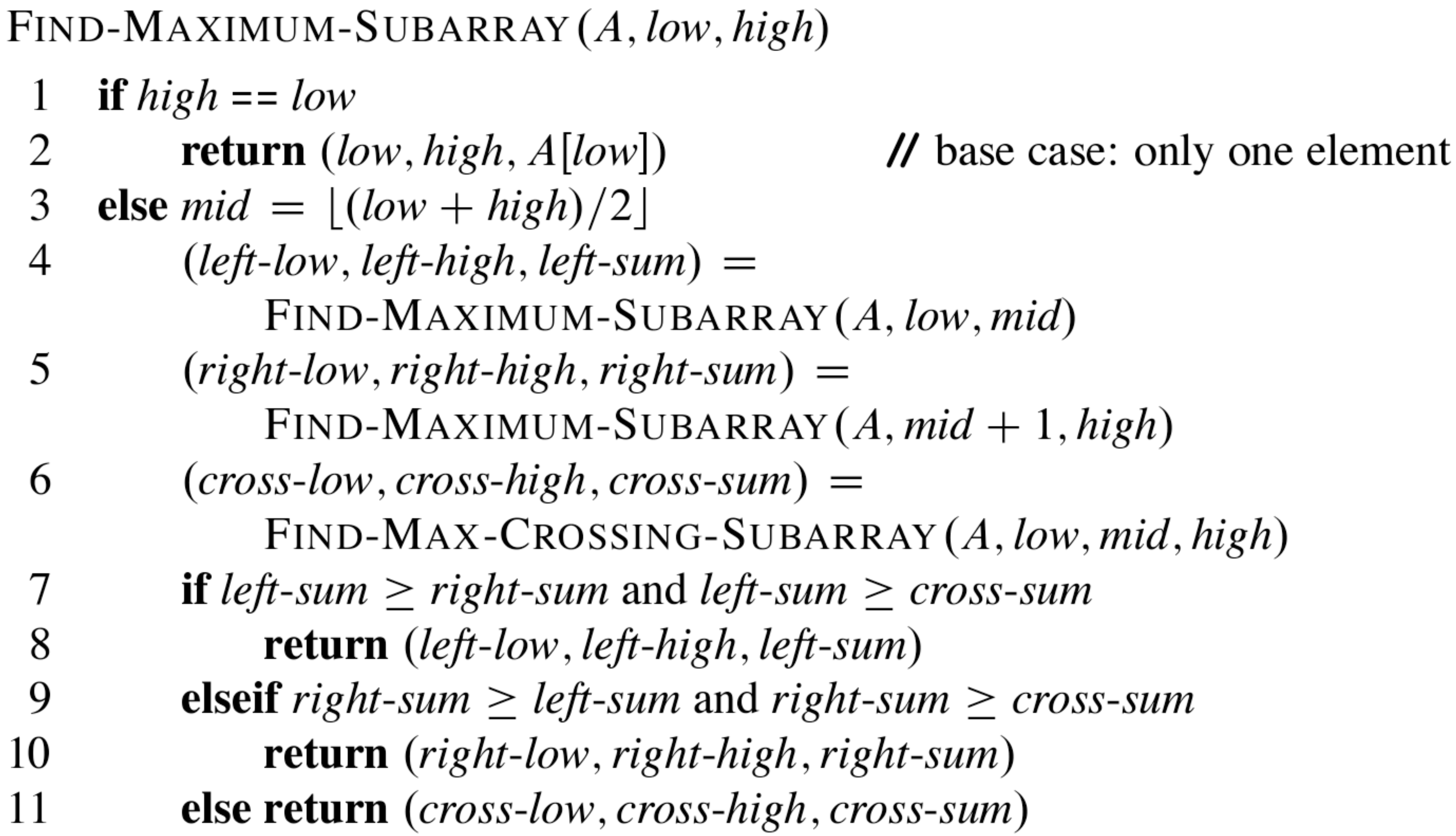
line 1~7: find a maximum subarray of A[i..mid]

line 8~14: find a maximum subarray of A[mid+1..j]

$\Theta(n)$

The Maximum-subarray problem (Cont’d)

Algorithm to solve the whole problem



```
FIND-MAXIMUM-SUBARRAY($A, low, high$)
if $high == low$
    return ($low, high, A[low]$)          // base case: only one element
else $mid = \lfloor (low + high)/2 \rfloor$
    ($left\text{-}low, left\text{-}high, left\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A, low, mid$)
    ($right\text{-}low, right\text{-}high, right\text{-}sum$) =
        FIND-MAXIMUM-SUBARRAY($A, mid + 1, high$)
    ($cross\text{-}low, cross\text{-}high, cross\text{-}sum$) =
        FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$)
    if $left\text{-}sum \geq right\text{-}sum$ and $left\text{-}sum \geq cross\text{-}sum$
        return ($left\text{-}low, left\text{-}high, left\text{-}sum$)
    elseif $right\text{-}sum \geq left\text{-}sum$ and $right\text{-}sum \geq cross\text{-}sum$
        return ($right\text{-}low, right\text{-}high, right\text{-}sum$)
    else return ($cross\text{-}low, cross\text{-}high, cross\text{-}sum$)
```

The Maximum-subarray problem (Cont’d)

Divide and Conquer Algorithm Asymptotic Analysis:

Base case (line1~2): Θ(1)

Recursive case:

dividing (line 3): Θ(1)

conquer subproblem (line 4~5): 2T(n/2)

combing (line 7~11): Θ(n) + Θ(1)

T(n) = Θ(1) + Θ(1) + 2T(n/2) + Θ(n) + Θ(1)

= 2T(n/2) + Θ(n) (*absorb Θ(1) terms into Θ(*

The Maximum-subarray problem (Cont'd)

Divide and Conquer Algorithm Analysis:

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

T(n) = Θ(*n* lgn) (same recurrence as for merge sort)

Better than θ(n^2)

Can we do better than this?

The Maximum-subarray problem (Cont'd)

Actually, there is an algorithm with linear time, by Prof. Joseph Kadane

A classic interview question —

think about the solution by yourself!

Homework 3

Due Date: March 1st, 2019 (F)

BEFORE the class starts

Honor Statement needs to be enclosed for each assignment, otherwise the homework will not be graded

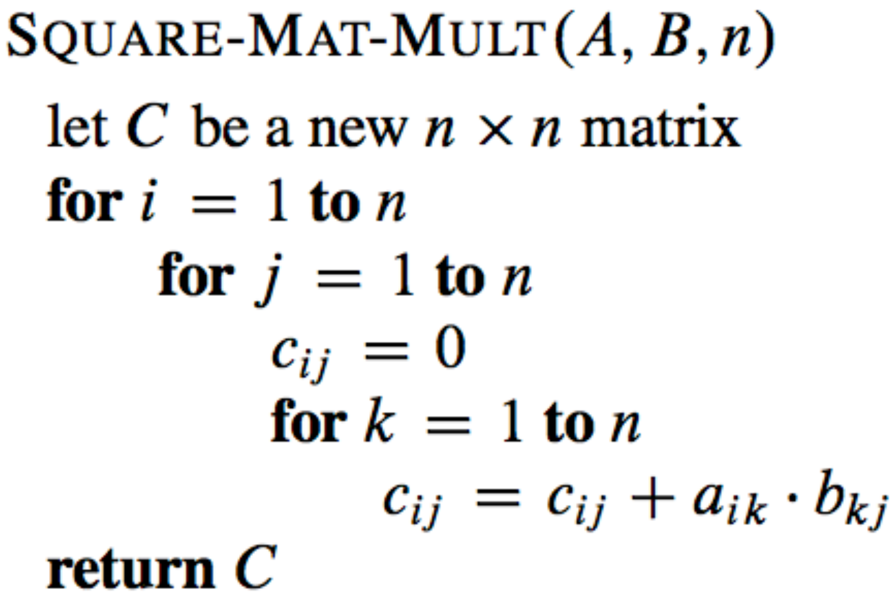
Example II — Matrix Multiplication

Matrix Multiplication

a quick review of matrix multiplication

Matrix Multiplication

the regular algorithm



```
SQUARE-MAT-MULT($A, B, n$)
  let $C$ be a new $n \times n$ matrix
  for $i = 1$ to $n$
      for $j = 1$ to $n$
          $c_{ij} = 0$
          for $k = 1$ to $n$
              $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
  return $C$
```

innermost loop, determines the running time of this algorithm, $\Theta(n^3)$

Can we do better than this?

Matrix Multiplication

Divide and Conquer Method (assume n is a power of 2)

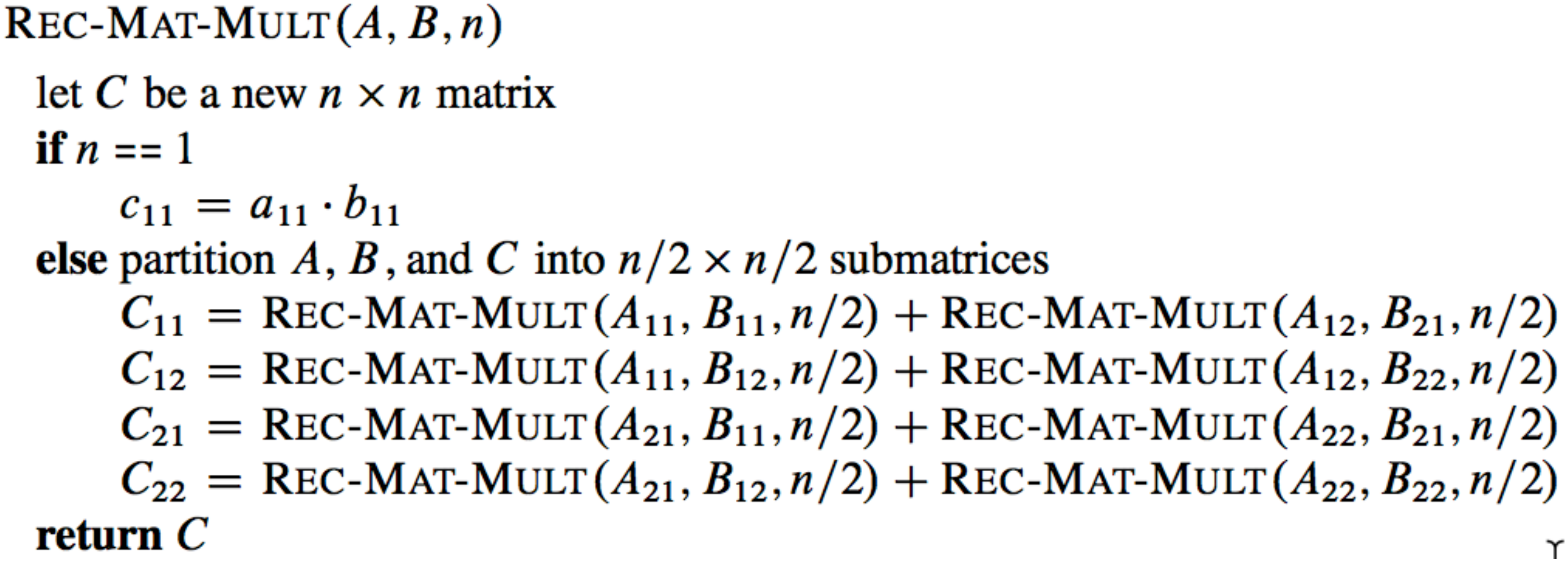
partition each of A, B, into four *n*/2 x *n*/2 matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} , \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} , \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} , \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} . \end{aligned}$$

each multiplies two n/2 x n/2 matrices and then adds their n/2 x n/2 products

Matrix Multiplication - D & C



```
REC-MAT-MULT($A, B, n$)
  let $C$ be a new $n \times n$ matrix
  if $n == 1$
      $c_{11} = a_{11} \cdot b_{11}$
  else partition $A$, $B$, and $C$ into $n/2 \times n/2$ submatrices
      $C_{11} =$ REC-MAT-MULT($A_{11}, B_{11}, n/2$) $+$ REC-MAT-MULT($A_{12}, B_{21}, n/2$)
      $C_{12} =$ REC-MAT-MULT($A_{11}, B_{12}, n/2$) $+$ REC-MAT-MULT($A_{12}, B_{22}, n/2$)
      $C_{21} =$ REC-MAT-MULT($A_{21}, B_{11}, n/2$) $+$ REC-MAT-MULT($A_{22}, B_{21}, n/2$)
      $C_{22} =$ REC-MAT-MULT($A_{21}, B_{12}, n/2$) $+$ REC-MAT-MULT($A_{22}, B_{22}, n/2$)
  return $C$
```

Let’s analyze the running time

Matrix Multiplication — D & C

Divide and Conquer Algorithm Analysis:

- Base case: Θ(1)
- Recursive case:
  - dividing: Θ(1)
  - conquer subproblem: 8 recursive calls, 8T(n/2)
  - combing: Θ(n^2) to add n/2 x n/2 matrices four times

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases}$$

T(n) = Θ(n^3) (Not better than brute-force) ☹

Matrix Multiplication — D & C (Cont’d)

Our attempt to use Divide and Conquer failed

FAIL — “First Attempt in Learning”

Don’t give up, try again!

Matrix Multiplication — Strassen's method

runs in $\Theta(n^{\lg 7})$ time ⇒ $O(n^{2.81})$

$(2.80 \le \lg 7 \le 2.81)$

Idea: perform 7 recursive multiplications of n/2 x n/2 matrices, rather than 8.

Matrix Multiplication — Strassen's method (Cont'd)

Algorithm:

Step 1. partition each of the matrices into four n/2 x n/2 sub-matrices, i.e., A_{11}, A_{12}, A_{21}, A_{22}

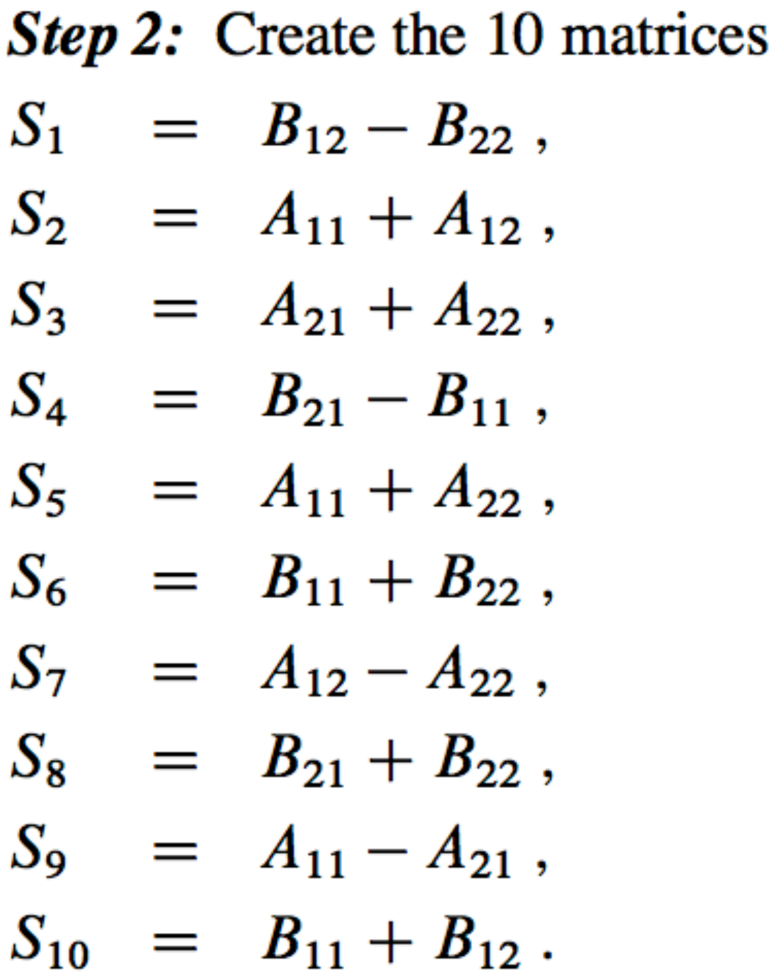
Time: Θ(1)

Matrix Multiplication — Strassen's method (Cont'd)

Step 2. create 10 matrices $S_{1,}$ $S_{2, \dots,}$ S_{10}.

Each is n/2 x n/2 and is the sum or difference of two matrices created in previous step.

Time: $\Theta(n^2)$ to create all 10 matrices.

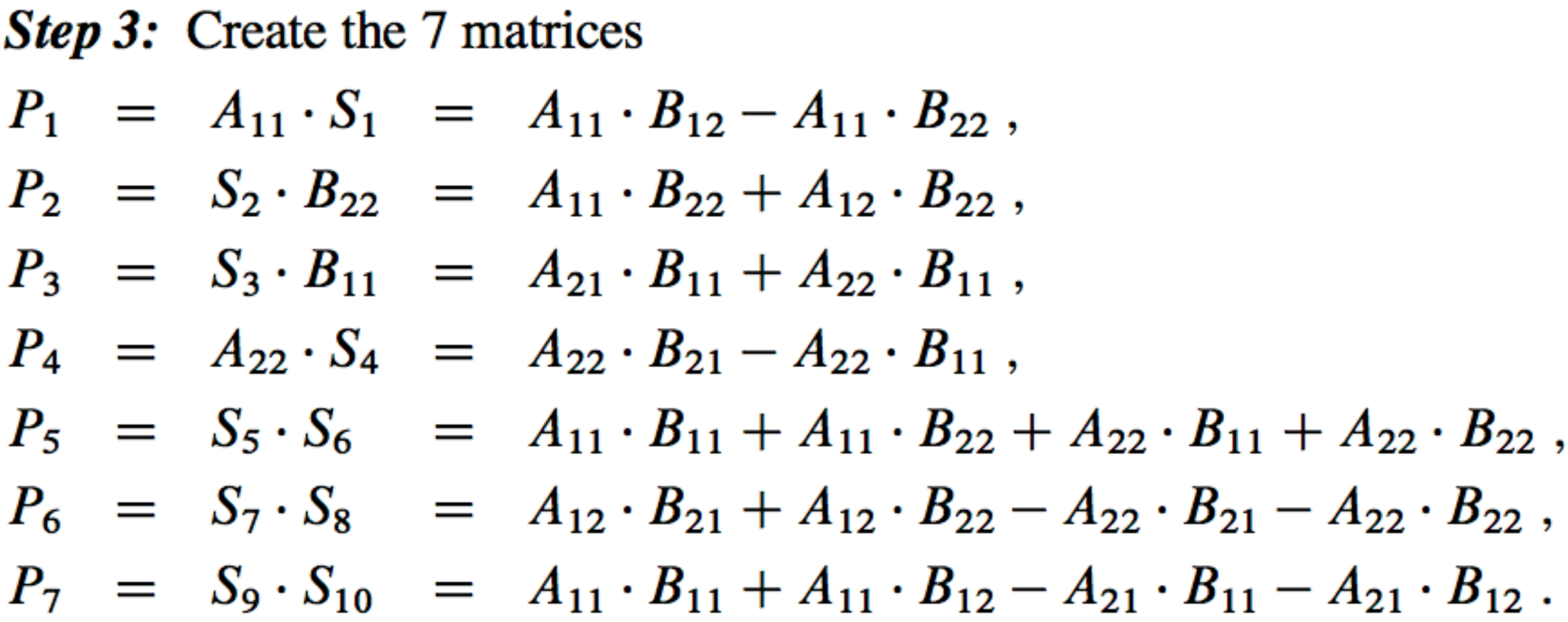


***Step* 2:** Create the 10 matrices

$$\begin{aligned}
S_1 &= B_{12} - B_{22} \,, \\
S_2 &= A_{11} + A_{12} \,, \\
S_3 &= A_{21} + A_{22} \,, \\
S_4 &= B_{21} - B_{11} \,, \\
S_5 &= A_{11} + A_{22} \,, \\
S_6 &= B_{11} + B_{22} \,, \\
S_7 &= A_{12} - A_{22} \,, \\
S_8 &= B_{21} + B_{22} \,, \\
S_9 &= A_{11} - A_{21} \,, \\
S_{10} &= B_{11} + B_{12} \,.
\end{aligned}$$

Matrix Multiplication — Strassen's method (Cont'd)

Step 3. Recursively compute 7 matrix products P_1, P_2, …,P_7, each n/2 x n/2.



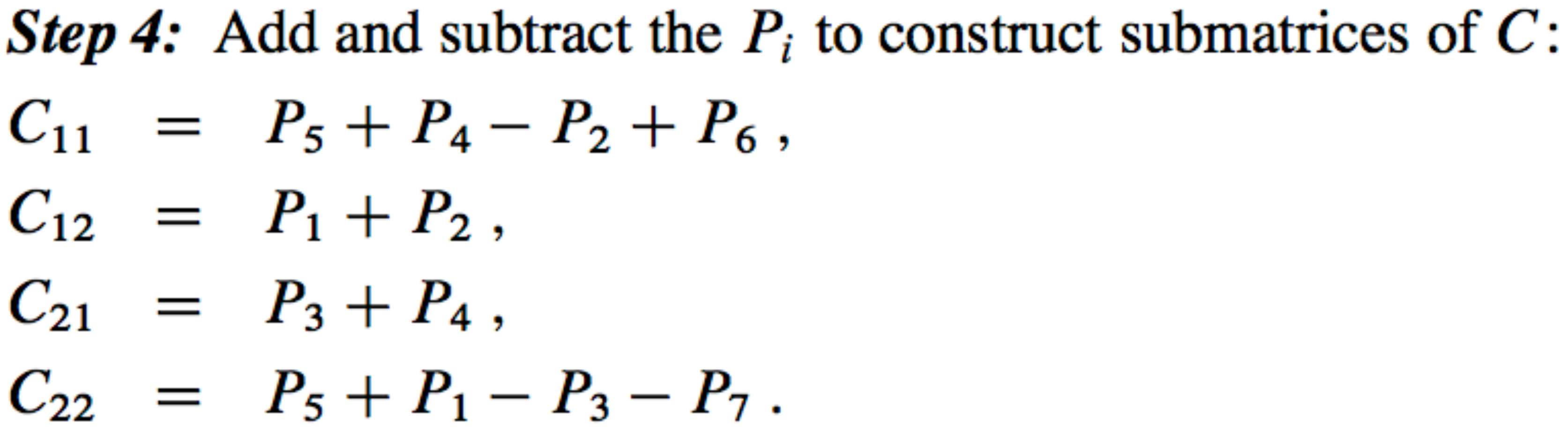
Step 3: Create the 7 matrices

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22}\ , \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22}\ , \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11}\ , \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11}\ , \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}\ , \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}\ , \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}\ .
\end{aligned}
$$

Time: 7T(n/2)

Matrix Multiplication — Strassen’s method (Cont’d)

Step 4. Compute n/2 x n/2 sub-matrices of C by adding and subtracting various combinations of the P_i .



Step 4: Add and subtract the P_i to construct submatrices of C:

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 , \\ C_{12} &= P_1 + P_2 , \\ C_{21} &= P_3 + P_4 , \\ C_{22} &= P_5 + P_1 - P_3 - P_7 . \end{aligned}$$

Time: $\Theta(n^2)$

Matrix Multiplication — Strassen's method (Cont'd)

Asymptotic Analysis:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \,, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \,. \end{cases}$$

runs in $\Theta(n^{\lg 7})$ time ⇒ $O(n^{2.81})$ $(2.80 \leq \lg 7 \leq 2.81)$

Matrix Multiplication — Strassen's method (Cont'd)

Notes about Strassen's method

the first to beat $\Theta(n^3)$time — a clever method

sub-matrices consume space

asymptotically fastest known algorithm: Coppersmith and Winograd $O(n^{2.376})$

Recurrences need to solve

Merge Sort, Maximum Subarray:

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

Simple Divide-Conquer for Matrix Multiplication

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

Strassen’s Method for Matrix Multiplication

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

Divide and Conquer Summary

- understand procedure of writing D&C algorithm
  - maximum-subarray problem
  - matrix multiplication
  - write algorithm using D&C
- analysis of running time using recurrence

Methods to solve recurrence

Methods to solve recurrence

Unfortunately, no general methods to solve recurrence

- Substitution method
- Recursion trees
- Master theorem (Master method)

Method 1: Substitution method

Step 1: Guess the solution

Step 2: Use mathematical induction to find the constants and show that the solution works

a powerful method, but how can we guess it correctly?

recursion tree, or Master Method

Method 1: Substitution method (Cont’d)

Write T(n) with recurrence and Θ-notation

Use substitution method to establish either upper or lower bounds on recurrence.

- to prove a tight Θ bound, show the upper (O) and lower (Ω) bounds separately
- might need to use different constants for each bound

Method 1: Substitution method (Cont'd)

Examples (notes)

- $T(n) = 2T(n/2) + n$
- $T(n) = 4T(n/4) + n$
- $T(n) = 8T(n/2) + \Theta(n^2)$ (subtract off a lower-order term)

Method 1: Substitution method (Cont'd)

Making a good guess

- experience, creativity, heuristics (e.g., use a recursion tree to generate a good guess)

Other techniques that may be used

- subtract off a lower-order term, e.g., example 2
- changing variables (e.g., $T(n) = 2T(n^{1/2}) + \lg n$) — self study

Method 2: Recursion Tree

Method 2: Recursion Tree

Help to generate a good guess, then verify by substitution method

we can often tolerate a small amount of "sloppiness", since we'll verify the guess later

Method 2: Recursion Tree (Cont'd)

Method:

each node: the cost of a single subproblem somewhere in the set of recursive function invocations

sum the costs within each level of the tree to obtain a set of per-level costs

sum all the per-level costs to determine the total cost of all levels of the recursion

Method 2: Recursion Tree (Cont'd)

Examples:

$T(n) = 3T(n/3) + cn^2$

$T(n) = 3T(n/4) + \Theta(n^2)$, textbook: p88~90, self study

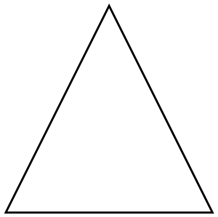
$T(n) = T(n/3) + T(2n/3) + \Theta(n)$ (notes)

$T(n) = T(n-1) + n$ (notes)

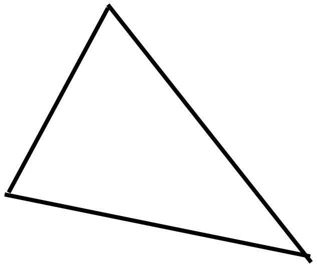
Method 2: Recursion Tree (Cont’d)

Recursion Tree provides a visual picture of the recurrence cost

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



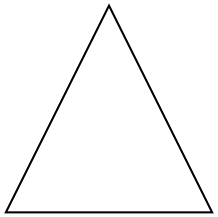
$T(n) = T(n/3) + T(2n/3) + \Theta(n)$



Method 2: Recursion Tree (Cont’d)

When subproblem size is equal: T(n) = aT(n/b) + f(n)

- the subproblem size at level *i* (i = 0, 1, …) is n/b^i
    - the subproblem size at the last level is 1, so $n=b^i$, $i = \log_b n$, the index for last level is $\log_b n$
- height of the tree: $\log_b n+1$
- *the number of* subproblems at level *i* is a^i
    - the number of leaves at the last level is a^$\log_b n$, where $i=\log_b n$



Method 2: Recursion Tree (Cont'd)

When subproblem size is NOT equally split,

e.g., $T(n) = T((1-\alpha)n) + T(\alpha n) + \Theta(n)$, $(1/2 \leq \alpha < 1)$

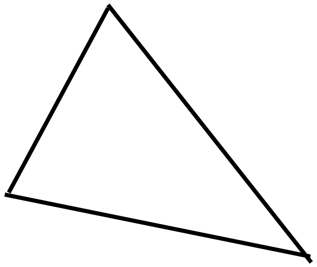
the tree is not balanced

the subproblem size at level *i* ($i = 0, 1, \dots$) is $n(1-\alpha)^i$ and $n\alpha^i$

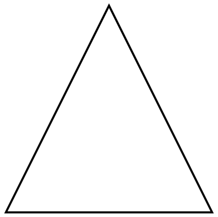
the tree is full for $\log_{1/(1-\alpha)} n$ levels, each contributing cn

the leftmost branch has $\log_{1/(1-\alpha)} n$ levels

the rightmost branch has $\log_{1/\alpha} n$ levels



Method 3: Master Method



Method 3: Master Method

a “cookbook” for recurrences in form:

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

- T(n): the running time of an algorithm (problem size n)
- divide the problem into *a* subproblems, each of size is n/b
- the subproblems are resolved recursively

only applies to the problem when <u>subproblem sizes are equal</u>

Method 3: Master Method (Cont'd)

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

<u>T(n/b)</u>: running time to solve each subproblem recursively

floor and celling of n/b will not affect the asymptotic behavior of the recurrence

<u>f(n)</u>: encompasses the cost of dividing the problem and combing the results of the subproblems

e.g., Strassen's algorithm, $a = 7$, $b = 2$, and $f(n) = \Theta(n^2)$

Method 3: Master Method (Cont'd)

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

a: the number of subproblems (the number of recursive calls), **≥ 1**

n/b: the size of each subproblem

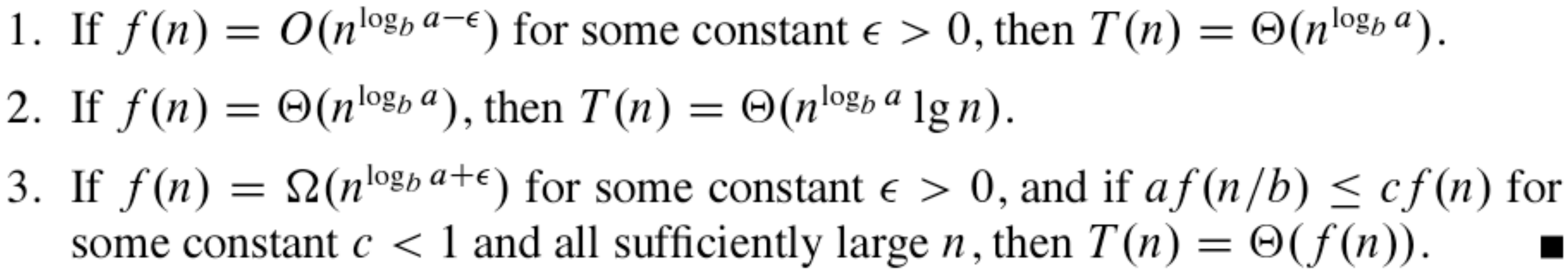
b: input size shrinkage factor, **>1**

f(n): **asymptotically positive function**

Method 3: Master Method (Cont'd)

Master theorem

T(n) has the following asymptotic bounds, in 3 cases:



1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Method 3: Master Method — Case 1

What does the theorem mean?

we want to Compare $n^{\log_b a}$ vs. $f(n)$

case 1: $f(n) = O(n^{\log_b a - \varepsilon})$

f(n) polynomially smaller than n ^log$_b$a by a factor of n$^\varepsilon$ for some constant ε>0
recursion tree: cost is dominated by leaves

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

Method 3: Master Method — Case 2

case 2: $f(n) = \Theta(n^{\log_b a})$

recursion tree: cost of each level contributes equally (n ^ log$_b$a), there are lg*n* levels

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

Method 3: Master Method — Case 3

case 3: $f(n) = \Omega(n^{\log_b a+\varepsilon})$

f(n) asymptotically greater than n ^$\log_b$a by a factor of n^{ε} for some constant ε>0

cost is donated by root

$$\mathrm{T}(n) = \Theta(f(n))$$

Method 3: Master Method (Cont'd)

Case 3 regularity condition:

generally not a problem

always holds whenever f(n) is a polynomial function

f(n) = n^k and $f(n) = \Omega(n^{\log_b a+\varepsilon}), \varepsilon > 0$

Method 3: Master Method (Cont'd)

Examples:

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

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

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

$T(n) = 3T(n/3) + 2^{\log_4 n}$

$T(n) = 7T(n/3) + n^2$

$T(n) = 3T(n/3) + cn^2$

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

Method 3: Master Method (Cont’d)

Examples (Mater Theorem doesn’t apply):

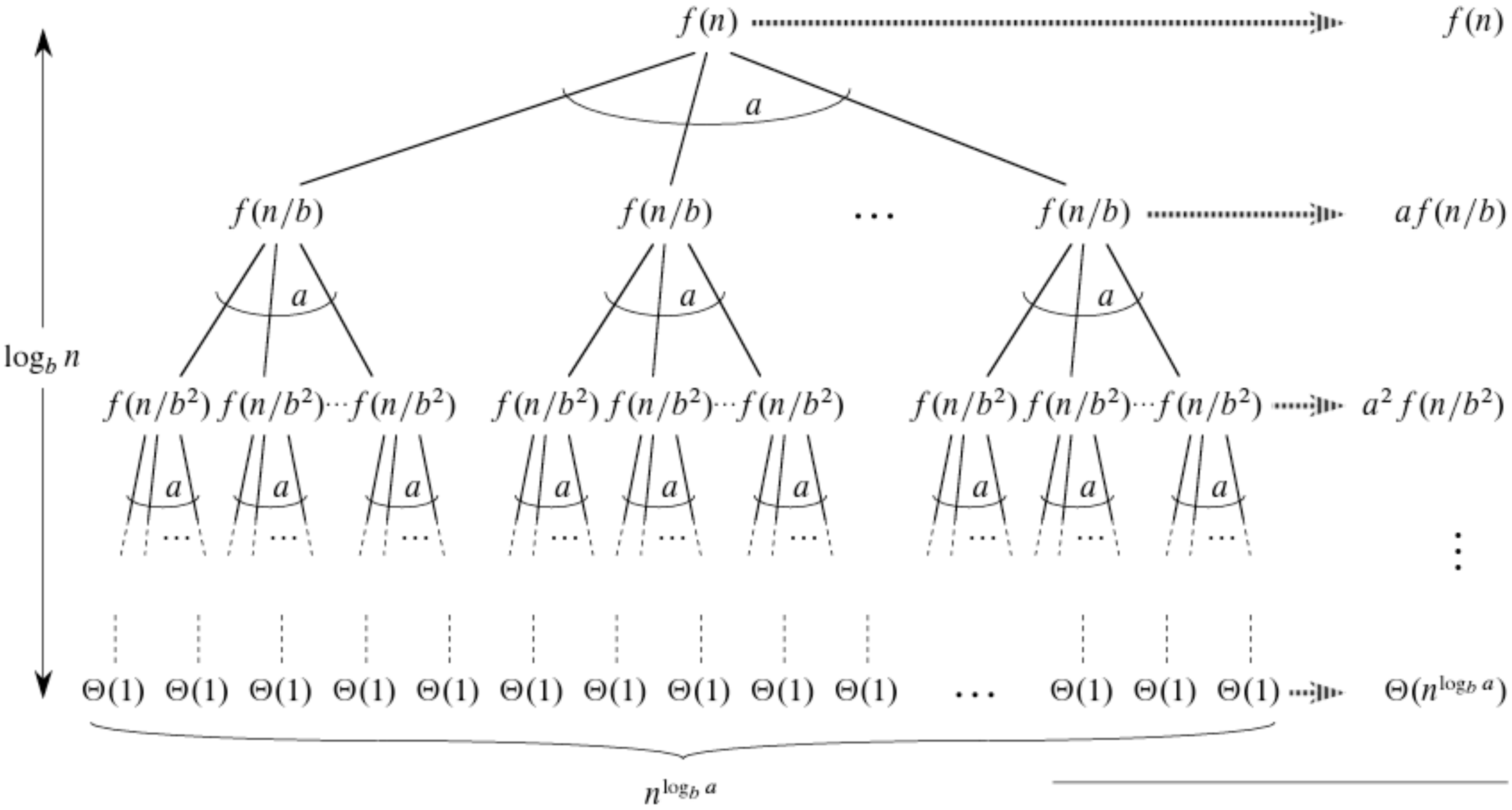
$T(n) = 0.5T(n/2) + 1/n$

$T(n) = 64T(n/8) - n^2$

$T(n) = 27T(n/3) + \Theta(n^3/\lg n)$

$T(n) = \lg n\, T(n/2) + n^3$

$T(n) = 2T(n/2) + n\lg n$



the cost of leaves: the cost of doing all subproblems of size 1

the cost of internal nodes: the cost of dividing problems into subproblems and then recombining the subproblems.

$$\text{Total: } \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$

cost of leaves

cost of internal nodes

Master Method and Recursion Tree

Decide which case of MM and draw recursion tree for each one, and compare:

player1

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

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

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