

Lecture 3

Divide and Conquer

Our Roadmap



- ◆ The idea of *Divide-and-Conquer*
- ◆ Examples for *Divide-and-Conquer*
- ◆ How do we solve recurrences?

Guess the Number Game

◆ Guess the Number Game

- ◆ Host: pick a secret integer X from 1 to 20
- ◆ Guest: guess V as the answer
- Host: “ V is too low” / “ V is too high” / “ V is **correct!**”

◆ Simple strategy: test each integer in ascending order

- ◆ Guess 1 → too low
- ◆ Guess 2 → too low
- ◆
- ◆ Guess 16 → **correct!**



◆ Can you suggest a faster approach?

Guess the Number Game

pick
 $X=16$



◆ Divide-and-conquer strategy

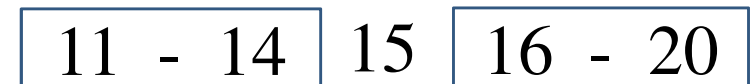
◆ Guess 10 $\rightarrow$ too low

- ◆ [Think] Is X between 1 and 9? NO
- ◆ [Think] Is X between 11 and 20? YES



◆ Guess 15 $\rightarrow$ too low

- ◆ [Think] Is X between 11 and 14? NO
- ◆ [Think] Is X between 16 and 20? YES

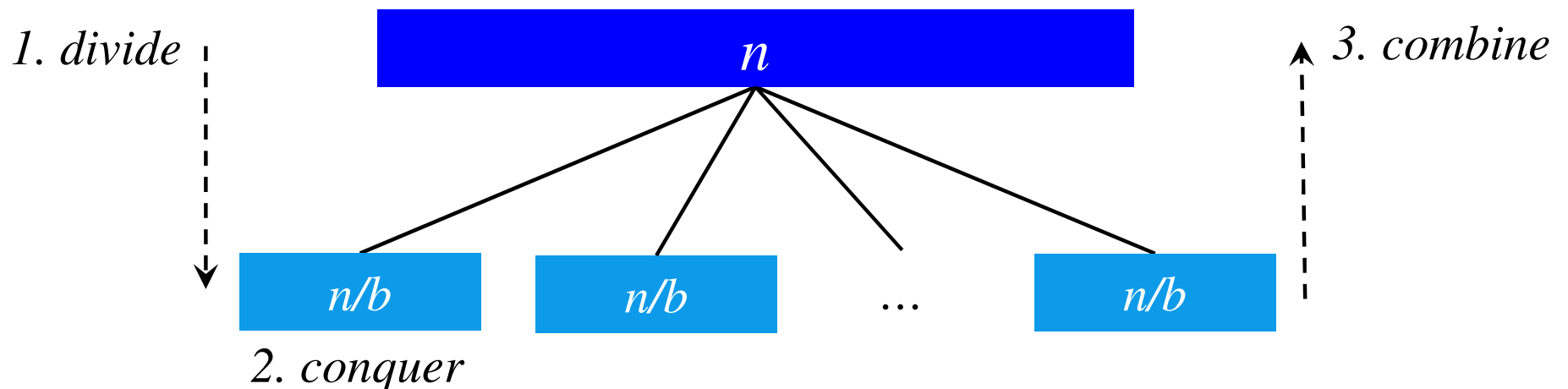


◆ Guess 18 $\rightarrow$ too high

◆ Guess 16 $\rightarrow$ correct!

Divide and Conquer

- ◆ Divide and Conquer: an algorithmic technique
 - ◆ **Divide**: divide the problem into smaller subproblems
 - ◆ **Conquer**: solve each subproblem recursively
 - ◆ **Combine**: combine the solutions of subproblems into the solution of the original problem



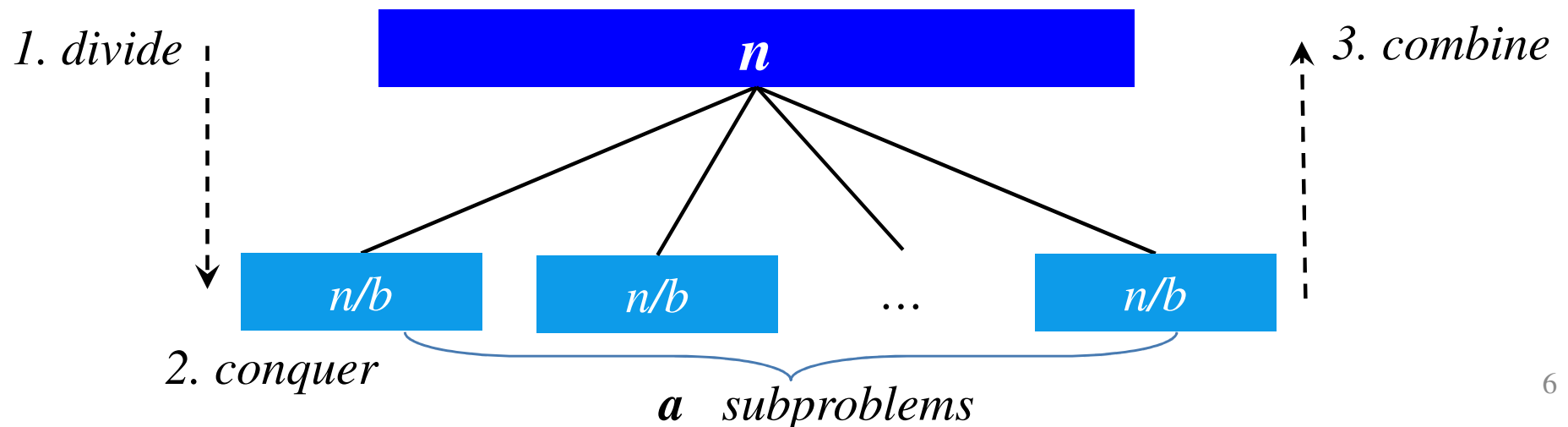
Time Complexity & Recurrence

- ◆ $T(n)$: time complexity of algorithm at input size n
 - ◆ Divide the problem into a subproblems
 - ◆ Size of each subproblem is $1/b$ of the original problem
 - ◆ Suppose that the combine phase takes $f(n)$ time

Note: a and b can have different values

◆ Recurrence

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



Our Roadmap

- ◆ The idea of *Divide-and-Conquer*



- ◆ Examples for *Divide-and-Conquer*

1. Searching
2. The maximum subarray sum
3. Integer multiplication

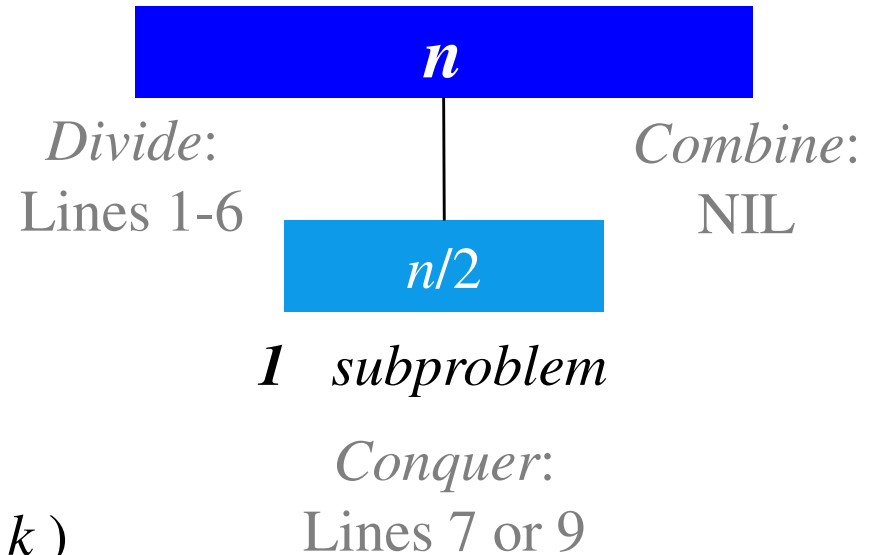
- ◆ How do we solve recurrences?

Binary Search

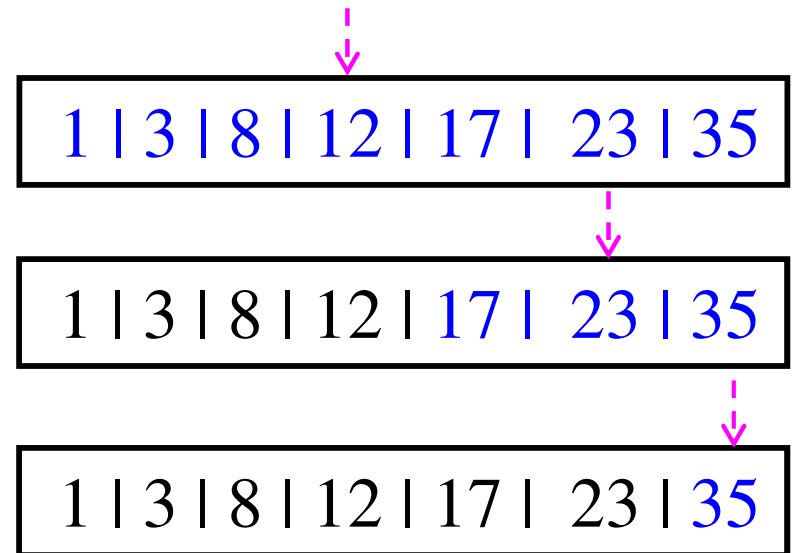
Initial call: BinarySearch (A, 0, $n-1$, k)

BinarySearch(Array A, Integers low , $high$, Key k)

1. $mid \leftarrow \lfloor (low+high)/2 \rfloor$
2. if $A[mid] = k$
3. return mid
4. else if $low = high$
5. return -1
6. if $k < A[mid]$
7. return **BinarySearch**(A, low , $mid-1$, k)
8. else
9. return **BinarySearch**(A, $mid+1$, $high$, k)



Example: find the key “35”



Binary Search: running time analysis

Initial call: BinarySearch (A, 0, $n-1$, k)

BinarySearch(Array A, Integers low , $high$, Key k)

```
1.  $mid \leftarrow \lfloor (low+high)/2 \rfloor$ 
2. if  $A[mid] = k$ 
3.   return  $mid$ 
4. else if  $low = high$ 
5.   return  $-1$ 
6. if  $k < A[mid]$ 
7.   return BinarySearch(A,  $low$ ,  $mid-1$ ,  $k$ )
8. else
9.   return BinarySearch(A,  $mid+1$ ,  $high$ ,  $k$ )
```

- ◆ Let $T(n)$ be the running time
 - ◆ where n is the size of the input subarray $A[low..high]$
 - ◆ Lines 1-6: $O(1)$
 - ◆ Either Line 7 or Line 9 is executed
 - ◆ Line 7: $T(n/2)$
 - ◆ Line 9: $T(n/2)$
- ◆ Thus, we get the recurrence
$$T(n) = T(n/2) + O(1)$$
- ◆ Solving it, we get:
$$T(n) = O(\log n)$$

We will solve this recurrence later

The Maximum Subarray Sum

- ◆ Problem definition

- ◆ *Input:* an array $A[0..n-1]$

- ◆ *Output:* the largest possible sum of $\sum_{i=x..y} A[i]$
 - ◆ among all possible intervals $[x..y]$

- ◆ Application scenario

- ◆ $A[i]$ denotes the stock closing price – the stock opening price

- ◆ You can buy once and then sell once. How to earn the most?

- ◆ Example

- ◆ The largest possible sum is 43 in the following array $A[0..15]$

- ◆ Obtained from the subarray $A[7..10]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

The Maximum Subarray Sum

- ◆ Brute force algorithm
 - ◆ consider each start position x
 - ◆ find the sum from x to each position y (where $y \geq x$)
 - ◆ keep the highest sum
- ◆ Running time: $O(n^2)$

BruteForce(Array A[0..n-1])

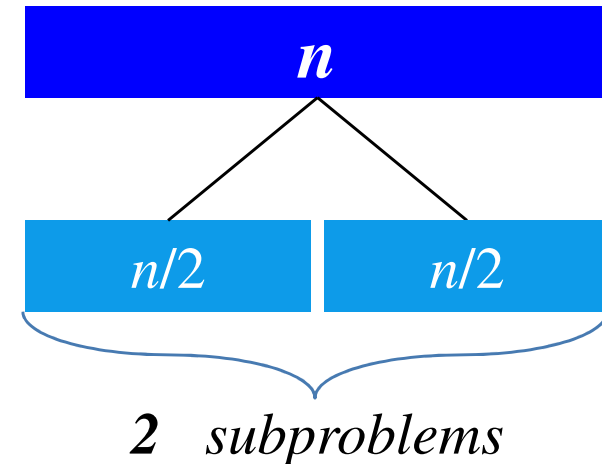
```

1. bestSum  $\leftarrow -\infty$ 
2. for x  $\leftarrow$  0 to n-1
3.     sum  $\leftarrow$  0
4.     for y  $\leftarrow$  x to n-1
5.         sum  $\leftarrow$  sum + A[y]
6.         if (sum > bestSum)
7.             bestSum  $\leftarrow$  sum
8. return bestSum
    
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
$x=0$:	13	10	-15	5	2	-14	-37	-19	1	-6	6	1	-21	-6	-10	-3
$x=1$:		-3	-28	-8	-11	-27	-50	-32	-12	-19	-7	-12	-34	-19	-23	-16
$x=2$:			-25	-5	-8	-24	-47	-29	-9	-16	-4	-9	-31	-16	-20	-13
$x=3$:																

The Maximum Subarray Sum

- ◆ Divide the array into two halves
- ◆ The maximum subarray must be either:
 - ◆ In the left subarray [*solve recursively*]
 - ◆ E.g., Sum of $A[3..3] = 20$
 - ◆ In the right subarray [*solve recursively*]
 - ◆ E.g., Sum of $A[8..10] = 25$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

- ◆ Across both subarrays
 - ◆ E.g., Sum of $A[7..10] = 43$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

The Maximum Subarray Sum

MaxCrossSum (Array A, Integers low,mid,high)

```
1. leftSum  $\leftarrow -\infty$ 
2. sum  $\leftarrow 0$ 
3. for i  $\leftarrow$  mid downto low
4.     sum  $\leftarrow$  sum + A[i]
5.     if (sum > leftSum)
6.         leftSum  $\leftarrow$  sum
7. rightSum  $\leftarrow -\infty$ 
8. sum  $\leftarrow 0$ 
9. for j  $\leftarrow$  mid+1 to high
10.    sum  $\leftarrow$  sum + A[j]
11.    if (sum > rightSum)
12.        rightSum  $\leftarrow$  sum
13. return (leftSum+rightSum)
```

- ◆ How to find the highest cross sum?
- ◆ Sum from mid to the left
 - ◆ Keep the highest sum: leftSum
- ◆ Sum from mid+1 to the right
 - ◆ Keep the highest sum: rightSum

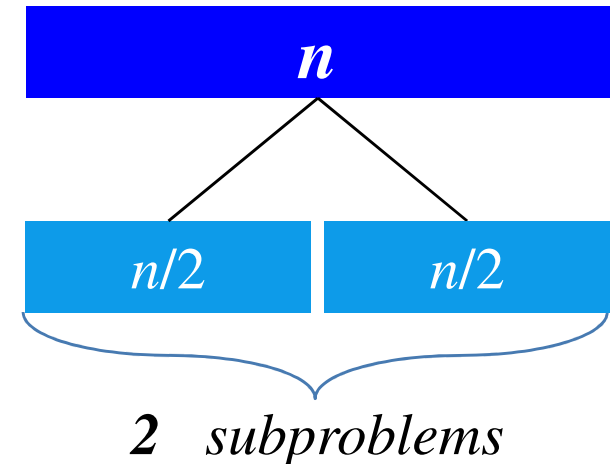
				←-----				-----→							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

The Maximum Subarray Sum

MaxSubarraySum (Array A, Integers low,high)

```

1.  if high=low
2.      return A[low]
3.  else
4.      mid  $\leftarrow \lfloor (low+high)/2 \rfloor$ 
5.      leftSum  $\leftarrow$  MaxSubarraySum(A,low,mid)
6.      rightSum  $\leftarrow$  MaxSubarraySum(A,mid+1,high)
7.      crossSum  $\leftarrow$  MaxCrossSum(A,low,mid,high)
8.      return max(leftSum,rightSum,crossSum)
    
```



Divide: Lines 1-4
Conquer: Lines 5-6
Combine: Line 7-8

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

The Maximum Subarray Sum

MaxSubarraySum (Array A, Integers low,high)

```
1. if high=low
2.   return A[low]
3. else
4.   mid  $\leftarrow \lfloor (low+high)/2 \rfloor$ 
5.   leftSum  $\leftarrow$  MaxSubarraySum(A,low,mid)
6.   rightSum  $\leftarrow$  MaxSubarraySum(A,mid+1,high)
7.   crossSum  $\leftarrow$  MaxCrossSum(A,low,mid,high)
8.   return max(leftSum,rightSum,crossSum)
```

- ◆ Let $T(n)$ be the running time
 - ◆ where n is the size of the input subarray $A[low..high]$
 - ◆ Lines 1-4: $O(1)$
 - ◆ Line 5: $T(n/2)$
 - ◆ Line 6: $T(n/2)$
 - ◆ Line 7: $O(n)$
 - ◆ Line 8: $O(1)$
- ◆ Thus, we get the recurrence
 - ◆ $T(n) = 2 T(n/2) + O(n)$
- ◆ Solving it, we get:
 - ◆ $T(n) = O(n \log n)$

Multiplication Problem

- ◆ Multiply two n -bit numbers (very large numbers)
 - ◆ Application: cryptography
 - ◆ Represent each number as a length- n bit array
 - ◆ It takes $O(1)$ time to access a bit
- ◆ Algorithms
 - ◆ Long-multiplication (from school): $O(n^2)$
 - ◆ Simple divide-and-conquer method: $O(n^2)$
 - ◆ Improved divide-and-conquer method: $O(n^{1.585})$
 - ◆ Karatsuba algorithm
- ◆ Interesting questions:
 - ◆ Why “divide-and-conquer” still takes $O(n^2)$ time?
 - ◆ How to achieve $O(n^{1.585})$ running time?

Long Multiplication

Long-Multiply ($A[0..n-1]$, $B[0..n-1]$)

1. create an array $R[0..2n]$, entries as zero

2. for $j \leftarrow 0$ to $n-1$

3. $c \leftarrow 0$

4. for $i \leftarrow 0$ to $n-1$

5. $c \leftarrow c + R[i+j] + A[i] \cdot B[j]$

6. $R[i+j] \leftarrow c \bmod 2$

7. $c \leftarrow \lfloor c/2 \rfloor$

8. $R[j+n] \leftarrow c$

3 2 1 0

A 1 | 0 | 1 | 1

B 1 | 1 | 0 | 1

1 | 0 | 1 | 1

0 | 0 | 0 | 0

1 | 0 | 1 | 1

1 | 0 | 1 | 1

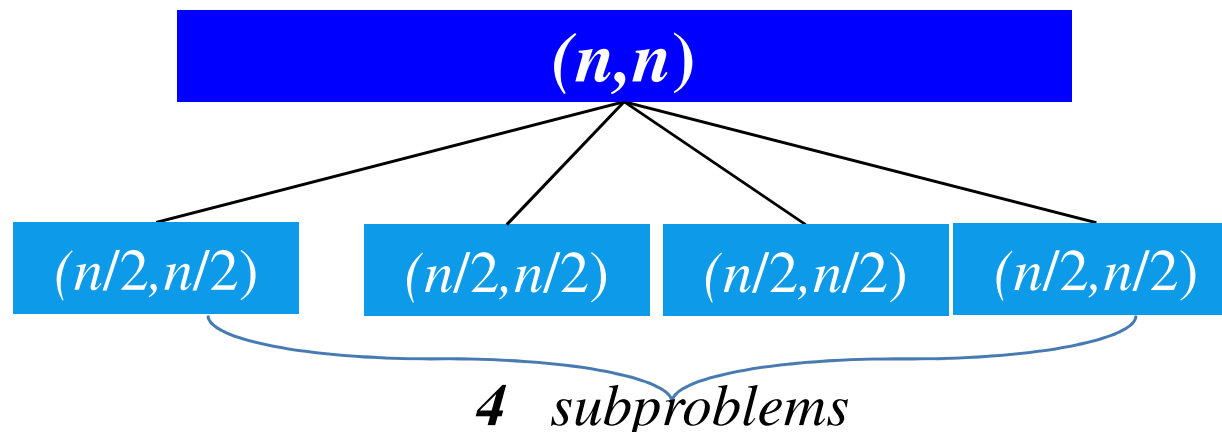
R 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1

Time = $O(n^2)$

Multiplication Problem

- ◆ Multiply two n -bit numbers $A[0..n-1]$, $B[0..n-1]$
 - ◆ Each number is stored as an length- n bit array
- ◆ What is a subproblem?
 - ◆ Multiply two smaller numbers: subarrays $A[l..r]$ and $B[l'..r']$
- ◆ Partition a n -bit number into two $\frac{1}{2}n$ -bit numbers
- ◆ Observe that: $A \times B =$

$$\underbrace{B_1 \times A_1}_{\text{subproblem}} \times \underbrace{2^0}_{\text{shift}} + \underbrace{B_1 \times A_2}_{\text{subproblem}} \times \underbrace{2^{n/2}}_{\text{shift}} + \underbrace{B_2 \times A_1}_{\text{subproblem}} \times \underbrace{2^{n/2}}_{\text{shift}} + \underbrace{B_2 \times A_2}_{\text{subproblem}} \times \underbrace{2^n}_{\text{shift}}$$



Simple Divide-and-Conquer Method

◆ Partition a n -bit number into two $\frac{1}{2}n$ -bit numbers

◆ Observe that: $A \times B =$

$$\underbrace{B_1 \times A_1}_{2^0} + \underbrace{B_1 \times A_2}_{2^{n/2}} + \underbrace{B_2 \times A_1}_{2^{n/2}} + \underbrace{B_2 \times A_2}_{2^n}$$

Recur-Multiply($A[0..n-1]$, $B[0..n-1]$)

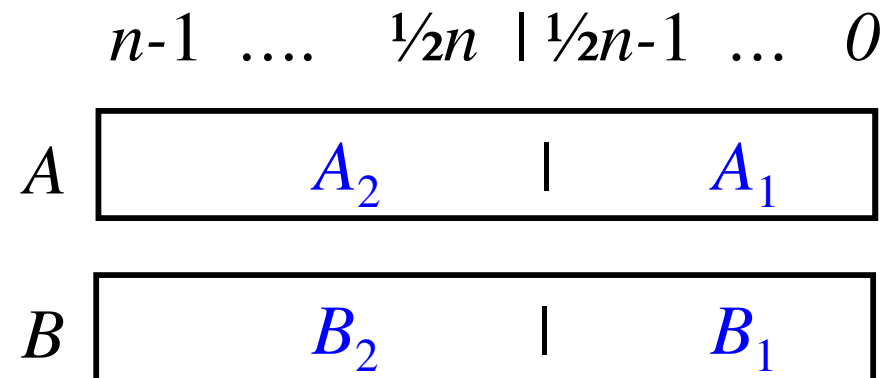
1. let $A = A_2 \circ A_1$; let $B = B_2 \circ B_1$
2. $R_{11} \leftarrow \text{Recur-Multiply}(B_1, A_1)$
3. $R_{12} \leftarrow \text{Recur-Multiply}(B_1, A_2)$
4. $R_{21} \leftarrow \text{Recur-Multiply}(B_2, A_1)$
5. $R_{22} \leftarrow \text{Recur-Multiply}(B_2, A_2)$
6. $R \leftarrow R_{11} + (R_{12} \ll (n/2)) + (R_{21} \ll (n/2))$
 $\quad + (R_{22} \ll n)$
7. return R

$\ll$: left bit-shift operation

Divide: Line 1

Conquer: Lines 2-5

Combine: Line 6



Running Time of “Recur-Multiply”

Recur-Multiply($A[0..n-1]$, $B[0..n-1]$)

1. let $A = A_2 \circ A_1$; let $B = B_2 \circ B_1$
2. $R_{11} \leftarrow \text{Recur-Multiply}(B_1, A_1)$
3. $R_{12} \leftarrow \text{Recur-Multiply}(B_1, A_2)$
4. $R_{21} \leftarrow \text{Recur-Multiply}(B_2, A_1)$
5. $R_{22} \leftarrow \text{Recur-Multiply}(B_2, A_2)$
6. $R \leftarrow R_{11} + (R_{12} \ll (n/2)) + (R_{21} \ll (n/2))$
 $+ (R_{22} \ll n)$
7. return R

Bit-shift operations,
done in $O(n)$ time

- ◆ Let $T(n)$ be the running time
 - ◆ Line 1: $O(n)$
 - ◆ Each of Lines 2, 3, 4, 5: $T(n/2)$
 - ◆ Line 6 uses bit-shift operations and additions $\rightarrow O(n)$ time
- ◆ Thus, we get the recurrence
- ◆ Solving it, we get:

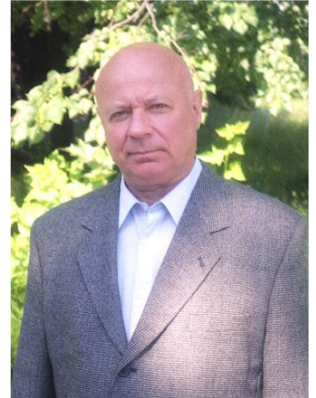
$$T(n) = 4 T(n/2) + O(n)$$

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

We will solve this recurrence later

Karatsuba Algorithm: Improved Divide-and-Conquer Multiplication Method

proposed by
Anatoly Karatsuba
in 1960



◆ Recall that: $A \times B =$

$$\underbrace{B_1 \times A_1}_{2^0} + \underbrace{B_1 \times A_2}_{2^{n/2}} + \underbrace{B_2 \times A_1}_{2^{n/2}} + \underbrace{B_2 \times A_2}_{2^n}$$

◆ Observations

◆ Multiplication (solving subproblem) is **slow**: $T(n/2)$

◆ Addition, subtraction, bit-shift are **fast**: $O(n)$

◆ Idea of Karatsuba algorithm:

◆ Reduce the number of subproblems to 3:

$$\underbrace{(A_1 + A_2) \times (B_1 + B_2)}_{\text{3 subproblems}}, B_1 \times A_1, B_2 \times A_2$$

◆ After solving these subproblems, we compute:

$$\underbrace{B_1 \times A_2 + B_2 \times A_1}_{\text{3 subproblems}} = (A_1 + A_2) \times (B_1 + B_2) - B_1 \times A_1 - B_2 \times A_2$$

Running Time of “Karatsuba”

Karatsuba ($A[0..n-1]$, $B[0..n-1]$)

1. let $A = A_2 \circ A_1$; let $B = B_2 \circ B_1$
2. $A' \leftarrow A_1 + A_2$; $B' \leftarrow B_1 + B_2$
3. $R_0 \leftarrow \text{Karatsuba} (B', A')$
4. $R_1 \leftarrow \text{Karatsuba} (B_1, A_1)$
5. $R_2 \leftarrow \text{Karatsuba} (B_2, A_2)$
6. $R_3 \leftarrow R_0 - R_1 - R_2$
7. $R \leftarrow R_1 + (R_3 \ll (n/2)) + (R_2 \ll n)$
8. return R

$\ll$: left bit-shift operation

Divide: Lines 1-2

Conquer: Lines 3-5

Combine: Lines 6-7

- ◆ Let $T(n)$ be the running time
 - ◆ Lines 1, 2: $O(n)$
 - ◆ Each of Lines 3, 4, 5: $T(n/2)$
 - ◆ Lines 6-7 use binary number additions/subtractions and bit-shift operations $\rightarrow O(n)$ time
- ◆ Thus, we get the recurrence
$$T(n) = 3 T(n/2) + O(n)$$
- ◆ Solving it, we get:
$$T(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

We will solve this recurrence later

Our Roadmap

- ◆ The idea of *Divide-and-Conquer*

- ◆ Examples for *Divide-and-Conquer*



- ◆ How do we solve recurrences?

Solving Recurrences

◆ Let's try to solve these recurrences

$$\text{◆ } T(n) = 2 T(n/2) + n \quad \rightarrow O(n \log_2 n)$$

$$\text{◆ } T(n) = 2 T(n/2) + \mathbf{O(n)} \quad \rightarrow O(n \log_2 n)$$

$$\text{◆ } T(n) = 3 T(n/2) + c n \quad \rightarrow O(n^{1.585})$$

$$\text{◆ } T(n) = 3 T(n/2) + c n^2 \quad \rightarrow O(n^2)$$

c is a constant

Solving Recurrences

◆ Substitution method

- ◆ Guess the function $T(n)$, substitute it into the recurrence, then check its correctness
- ◆ ☺ easy if you are lucky ☹ guess may not be tight

◆ Recursion tree method

- ◆ Draw a tree with the time used at each recursion level, then sum up the times at all levels
- ◆ ☺ no need to guess ☹ need to find the sum

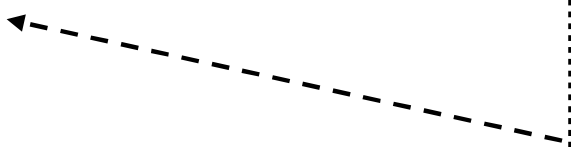
◆ Master method

- ◆ Obtain $T(n)$ by using a theorem directly
- ◆ ☺ no need to find the sum ☹ not applicable in some cases

Math. Equations

(to be used in the following methods)

- $\log_b c/d = \log_b c - \log_b d$
- $\log_b a = \log_x a / \log_x b$ (new base x : any positive number)
- $\log_b a^r = r \log_b a$
- $a^{\log_b n} = n^{\log_b a}$
- $\sum_{i=1..n} i = n(n+1)/2$
- $\sum_{i=0..n-1} a r^i = a(1-r^n)/(1-r)$
 - when $|r| > 1$, the series becomes larger and larger
 - when $|r| < 1$, the series converges to $a/(1-r)$



$$\begin{aligned} \text{let } F &= a^{\log_b n} \\ \log_b F &= \log_b a^{\log_b n} \\ \log_b F &= \log_b n \log_b a \\ F &= n^{\log_b a} \end{aligned}$$

Substitution

- ◆ Given recurrence: $T(n) = 2 T(n/2) + n$
 - ◆ Suppose that we guess $T(n) = O(n \log_2 n)$
 - ◆ 1. Rewrite it as $T(n) \leq c n \log_2 n$, for a constant c
 - ◆ 2. Substitute it into the right-side $T(n/2)$
$$\begin{aligned} 2 T(n/2) + n &= 2 c (n/2) \log_2 (n/2) + n \\ &= c n \log_2 n - c n \log_2 2 + n \\ &= c n \log_2 n - n (c - 1) \\ &\leq c n \log_2 n, \end{aligned}$$
by fixing $c > 1$
- Thus, we obtain $T(n) \leq c n \log_2 n$
- ◆ The guess is **correct**

Substitution

- ◆ Given recurrence: $T(n) = 2 T(n/2) + n$

- ◆ Suppose that we guess $T(n) = O(n)$

- ◆ 1. Rewrite it as $T(n) \leq c n$, for a constant c

- ◆ 2. Substitute it into the right-side $T(n/2)$

$$2 T(n/2) + n = 2 c (n/2) + n$$

$$= c n + n$$

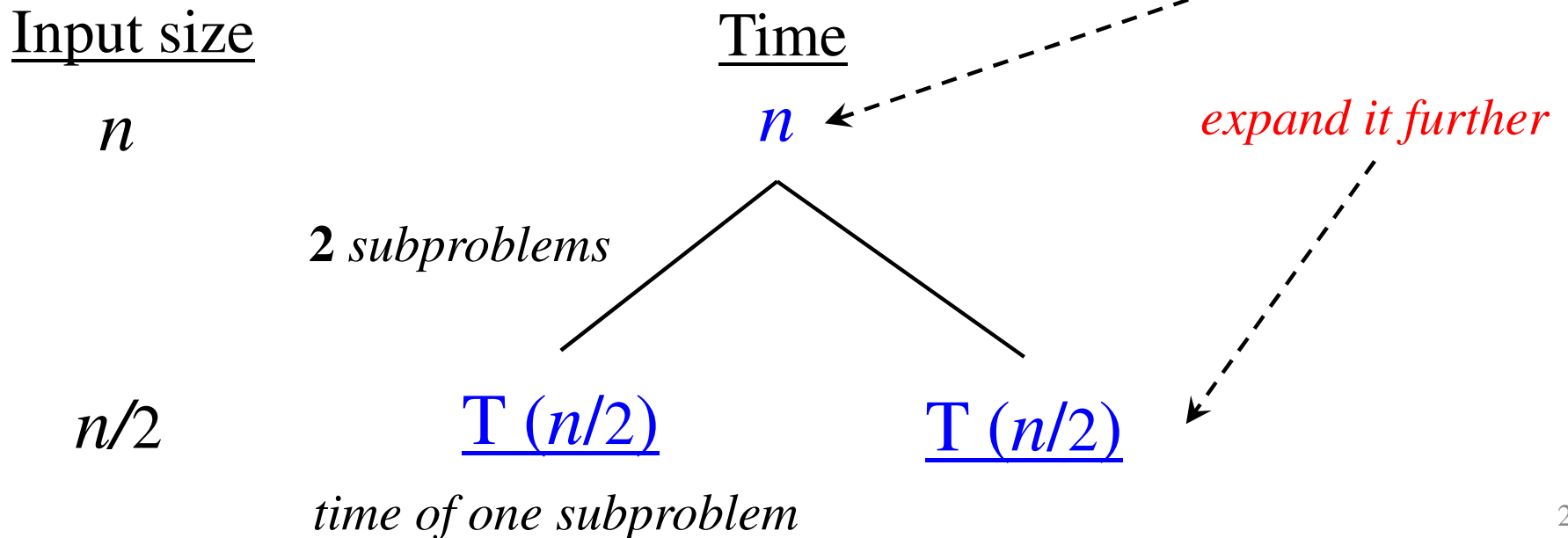
$$= (c + 1) n$$

Regardless of the value of c , we can't show: $(c + 1) n \leq c n$

- ◆ The guess is **wrong**! Substitution doesn't work here

Recursion Tree #1

- ◆ We have: $T(n) = 2 T(n/2) + n$
- ◆ Expand $T(n)$ in order to draw a recursion tree



Recursion Tree #1 for: $T(n) = 2 T(n/2) + n$

Input size

n

Time

n

$n/2$

$n/2$

$n/2$

$n/2^2$

$T(n/2^2)$

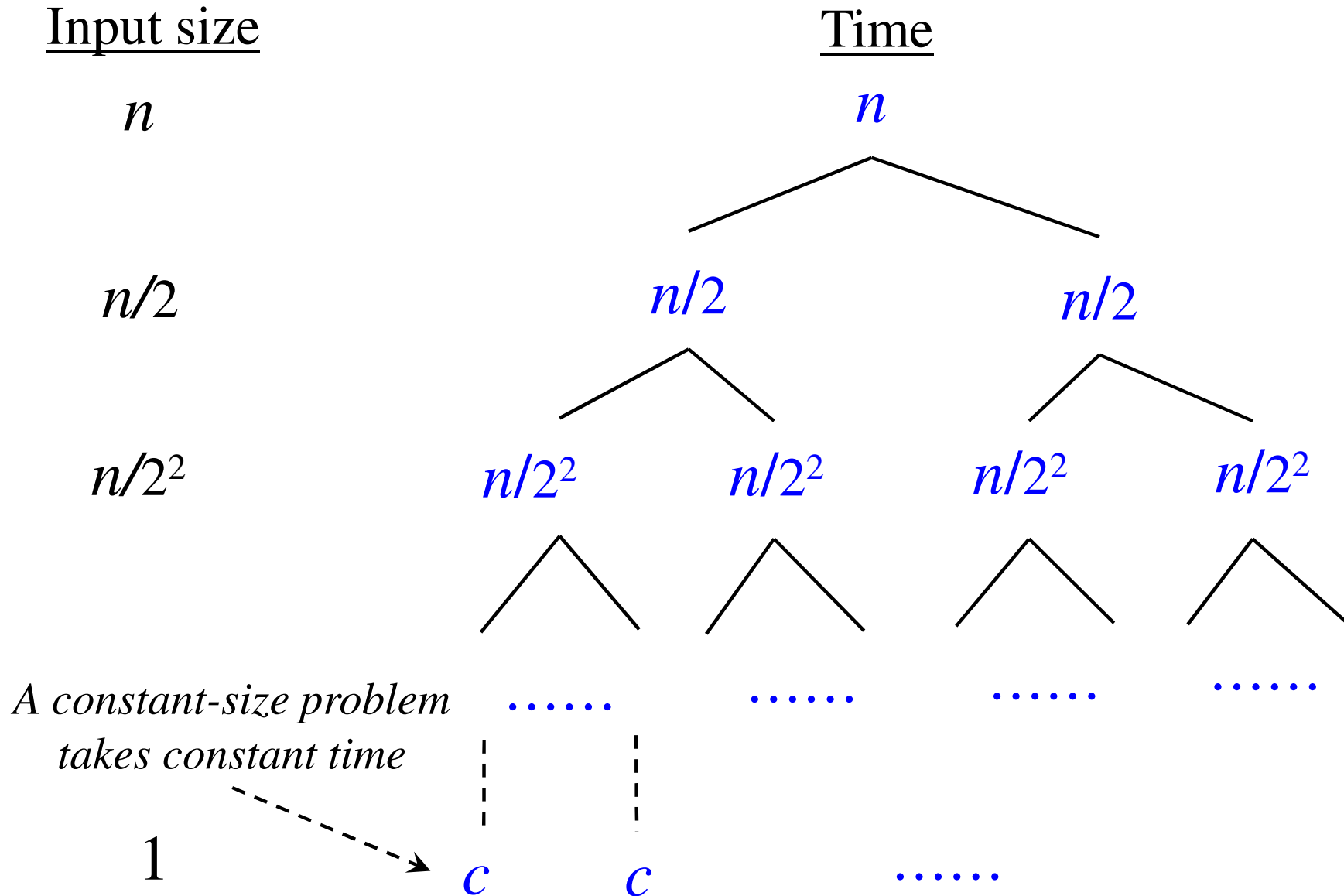
$T(n/2^2)$

$T(n/2^2)$

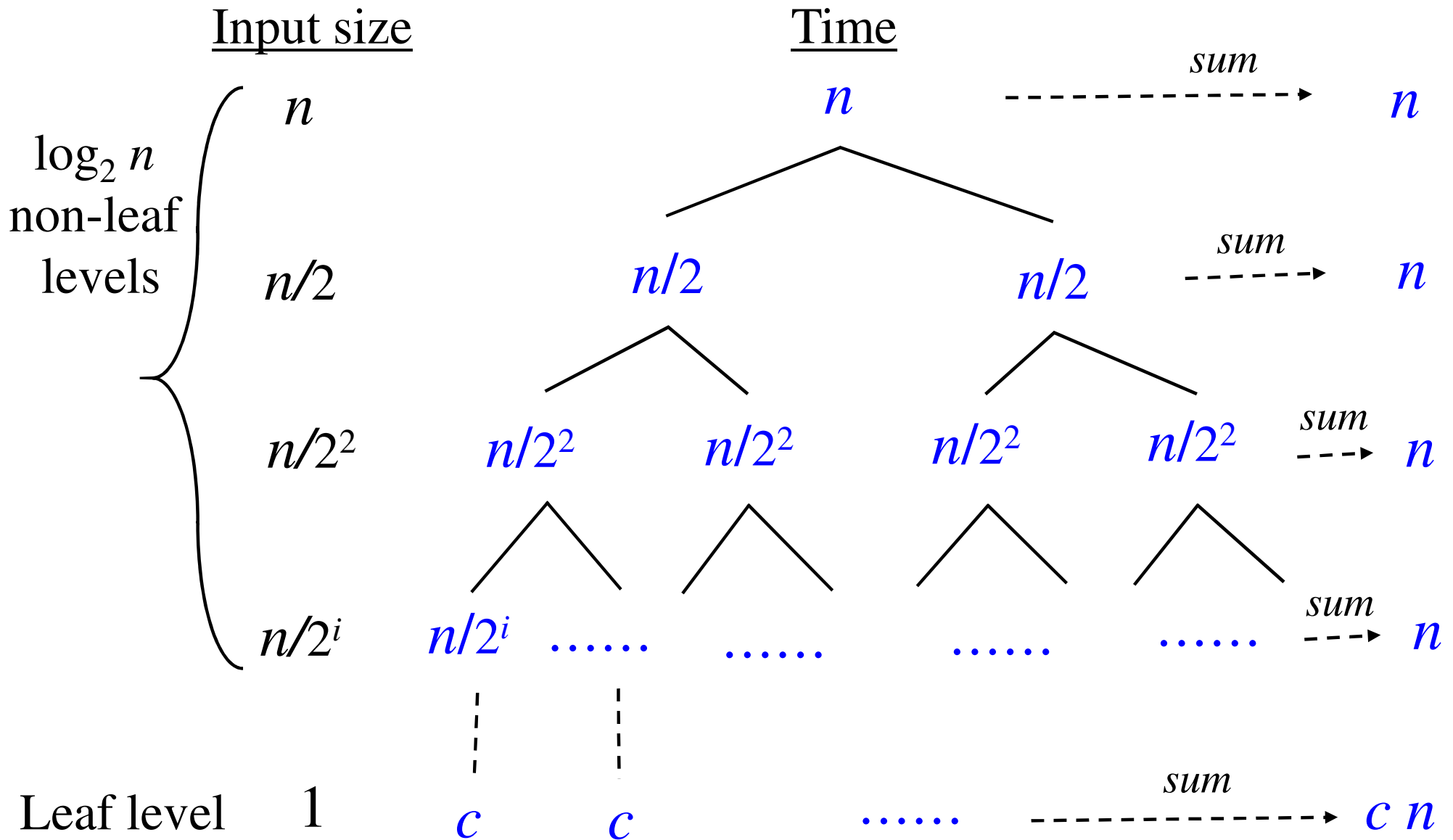
$T(n/2^2)$

expand it further

Recursion Tree #1 for: $T(n) = 2 T(n/2) + n$



Recursion Tree #1 for: $T(n) = 2 T(n/2) + n$

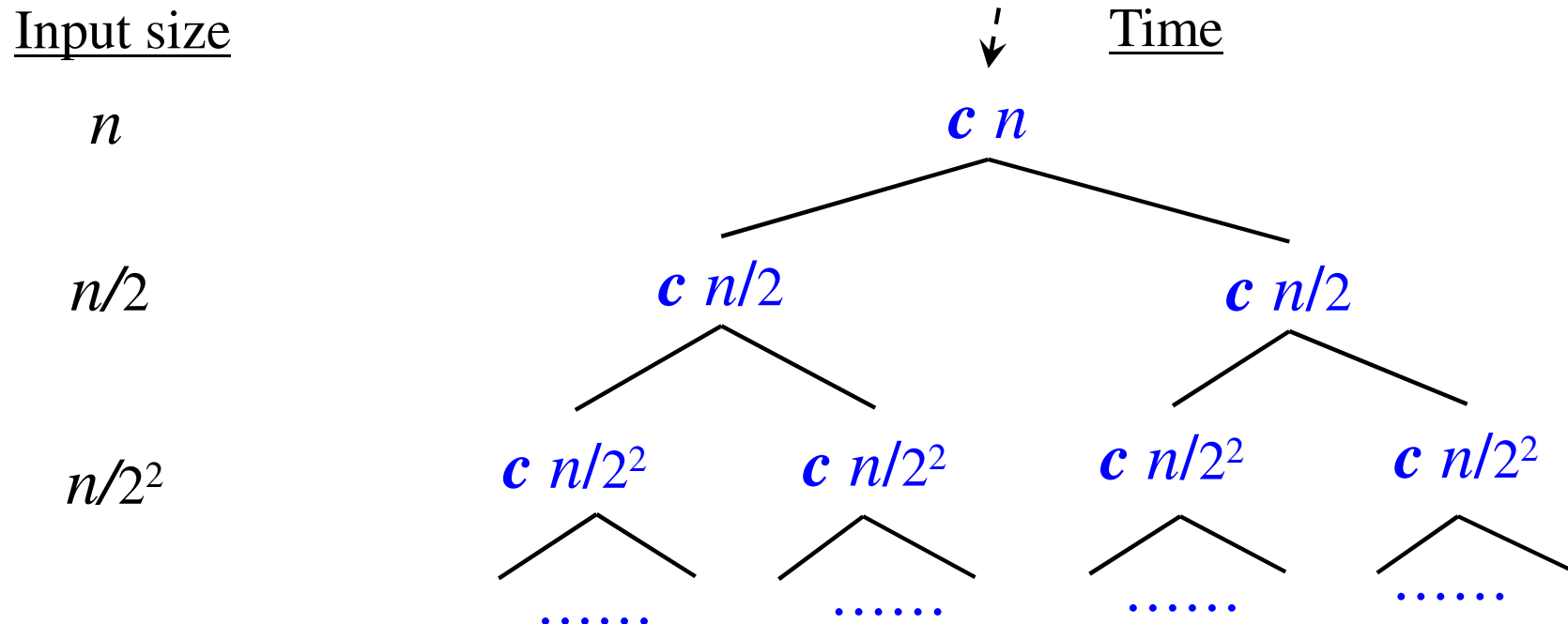


Recursion Tree #1 for: $T(n) = 2 T(n/2) + n$

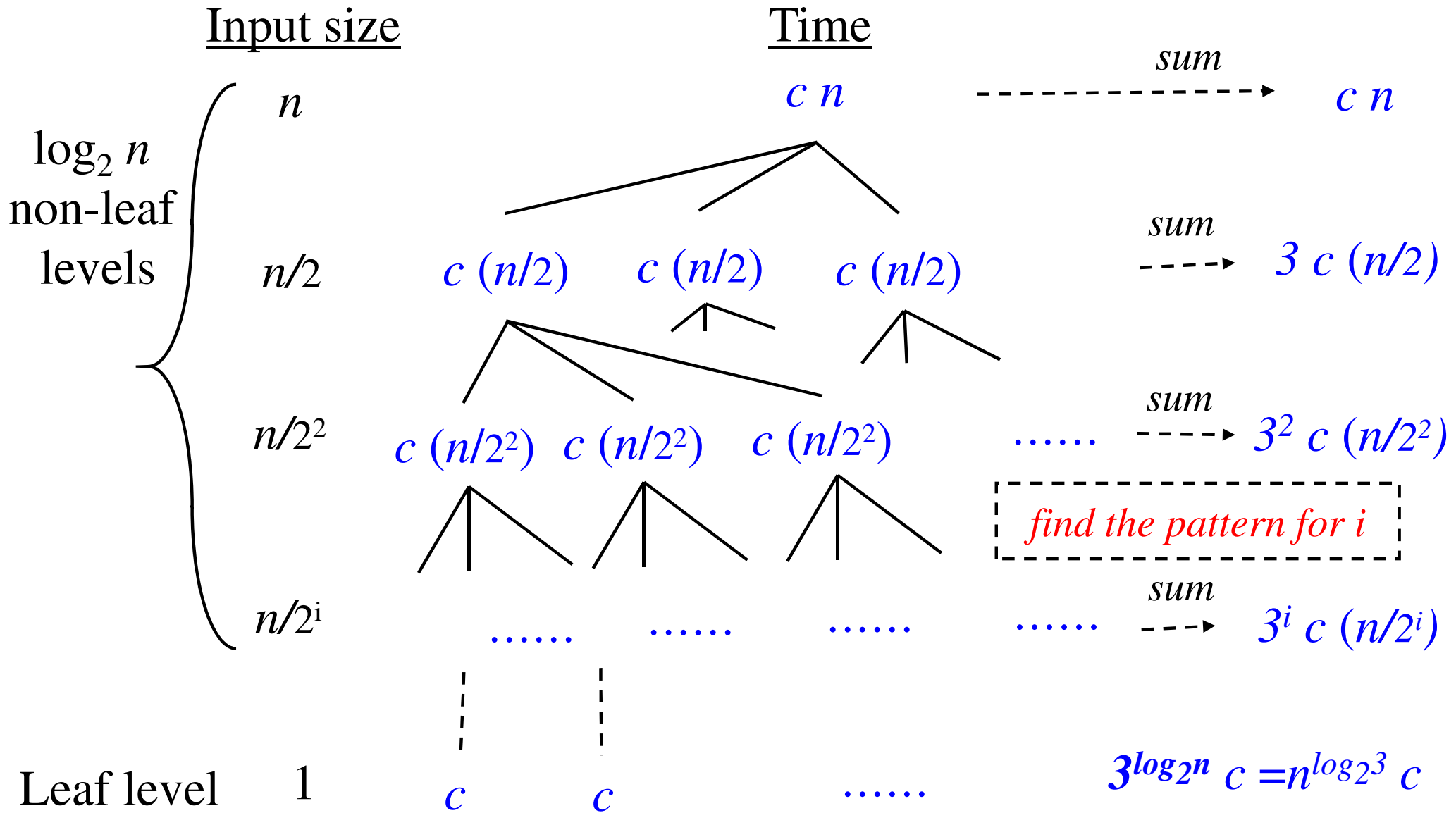
- ◆ Time at the **leaf** level: $c n$
- ◆ Number of **non-leaf** levels: $\log_2 n$
- ◆ Consider the level i
 - ◆ Input size of a subproblem: $n/2^i$
 - ◆ Number of subproblems: 2^i
 - ◆ Combine time of a subproblem: (size) = $n/2^i$
 - ◆ Total time at this level: $2^i (n/2^i) = n$
- ◆ Total time $T(n)$:
 - ◆ $n (\log_2 n) + c n = O (n \log_2 n)$

Recursion Tree #2

- Given the recurrence: $T(n) = 2 T(n/2) + \mathbf{O(n)}$
 - How is this recurrence different from the previous one?
- We avoid the term $O(n)$ in the recurrence
 - Rewrite it to: $T(n) = 2 T(n/2) + c n$, for some constant c
 - Then draw the recursion tree



Recursion Tree #3 for: $T(n) = 3 T(n/2) + c n$

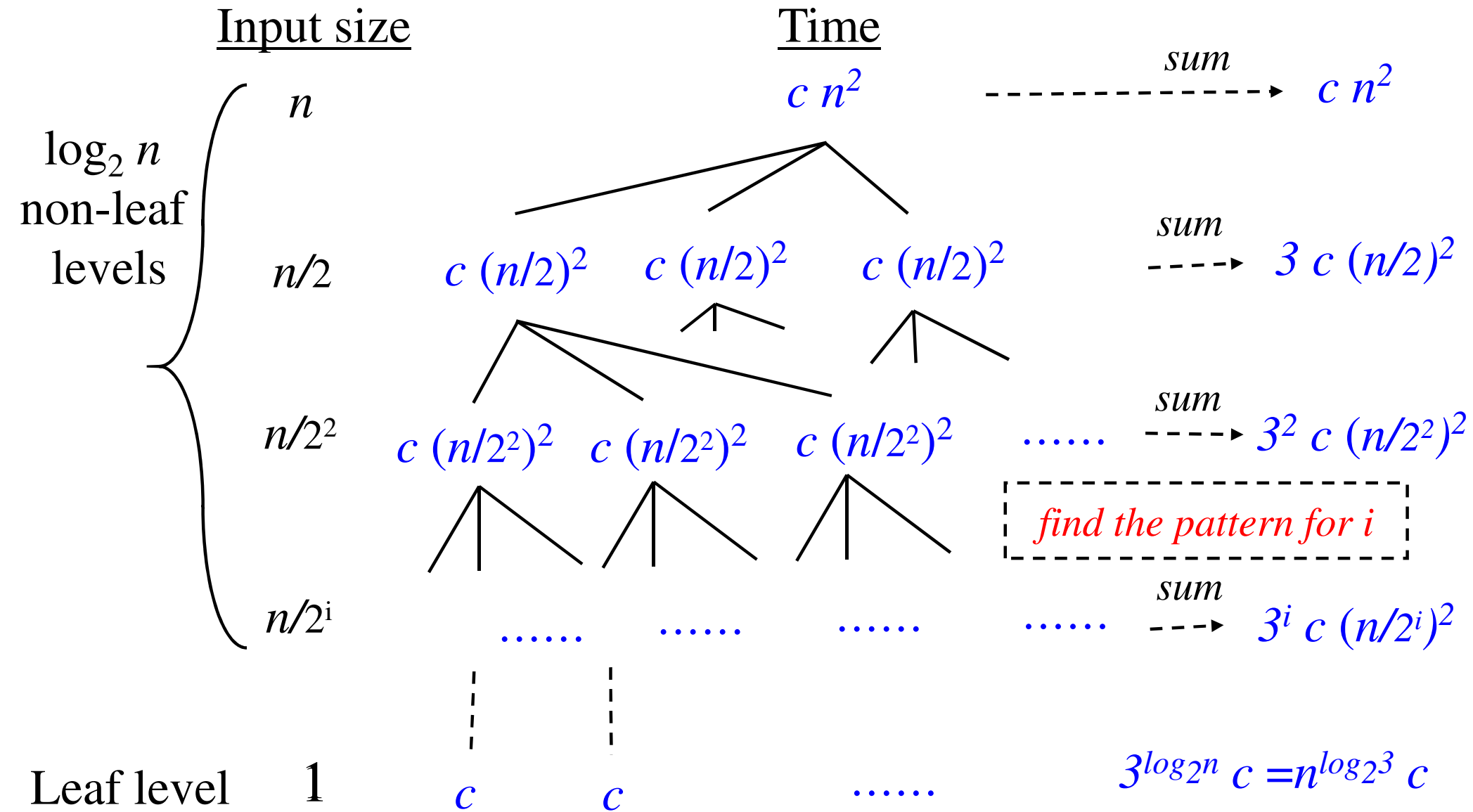


Recursion Tree #3 for: $T(n) = 3 T(n/2) + c n$

- Time at the **leaf** level: $c n^{\log_2 3}$
- Number of **non-leaf** levels $L: \log_2 n$
- Consider the level i (level 0 is the top level)
 - Input size of a subproblem: $n/2^i$
 - Number of subproblems: 3^i
 - Combine time of a subproblem: $c (n/2^i)$
 - Total time at this level: $3^i c (n/2^i)$
- Total time $T(n)$:

$$\begin{aligned} & c n^{\log_2 3} + \sum_{i=0..L-1} 3^i c (n/2^i) \\ = & c n^{\log_2 3} + c n (1.5^L - 1) / (1.5 - 1) \quad \boxed{\text{geometric sum equation}} \\ = & O(c n^{\log_2 3} + c n n^{\log_2 1.5}) \quad \boxed{\text{log base equation}} \\ = & O(n^{1.585}) \quad \boxed{\text{How do we get 1.585 ?}} \end{aligned}$$

Recursion Tree #4 for: $T(n) = 3 T(n/2) + c n^2$



Recursion Tree #4 for: $T(n) = 3 T(n/2) + c n^2$

- ◆ Time at the **leaf** level: $c n^{\log_2 3}$
- ◆ Number of **non-leaf** levels $L: \log_2 n$
- ◆ Consider the level i (level 0 is the top level)
 - ◆ Input size of a subproblem: $n/2^i$
 - ◆ Number of subproblems: 3^i
 - ◆ Combine time of a subproblem: $c (n/2^i)^2 = c n^2/4^i$
 - ◆ Total time at this level: $3^i c (n^2/4^i)$
- ◆ Total time $T(n)$:

$$\begin{aligned} & c n^{\log_2 3} + \sum_{i=0..L-1} 3^i c (n^2/4^i) \\ = & c n^{\log_2 3} + c n^2 (1 - 0.75^L) / (1 - 0.75) \\ \leq & c n^{\log_2 3} + c n^2 \\ = & O(n^2) \end{aligned}$$

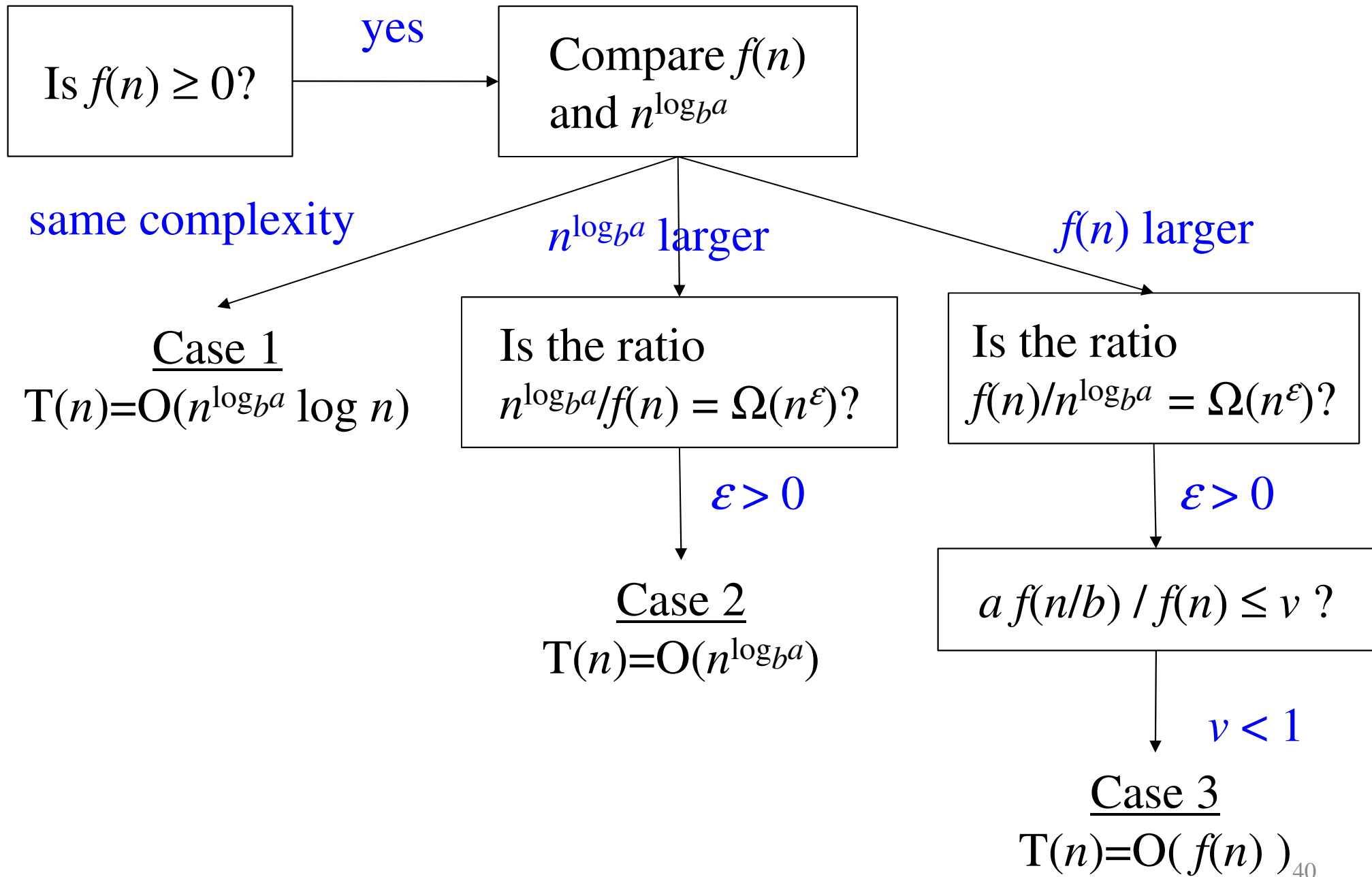
*With a larger “combine time”,
the total time is higher
than in the recursion tree #3*

Master Method

- ◆ Given the recurrence: $T(n) = a T(n/b) + f(n)$
 - $f(n)$ must be ≥ 0
- ◆ Compare $f(n)$ and $n^{\log_b a}$
 - ϵ, ν are some constants such that $\epsilon > 0, \nu < 1$

$$T(n) = \begin{cases} O(n^{\log_b a} \log n) & \text{if } f(n) = \Theta(n^{\log_b a}) \\ O(n^{\log_b a}) & \text{if } n^{\log_b a} / f(n) = \Omega(n^\epsilon) \quad \text{polynomially larger} \\ O(f(n)) & \text{if } f(n) / n^{\log_b a} = \Omega(n^\epsilon) \quad \text{polynomially larger} \\ & \text{and } a f(n/b) / f(n) \leq \nu \quad \text{regularity condition} \end{cases}$$

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



Master Method

◆ Given the recurrence $T(n) = 2 T(n/2) + c n$

◆ $a = 2, \quad b = 2, \quad n^{\log_b a} = n^{\log_2 2} = n$

◆ $f(n) = c n$

◆ Since $f(n) = \Theta(n^{\log_b a})$, we apply Case 1 and get:

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

◆ Given the recurrence $T(n) = 4 T(n/2) + c n$

◆ $a = 4, \quad b = 2, \quad n^{\log_b a} = n^{\log_2 4} = n^2$

◆ $f(n) = c n$

◆ Since $n^{\log_b a}/f(n) = \Omega(n^1)$, we apply Case 2 and get:

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

Master Method

- ◆ Given the recurrence $T(n) = 2 T(n/2) + c n \log n$
 - ◆ $a = 2, \quad b = 2, \quad n^{\log_b a} = n^{\log_2 2} = n$
 - ◆ $f(n) = c n \log n$
 - ◆ $f(n)$ is larger than $n^{\log_b a}$, maybe Case 3
 - ◆ $f(n) / n^{\log_b a} = c \log n$
 - ◆ Can you find some $\varepsilon > 0$ such that
$$c \log n = \Omega(n^\varepsilon) \quad ?$$
 - ◆ **NO.**
- ◆ The master method **can't be used** in this case

Master Method

Recurrence $T(n)$ $= aT(n/b) + f(n)$	$f(n)$	$n^{\log_b a}$	Which term is larger?	Ratio = $\Omega(n^\epsilon)$?	$a f(n/b) / f(n)$ $\leq \nu$?	$T(n)$
$2 T(n/2) + c n$	$c n$	$n^{\log_2 2} = n$	Same [case 1]			$n \log_2 n$
$4 T(n/2) + c n$	$c n$	$n^{\log_2 4} = n^2$	$n^{\log_b a}$ [case 2]	n		n^2
$2 T(n/2)$ $+ c n \log n$	$c n$ $\log n$	$n^{\log_2 2} = n$	$f(n)$ [case 3]	$\log n$		N.A.
$2 T(n/2) + c n^2$	$c n^2$	$n^{\log_2 2} = n$	$f(n)$ [case 3]	n	$2(n/2)^2/n^2$ $= 1/2$	$c n^2$

Summary

- ◆ Apply *Divide-and-Conquer* to solve a problem
 - ◆ Define a subproblem
 - ◆ Combine the solutions of subproblems
- ◆ Running time analysis
 - ◆ Find a recurrence for the algorithm
 - ◆ Solve the recurrence
- ◆ Please read Chapter 4 in the textbook
- ◆ *Next lecture*: sorting by using divide-and-conquer