分治策略 (续)
Divide and Conquer Cont'd

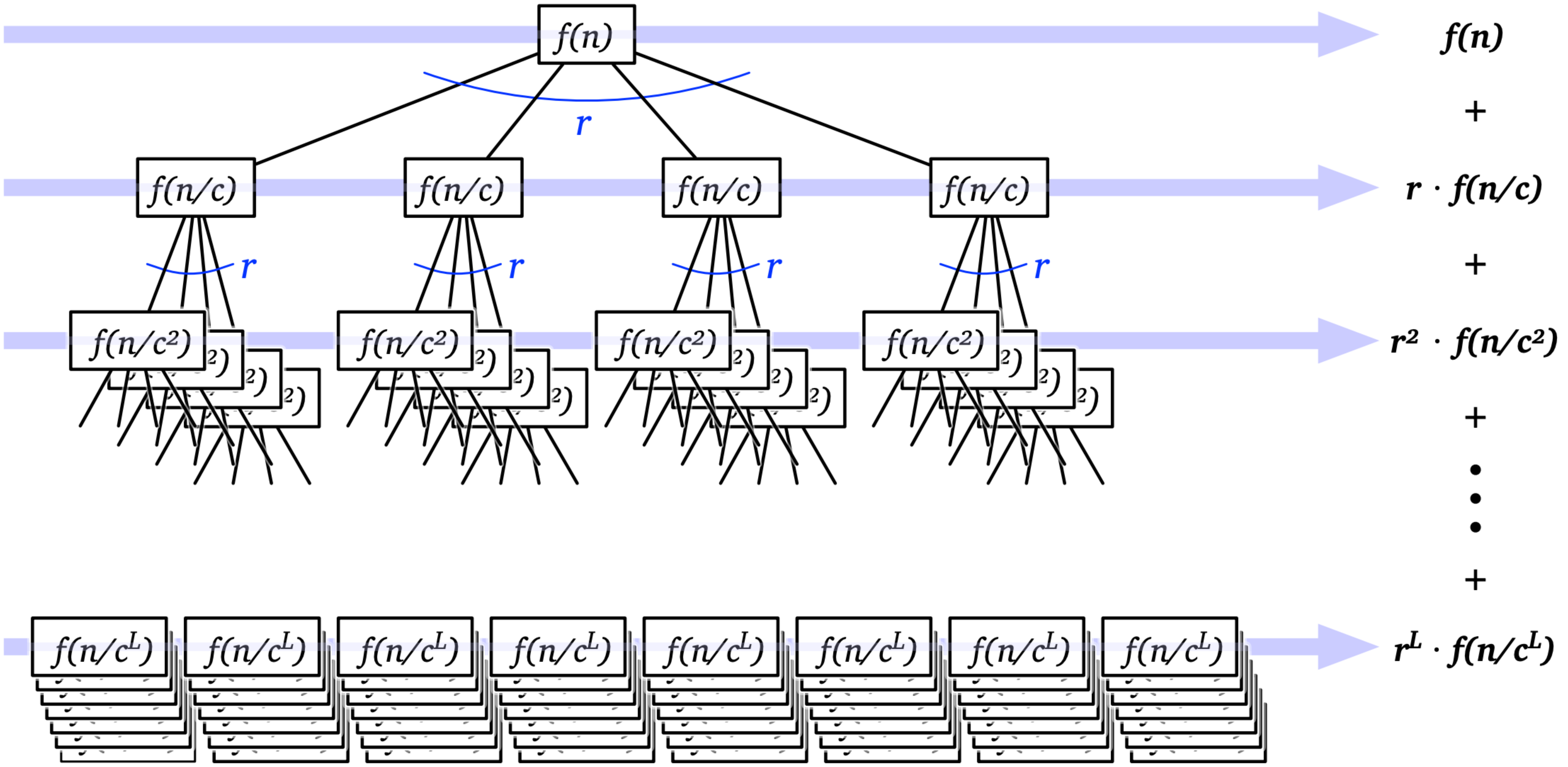
钮鑫涛
Nanjing University
2023 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports! We also use some materials from stanford-cs161.

智能软件与工程学院
School of Intelligent Software and Engineering

The recursion-tree method

- A great tool for solving divide-and-conquer recurrences.
  - Simple, pictorial, yet general.
- A **recursion tree** is a rooted tree with one node for each recursive subproblem.
- The **value of each node** is the time spent on that subproblem **excluding** recursive calls.
- The **sum of all values** is the runtime of the algorithm.

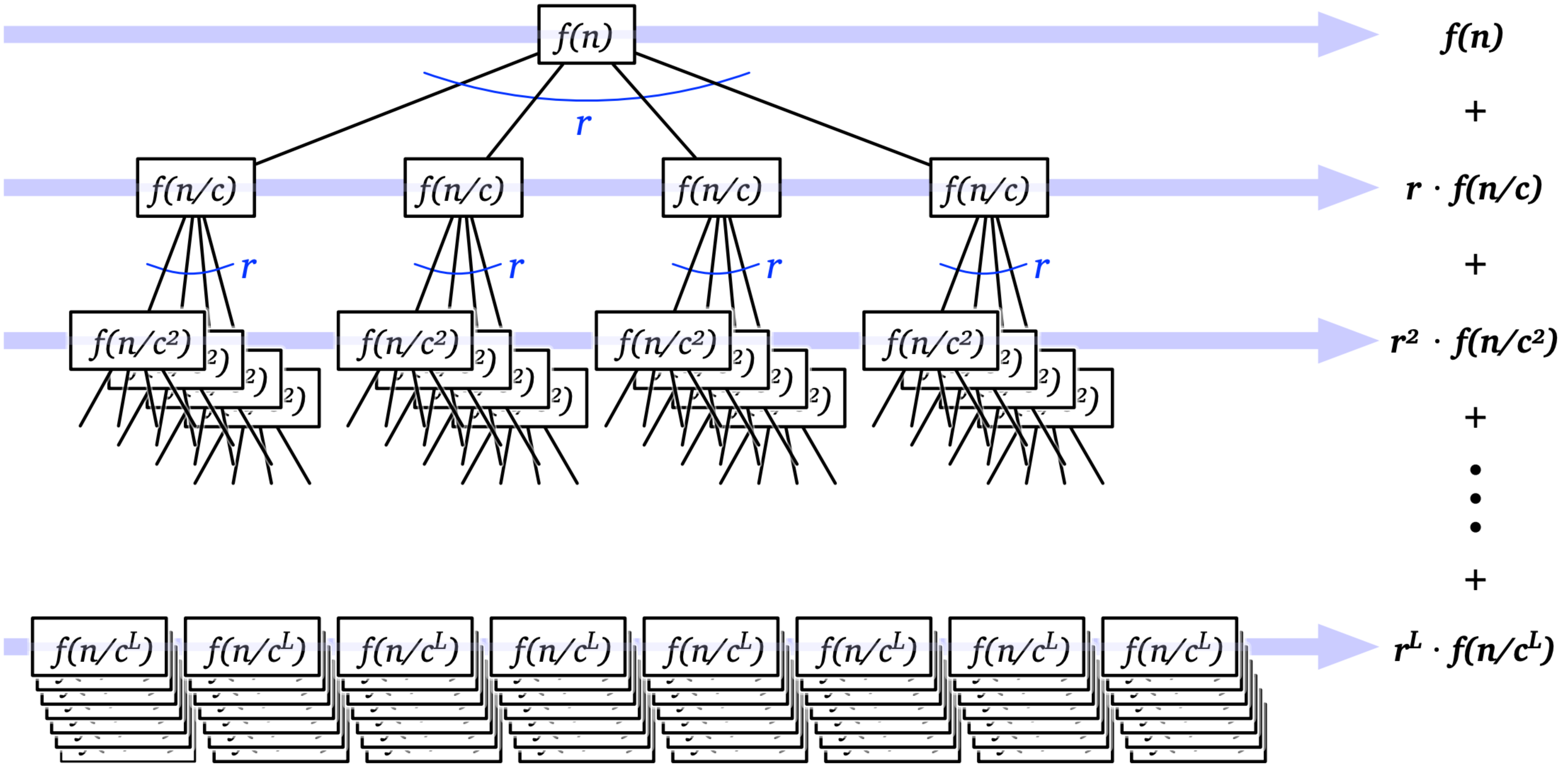


A recursion tree for the recurrence $T(n) = rT(n/c) + f(n)$

The recurrence-tree method

- Typical divide-and-conquer approach:
  - Divide a size n problem into r subproblems each of size n/c, the cost for “divide” and “combine” is $f(n)$
  - Solve problem directly if $n \leq n_0$ for some small constant n_0.
  - $T(n) = r \cdot T(n/c) + f(n), T(n_0) = c_0$
  - $T(n) = r \cdot T(n/c) + f(n), T(1) = f(1)$

we can choose whatever value of n_0 is most convenient for our analysis



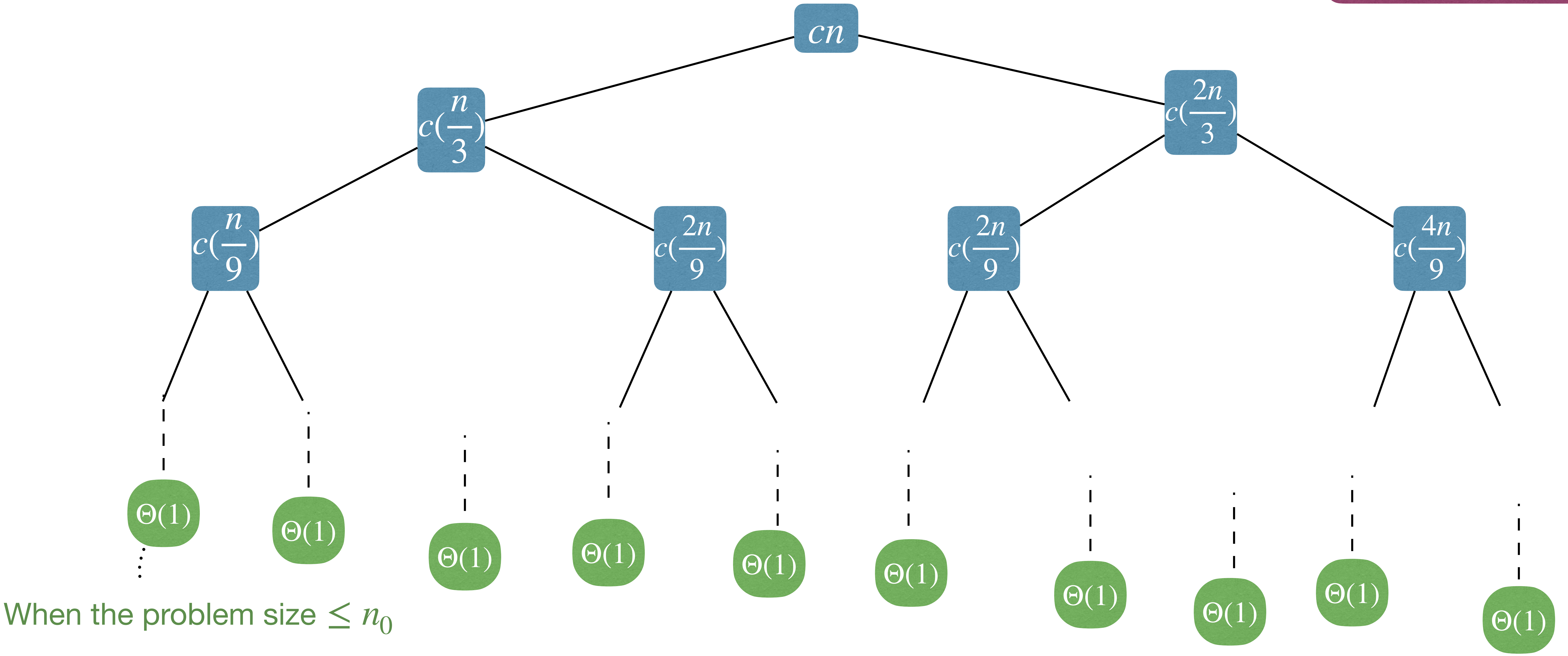
A recursion tree for the recurrence $T(n) = rT(n/c) + f(n)$

Total cost is $\sum_{i=0}^{L} r^i \cdot f(n/c^i)$, where $L = \log_c n$

Combine recursion tree and substitution

- What if subproblems are of different sizes?
  - E.g., $T(n) = T(n/3) + T(2n/3) + \Theta(n)$

Unbalanced, different root-to-leaf having different lengths



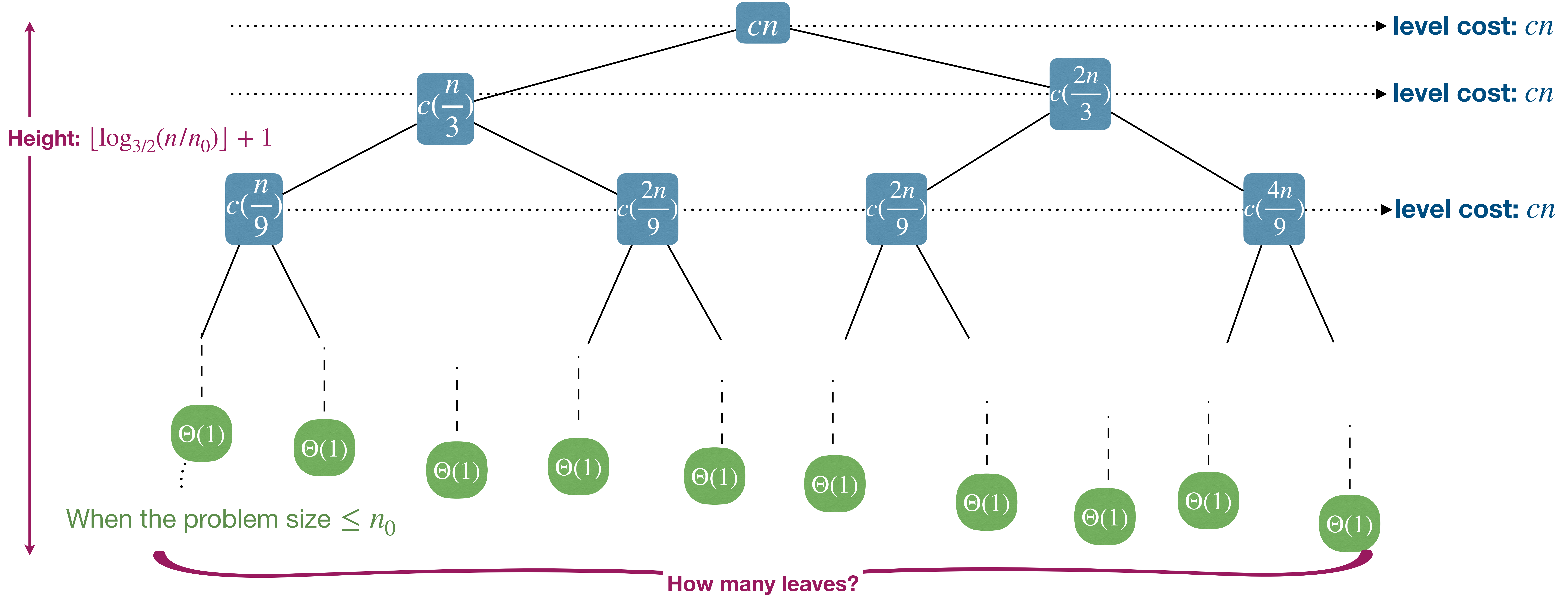
Combine recursion tree and substitution

- What if subproblems are of different sizes?
  - E.g., $T(n) = T(n/3) + T(2n/3) + \Theta(n)$

智能软件与工程学院
School of Intelligent Software and Engineering

Combine recursion tree and substitution

- What if subproblems are of different sizes?
  - E.g., $T(n) = T(n/3) + T(2n/3) + \Theta(n)$



Combine recursion tree and substitution

- Since the height (max lengths from root to leaf) $h = \lfloor \log_{3/2}(n/n_0) \rfloor + 1$, then the size of leaves will is smaller than $2^h = 2^{\lfloor \log_{3/2} n \rfloor + 1} \le 2n^{\log_{3/2} 2}$?
  - Leads to the cost of leaves to be $O(n^{\log_{3/2} 2}) \sim O(n^{1.71})$
  - $O(n \lg n) + O(n^{1.71}) = O(n^{1.71}) \ \rightarrow$ not tight!
- Can we get a more accurate number of leaves?
  - Guess, e.g., the leaves are also $O(n \lg n)$?

Combine recursion tree and substitution

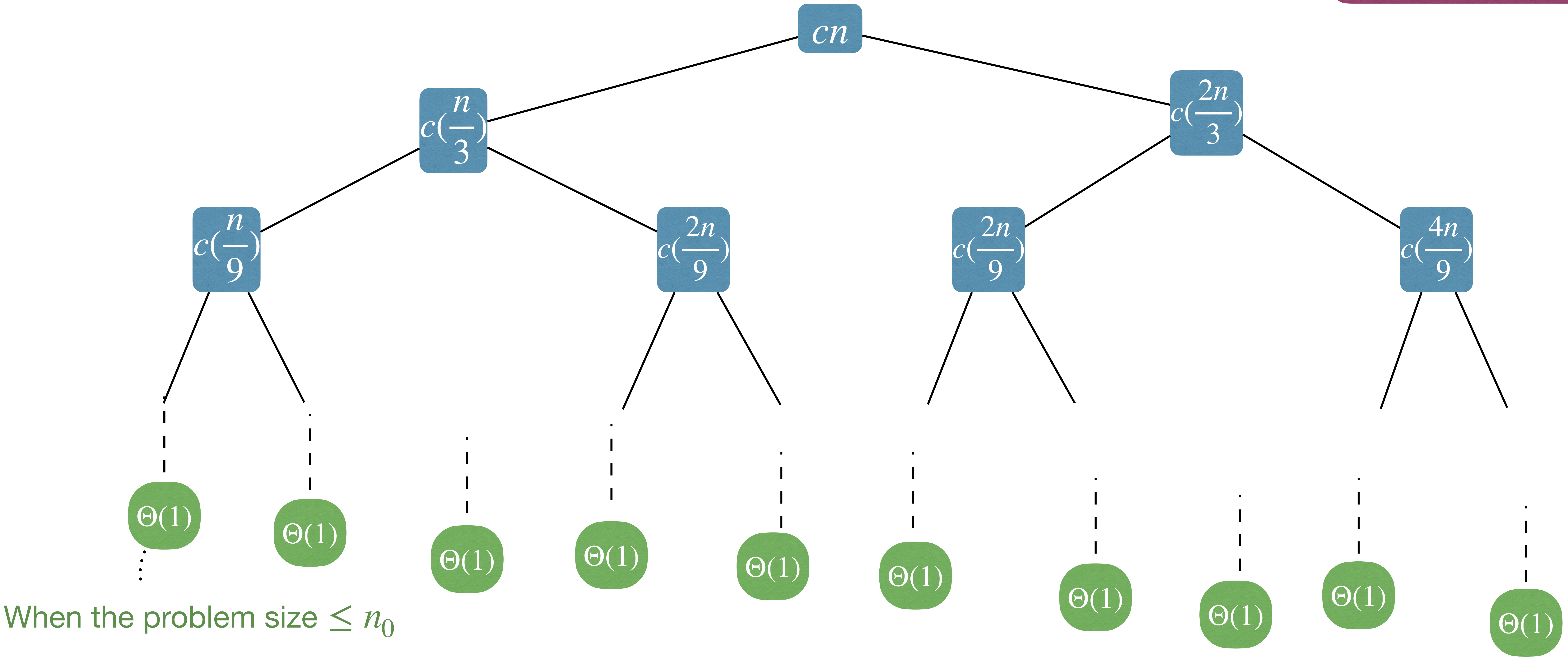
- Guess and verify
  - Let $L(n)$ be the number of leaves in the recursion tree for $T(n)$
  - $$L(n) = \begin{cases} 1 & if\ n < n_0 \\ L(n/3) + L(2n/3) & if\ n \geq n_0 \end{cases}$$
  - Inductive hypothesis: $L(n) \leq d \cdot n \lg(n+1)$, for all values less than n
  - Base case: $L(1) \leq d \cdot \lg 2$, which is very easy to be satisfied by choosing proper d
  - Inductive step:
    $$L(n) = L(n/3) + L(2n/3) \leq d \cdot n/3 \lg(n/3+1) + d \cdot 2n/3 \lg(2n/3+1)$$
    $$< d \cdot n \lg(2n/3+1) < d \cdot n \lg(n+1)$$

The cost of all leaves is: $O(n \lg n)$

Combine recursion tree and substitution

- What if subproblems are of different sizes?
  - E.g., $T(n) = T(n/3) + T(2n/3) + \Theta(n)$

Complexity: $O(n \lg n)$



Master Method

Simple version of Master Theorem

Theorem (**Master theorem, 主定理**) Let $a \geq 1$ and $b > 1$, and d be constants (independent of n), and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = a \cdot T(n/b) + \Theta(n^d)$$

Interpret n/b to either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$, and the theorem is still true

Then $T(n)$ has the following asymptotic bounds:

$$T(n) = \begin{cases} \Theta(n^d \lg n) & \textit{if } a = b^d \\ \Theta(n^d) & \textit{if } a < b^d \\ \Theta(n^{\log_b a}) & \textit{if } a > b^d \end{cases}$$

a: number of subproblems

b : factor by which input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions

Applications of master theorem

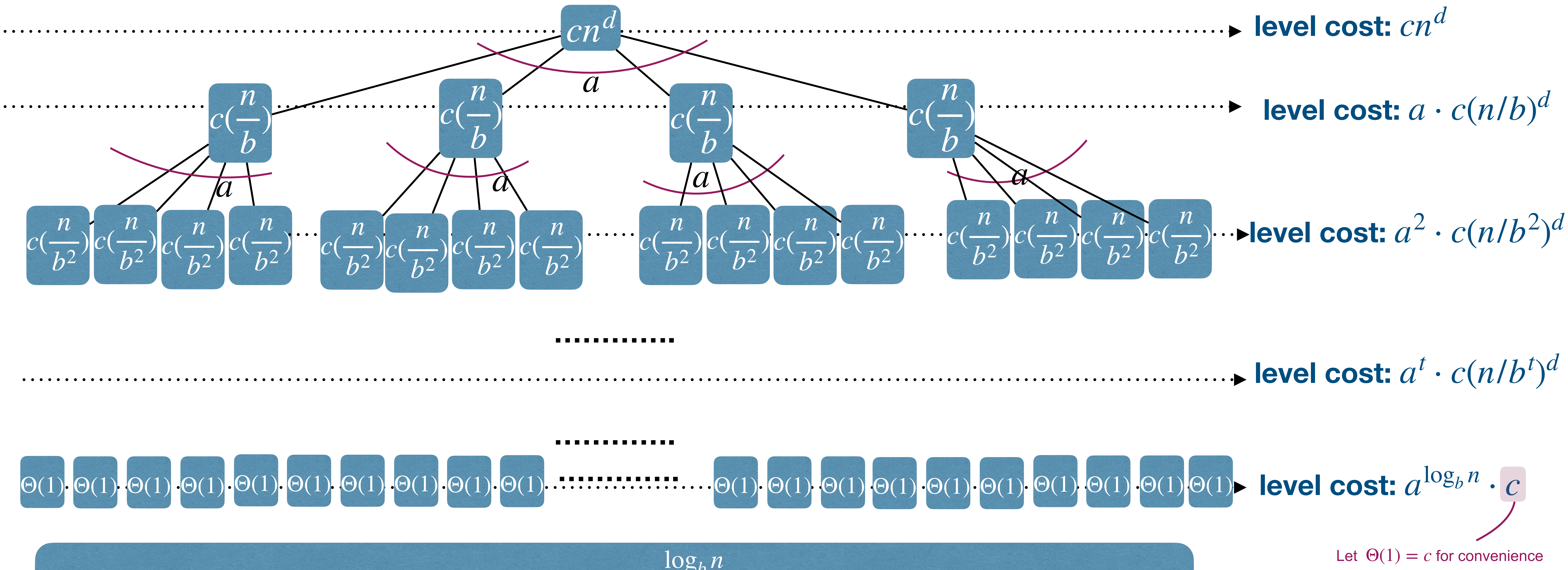
$$T(n) = a \cdot T(n/b) + \Theta(n^d)$$

$$T(n) = \begin{cases} \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- Karatsuba integer multiplication
  - $T(n) = 3T(n/2) + \Theta(n) \ \rightarrow \ a = 3, b = 2, d = 1$, leading to $a > b^d$
  - $T(n) = \Theta(n^{\log_2 3}) \sim \Theta(n^{1.6})$
- MergeSort
  - $T(n) = 2T(n/2) + \Theta(n) \ \rightarrow \ a = 2, b = 2, d = 1$, leading to $a = b^d$
  - $T(n) = \Theta(n \lg n)$
- Consider the following:
  - $T(n) = T(n/2) + \Theta(n) \ \rightarrow \ a = 1, b = 2, d = 1$, leading to $a < b^d$
  - $T(n) = \Theta(n)$

Proof of the master theorem

- Suppose $T(n) = a \cdot T(n/b) + c \cdot n^d$



Total cost is $c \cdot n^d \cdot \sum_{i=0}^{\log_b n} (a/b^d)^t$

Now let's check all the cases

$$T(n) = \begin{cases} \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- Case 1: $a = b^d$
  - $$T(n) = cn^d \sum_{t=0}^{\log_b n} (a/b^d)^t$$

$$= cn^d \sum_{t=0}^{\log_b n} 1$$

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

$$= cn^d(\lg n / \lg b + 1)$$

$$= \Theta(n^d \lg n)$$

Now let's check all the cases

$$T(n) = \begin{cases} \Theta(n^d \lg n) & \textit{if } a = b^d \\ \Theta(n^d) & \textit{if } a < b^d \\ \Theta(n^{\log_b a}) & \textit{if } a > b^d \end{cases}$$

- Case 2: $a < b^d$
  - $T(n) = cn^d \sum_{t=0}^{\log_b n} (a/b^d)^t$

Geometric sums

- $\sum_{t=0}^{N} x^t = \dfrac{x^{N+1}-1}{x-1}$ for $x \neq 1$
  - If $0 < x < 1$, $1 \leq \dfrac{x^{N+1}-1}{x-1} \leq \dfrac{1}{1-x}$, with N growing and x being constant, it is $\Theta(1)$
  - If x = 1, all terms are the same
  - If x > 1, $x^N \leq \dfrac{x^{N+1}-1}{x-1} \leq x^N \cdot (\dfrac{x}{x-1})$, with N growing and x being constant, it is $\Theta(x^N)$

Now let's check all the cases

$$T(n) = \begin{cases} \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- Case 2: $a < b^d$
  - $$T(n) = cn^d \sum_{t=0}^{\log_b n} (a/b^d)^t$$

$$= cn^d \cdot [\text{some constant}]$$

$$= \Theta(n^d)$$

Now let's check all the cases

$$T(n) = \begin{cases} \Theta(n^d \lg n) & if\ a = b^d \\ \Theta(n^d) & if\ a < b^d \\ \Theta(n^{\log_b a}) & if\ a > b^d \end{cases}$$

- Case 3: $a > b^d$
  - $$T(n) = cn^d \sum_{t=0}^{\log_b n} (a/b^d)^t$$

$$= \Theta(n^d (a/b^d)^{\log_b n})$$

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

Understanding the master theorem

$$T(n) = a \cdot T(n/b) + \Theta(n^d)$$

$$T(n) = \begin{cases} \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

a: number of subproblems

b : factor by which input size shrinks

d : need to do n^d work to create all the subproblems and combine their solutions

- Branching causes the number of problems to explode! The most work is at the bottom of the tree!

- The problems lower in the tree are smaller! The most work is at the top of the tree!

General Master Theorem

Theorem (**Master theorem, 主定理**) Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

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 $a \cdot f(n/b) \leq c \cdot f(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

General Master Theorem

Theorem (**Master theorem, 主定理**) Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

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 $a \cdot f(n/b) \leq c \cdot f(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

- The Master Theorem does not cover all cases!
- For example to $f(n) = O(n^{\log_b a - \epsilon})$
  - If $a = b = 2$ and $f(n) = n/\lg n$, case one does not apply
- When master theorem does not apply, we need to use substitution approach and recursion tree method.

Ignoring Floors and Ceilings is Okay

- When consider the **recurrence (递归式)** of MergeSort, i.e., $T(n) = 2 \cdot T(n/2) + \Theta(n)$
  - What if the given n is odd? What it is mean sort an array of size $\frac{13}{2}$?
- Actually, the actual recurrence of MergeSort is $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$
- How can we get the real time complexity of this recurrence?

Domain transformation

- We can transform the recurrence into a more familiar form, by defining a new function in terms of the one we want to solve, e.g:
  - First, let's overestimate the time bound, we have the following relation to eliminate the ceiling:
    $T(n) \leq 2 \cdot T(\lceil n/2 \rceil) + \Theta(n) \leq 2 \cdot T(n/2 + 1) + \Theta(n)$
  - Define a new function $S(n) = T(n + \alpha)$, choosing the constant α so that $S(n)$ satisfies the simpler recurrence $S(n) \leq 2S(n/2) + \Theta(n)$
    - $S(n) = T(n + \alpha) \leq 2 \cdot T(n/2 + \alpha/2 + 1) + \Theta(n + \alpha)$
      $= 2 \cdot S(n/2 + \alpha/2 + 1 - \alpha) + \Theta(n + \alpha)$
    - Setting $\alpha = 2$ simplifies this recurrence, i.e., $S(n) \leq S(n/2) + \Theta(n)$

Domain transformation

- $T(n) \le 2 \cdot T(\lceil n/2 \rceil) + \Theta(n) \le 2 \cdot T(n/2 + 1) + \Theta(n)$
- $S(n) = T(n + 2)$
- $S(n) \le S(n/2) + \Theta(n) \quad \rightarrow \quad S(n) = O(n \log n)$
- We have $T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n)$
- A similar argument implies the matching lower bound $T(n) = \Omega(n \log n)$
- Therefore, $T(n) = \Theta(n \log n)$

Domain transformation

- Similar domain transformations can be used to remove floors, ceilings, and even lower order terms from any divide and conquer recurrence

- But now that we realize this, we don't need to bother grinding through the details ever again!

A simple quiz

- Consider the recurrence:
  - $T(n) = 2 \cdot T(n/2 + 17) + \Theta(n)$
- Can you get its time complexity?

Summary until now

- **Divide**, **Conquer** (recursively or directly), and **Combine**.
- Same problem can be divided in different ways, leading to different algorithms with different performances!
  - MergeSort uses half-and-half split, how about 1-and-(n - 1) split?
- Correctness of divide-and-conquer algorithms:
  - Use mathematical induction
- Time complexity of divide-and-conquer algorithms:
  - Recursion-tree method, substitution method (Guess and Verify), Master method

Reduce-and-Conquer

Reduce and Conquer

- We might not need to consider all subproblems.
  - In fact, sometimes only need to consider one subproblem.
- The "combine" step will also be easier, or simply trivial…
- It is also called decrease and conquer

The Search Problem

- Input: an array A containing n elements, and an element x.
- Output: index of x if it's in A, otherwise return “no”.

9	3	7	1	5	6	1

returns “4”

9	3	7	1	5	6	8

returns “no”

- Simple solution: sequential scan.
- Worst-case runtime is $\Theta(n)$, but inevitable…
- What if the input array is sorted?

2	4	4	5	6	7	8	9	11	17	23	28	17

Binary Search

2	4	4	5	6	7	8	9	11	17	23	28

17

- Find the middle element: **7**.
- Compare 17 to the middle element: **7 < 17**.
- Reduce the array to one of the two splits: **the right half**.
- Recurse!

Binary Search

2	4	4	5	6	7	8	9	11	17	23	28	17

- Find the middle element: **11**.
- Compare 17 to the middle element: **11 < 17**.
- Reduce the array to one of the two splits: **the right half**.
- Recurse!

Binary Search

2	4	4	5	6	7	8	9	11	17	23	28	17

- Find the middle element: **23**.
- Compare 17 to the middle element: **23 > 17**.
- Reduce the array to one of the two splits: **the left half**.
- Recurse!

Binary Search

2	4	4	5	6	7	8	9	11	17	23	28	17

- Find the middle element: **23**.
- Compare 17 to the middle element: **17 = 17**.
- We have found the element, and we are done!

Binary Search

2	4	4	5	6	7	8	9	11	17	23	28

17

Reduce-and-Conquer:

- Start with a problem of size n.
- Compare the middle element to the specified element.
- Either we are done or have reduced the problem to size $n/2$.
  - Only consider one of the two subproblems. (**REDUCE!**)
- Repeat.

Binary Search

<u>BinarySearch(A, x):</u>

$left := 1, right := n$

while $true$

  $middle :=$ $(left+right)/2$ **floor or ceil?**

  if $A[middle] = x$

    return $middle$

  else if $A[middle] < x$

    $left := middle + 1$

  else

    $right := middle - 1$

- Does it solve the search problem, given the input array is sorted?
  - No! (E.g., when x is not in A.)
- Does it solve the search problem, given the input is sorted and $x \in A$?
  - Yes?

- At the beginning of each iteration, $A[left] \leq x \leq A[right]$. (Use induction.)
- At the beginning of some iteration, it must be $left = right$.
- In that iteration, it must be $A[left] = A[right] = x$, and we are done!

Binary Search

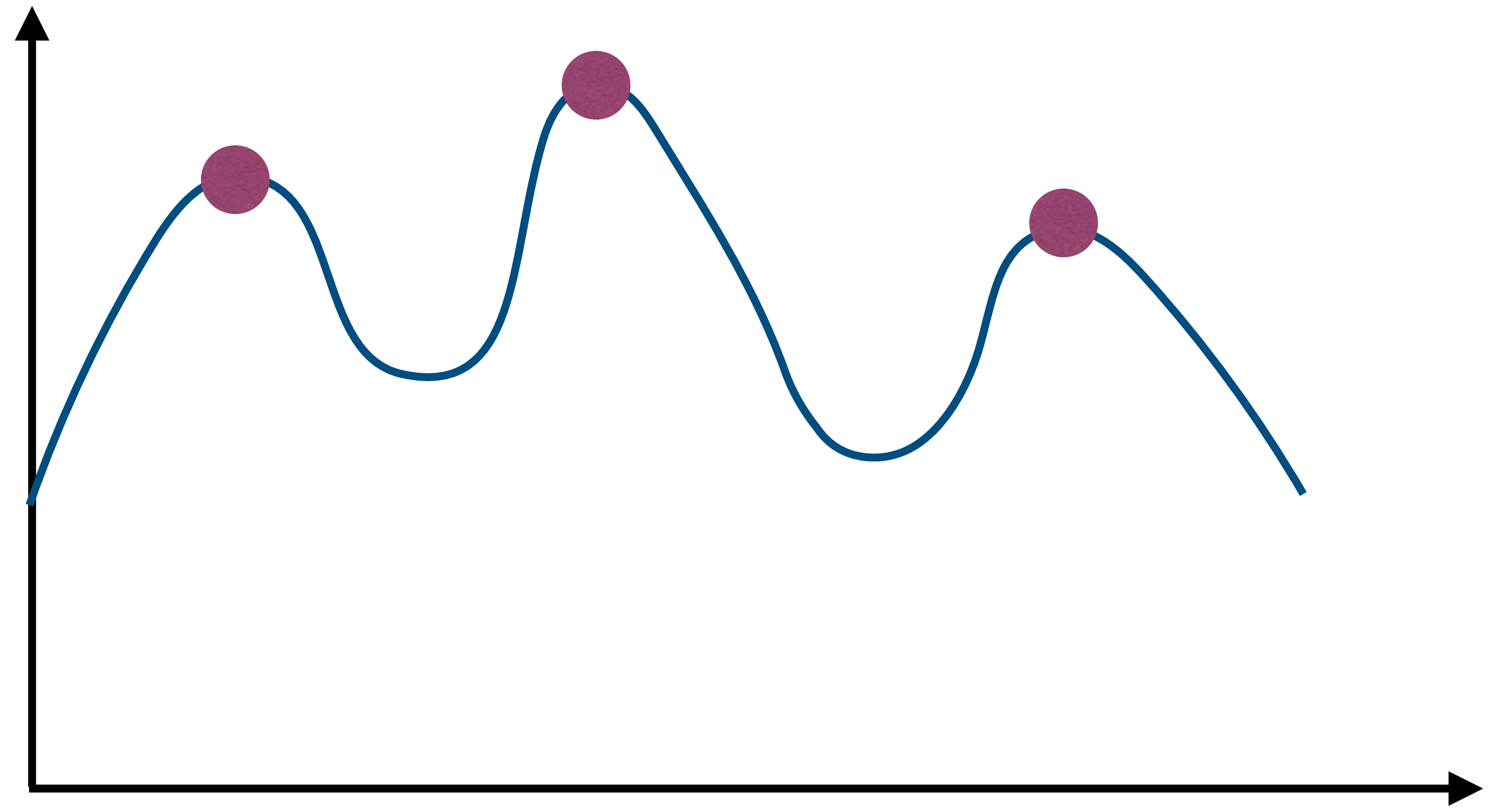
```
BinarySearch(A, x):
left := 1, right := n
while left <= right
    middle := (left+right)/2
    if A[middle] = x
        return middle
    else if A[middle] < x
        left := middle + 1
    else
        right := middle - 1
```

- Why this algorithm works?
  - if $x \in A$ previous argument still holds.
  - if $x \notin A$, then
    - After each iteration, we reduce input size by at least half.
    - At some iteration, $left = right$.
    - After that iteration, $left > right$.

Time complexity of `BinarySearch`? $T(n) \leq T(n/2) + \Theta(1)$ ⇒ $T(n) = O(\log n)$

Peak finding

- **Input**: an array A of n elements.
- **Output**: a ***local*** maximum; i.e., a ***peak***.
  - An element $A[i]$ is a peak if it is no smaller than its adjacent elements.
  - Every non-empty A has at least one peak. (Why?)



Local vs Global Maximum

- Global max is better, but finding it takes more time…
  - Sequential scan needs $O(n)$ time, and it's inevitable.
- Sometimes a peak is "good enough".
- Finding a peak costs much less time!

Peak finding

2	4	9	2	5	6	23	4	6	8	17	5

- Find middle element: 6.
- Compare middle element to its adjacent elements:
  - Middle element ≥ its left neighbor? Yes.
  - Middle element ≥ its right neighbor? No!
- Reduce the array to one of the two splits: the right half. (There must exist a peak in the part containing the large neighbor! [WHY?])
- Recurse!

Peak finding

2	4	9	2	5	6	23	4	6	8	17	5

- Find middle element: 6.
- Compare middle element to its adjacent elements:
  - Middle element ≥ its left neighbor? Yes.
  - Middle element ≥ its right neighbor? No!
- Reduce the array to one of the two splits: the right half. (There must exist a peak in the part containing the large neighbor! [WHY?])
- Recurse!

Peak finding

2	4	9	2	5	6	23	4	6	8	17	5

- Find middle element: **17**.
- Compare middle element to its adjacent elements:
  - Middle element ≥ its left neighbor? Yes.
  - Middle element ≥ its right neighbor? Yes, We are done!

Peak finding

PeakFinding(A):

$left := 1, right := n$

while $left <= right$

$middle := (left+right)/2$

if $middle > left$ **and** $A[middle - 1] > A[middle]$

$right := middle - 1$

else if $middle < right$ **and** $A[middle + 1] > A[middle]$

$left := middle + 1$

else

return $A[middle]$

Current array is not empty.

Get the middle element.

If middle < left neighbor (if there is such neighbor), recurse into left part.

If middle < right neighbor (if there is such neighbor), recurse into right part.

We find a peak!

Why this algorithm is correct?

1. It always terminates. (WHY?)

Peak finding

PeakFinding(A):

$left := 1, right := n$

while $left <= right$

$middle := (left+right)/2$

if $middle > left$ **and** $A[middle - 1] > A[middle]$

$right := middle - 1$

else if $middle < right$ **and** $A[middle + 1] > A[middle]$

$left := middle + 1$

else

return $A[middle]$

Current array is not empty.

Get the middle element.

If middle < left neighbor (if there is such neighbor), recurse into left part.

If middle < right neighbor (if there is such neighbor), recurse into right part.

We find a peak!

Why this algorithm is correct?

2. It always returns a right answer. (WHY?)

Peak finding

PeakFinding(A):

$left := 1, right := n$

while $left <= right$

$middle := (left+right)/2$

if $middle > left$ **and** $A[middle - 1] > A[middle]$

$right := middle - 1$

else if $middle < right$ **and** $A[middle + 1] > A[middle]$

$left := middle + 1$

else

return $A[middle]$

Current array is not empty.

Get the middle element.

If middle < left neighbor (if there is such neighbor), recurse into left part.

If middle < right neighbor (if there is such neighbor), recurse into right part.

We find a peak!

Runtime of this algorithm?

$O(\log n)$ Finding local max is faster!

Peak finding, now in 2D!

- **Input**: a 2D array A of $n \times n = n^2$ elements.
- **Output**: a *local* maximum; i.e., a *peak*.
    - An element $A[i][j]$ is a peak if it's no smaller than its four adjacent elements.

10	8	5	2	1
3	2	1	5	7
17	5	9	2	5
7	9	4	6	8
6	1	4	6	8

- Every non-empty A has at least one peak. (WHY?)
    - Proof: Start from a node, follow an "increasing path", eventually must reach a peak.

Peak finding in 2D, Algorithm I

- “Compress” each column into one element, resulting an 1D array.
  - Use max of each column to represent that column.
  - Run previous algorithm on the 1D array and return a peak.

10	8	5	2	1
3	2	1	5	7
17	5	9	2	5
7	9	4	6	8
6	1	4	6	8

$\Downarrow$

17	9	9	6	8

Peak finding in 2D, Algorithm I

- "Compress" each column into one element, resulting an 1D array.
  - Use max of each column to represent that column.
  - Run previous algorithm on the 1D array and return a peak.

Correctness?

		a		
	c	m	d	
	m_i	b	m_j	

$$m \geq a, m \geq b$$
$$m \geq m_i \geq c, m \geq m_j \geq d$$

Complexity

$$O(n^2) + O(\log n) = O(n^2)$$

too slow...

Peak finding in 2D, Algorithm II

- Scan the middle column and find the max element m.
- If m is a peak then return it, and we are done.
- Otherwise, left or right neighbor of m is bigger than m.
- Recurse into that part.

A divide (reduce) and conquer algorithm!

10	2	8	5	1	5	1
3	3	2	1	7	7	2
5	6	7	2	5	3	5
7	3	11	9	8	6	7
6	5	21	4	8	4	2
2	4	1	4	3	5	3
1	2	3	5	8	3	9

10	2	8	5	1	5	1
3	3	2	1	7	7	2
5	6	7	2	5	3	5
7	3	11	9	8	6	7
6	5	21	4	8	4	2
2	4	1	4	3	5	3
1	2	3	5	8	3	9

10	2	8	5	1	5	1
3	3	2	1	7	7	2
5	6	7	2	5	3	5
7	3	11	9	8	6	7
6	5	21	4	8	4	2
2	4	1	4	3	5	3
1	2	3	5	8	3	9

10	2	8	5	1	5	1
3	3	2	1	7	7	2
5	6	7	2	5	3	5
7	3	11	9	8	6	7
6	5	21	4	8	4	2
2	4	1	4	3	5	3
1	2	3	5	8	3	9

10	2	8	5	1	5	1
3	3	2	1	7	7	2
5	6	7	2	5	3	5
7	3	11	9	8	6	7
6	5	21	4	8	4	2
2	4	1	4	3	5	3
1	2	3	5	8	3	9

Peak finding in 2D, Algorithm II

- Scan the middle column and find the max element m.
- If m is a peak then return it, and we are done.
- Otherwise, left or right neighbor of m is bigger than m, then recurse into that part.

Correctness?

- Max of middle column is a peak; or a peak exists in the part containing the large neighbor, and that peak is the max of its column.
- A peak (found by the algorithm) in the part containing the large neighbor is also a peak in the original matrix.
- The algorithm eventually returns a peak of some (sub)matrix.

Peak finding in 2D, Algorithm II

- Scan the middle column and find the max element m.
- If m is a peak then return it, and we are done.
- Otherwise, left or right neighbor of m is bigger than m, then recurse into that part.

Runtime of this algorithm?

- $T(n) \le T(n/2) + \Theta(n)$ implying $T(n) = O(n)$

Not correct! 2D!

- $T(n, n') \le T(n/2, n') + O(n')$
- $T(n, n') \le (\lg n) \cdot O(n') = O(n' \lg n) = O(n \lg n)$

Much faster than the $O(n^2)$ algorithm, but can we do better?

Peak finding in 2D

- When considering the “reducing”, the smaller the size of the subproblem is, the better the performance is the algorithm
- Algorithm II reduce the problem into halve size, can it be smaller?

Peak finding in 2D, Algorithm III

- Scan the “**cross**” and find max element m.
- If m is a peak then return it, and we are done.
- Otherwise, some neighbor of m is bigger than m. Recurse into that quadrant.

10	8	5	2	1
3	2	1	5	7
5	11	9	2	5
7	21	4	6	8
6	1	4	6	8

10	8	5	2	1
3	2	1	5	7
5	11	9	2	5
7	21	4	6	8
6	1	4	6	8

10	8	5	2	1
3	2	1	5	7
5	11	9	2	5
7	21	4	6	8
6	1	4	6	8

10	8	5	2	1
3	2	1	5	7
5	11	9	2	5
7	21	4	6	8
6	1	4	6	8

Peak finding in 2D, Algorithm III

- Scan the "**cross**" and find max element m.
- If m is a peak then return it, and we are done.
- Otherwise, some neighbor of m is bigger than m. Recurse into that quadrant.

Correctness?

- Max in the cross is a peak; or a peak exists in the quadrant containing the large neighbor, and that peak is the max of some cross.
- A peak (found by the algorithm) in the quadrant containing the large neighbor is also a peak in the original matrix.
- The algorithm eventually returns a peak of some (sub)matrix.

Peak finding in 2D, Algorithm III

- Scan the "**cross**" and find max element m.
- If m is a peak then return it, and we are done.
- Otherwise, some neighbor of m is bigger than m. Recurse into that quadrant.

False Claim: A peak (found by the algorithm) in the quadrant containing the large neighbor is also a peak in the original matrix.

Peak finding in 2D, Algorithm III

- Scan the "**cross**" and find max element m.
- If m is a peak then return it, and we are done.
- Otherwise, some neighbor of m is bigger than m. Recurse into that quadrant.

False Claim: A peak (found by the algorithm) in the quadrant containing the large neighbor is also a peak in the original matrix.

			4			
			4			
			4			
4	4	4	4	4	4	5
			4		1	6
			4	1	1	1
			4	3	2	

Not a peak!

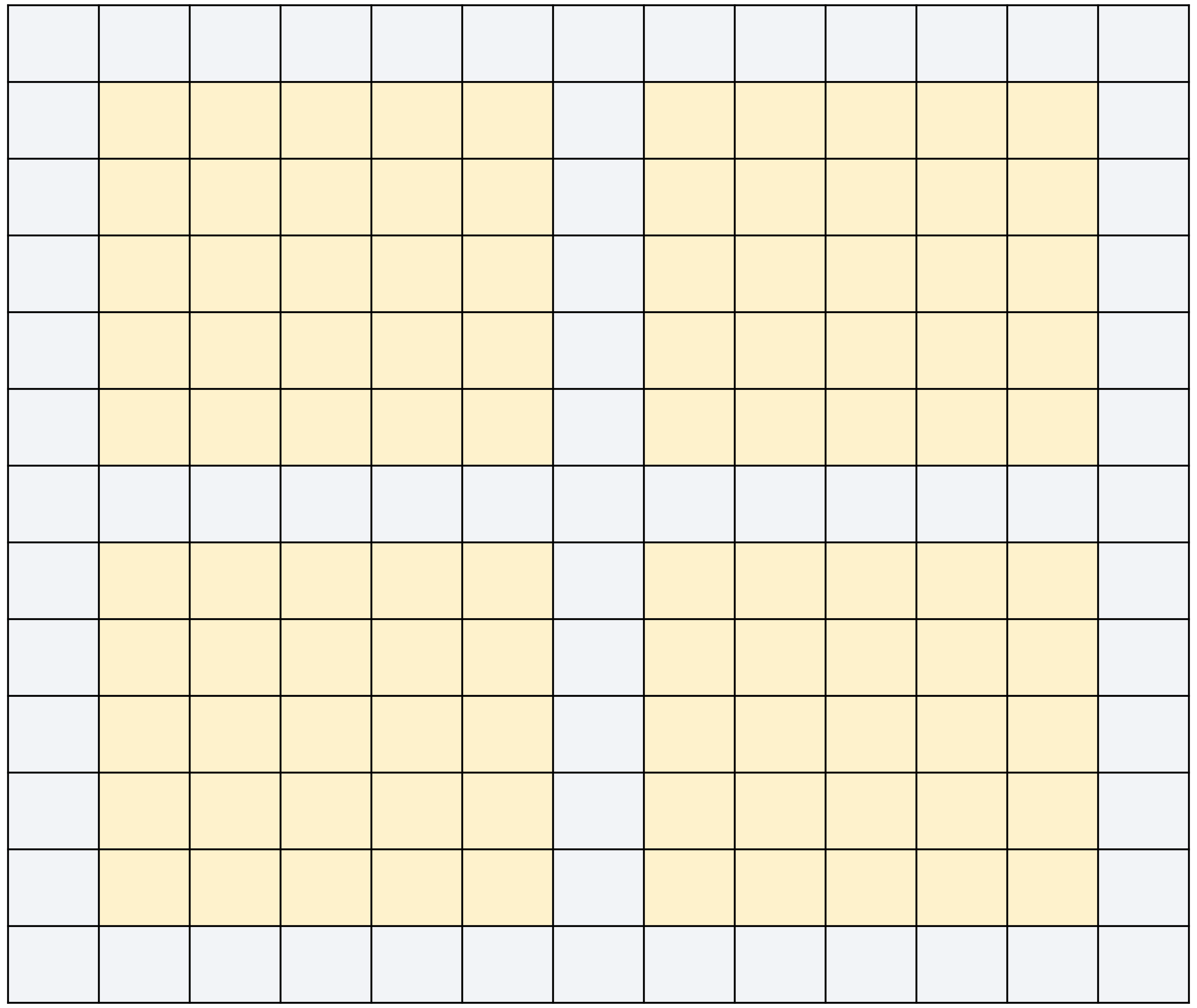
If the peak found in the quadrant is on the boundary of the quadrant, then it may be smaller than its neighbor that is in the original matrix!

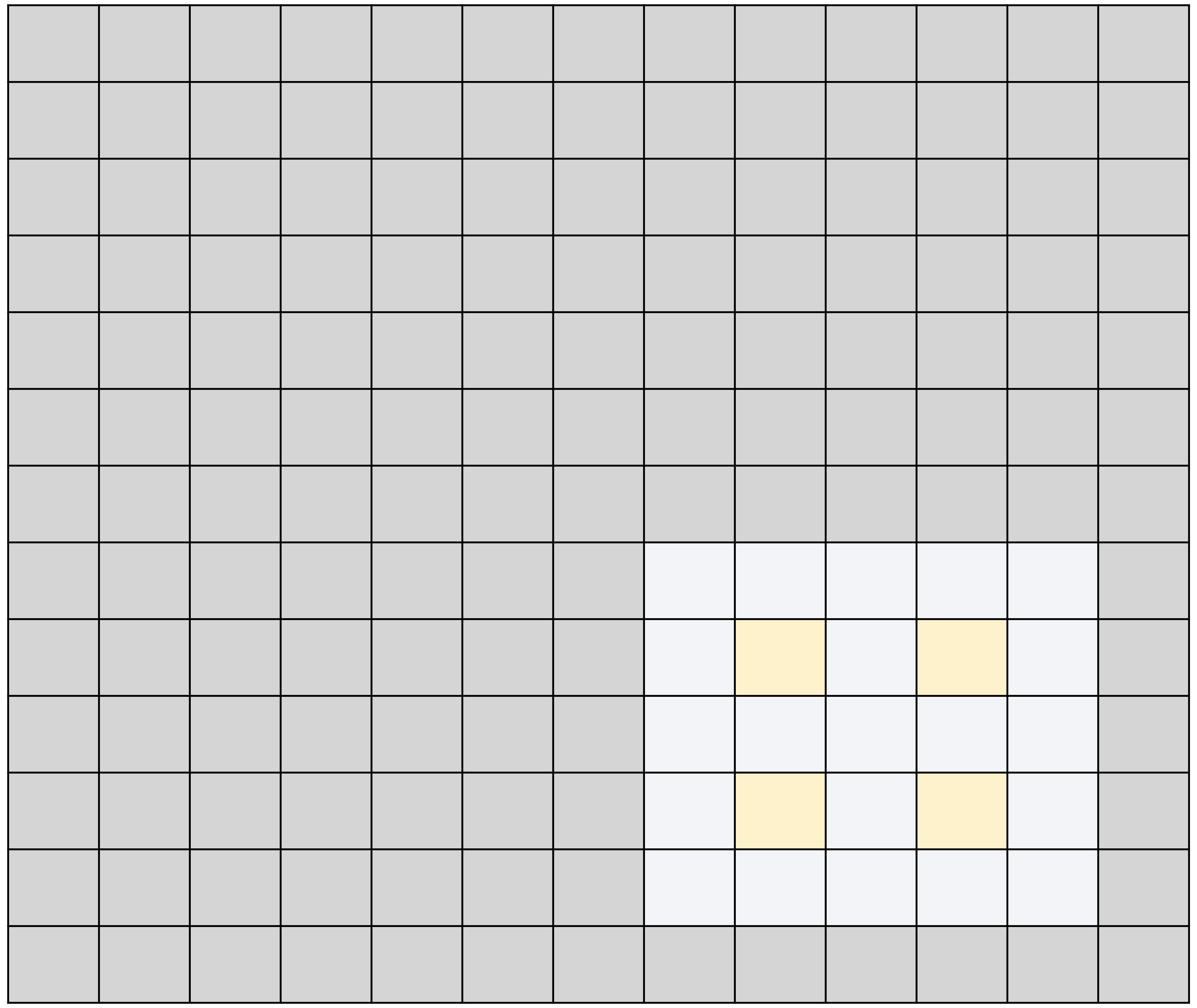
How to fix it?

Peak finding in 2D, Algorithm III

"cross and boundary" (i.e., a "window frame")

- Scan the "cross" and find max element m.
- If m is a peak then return it, and we are done.
- Otherwise, some neighbor of m is bigger than m. Recurse into that quadrant.

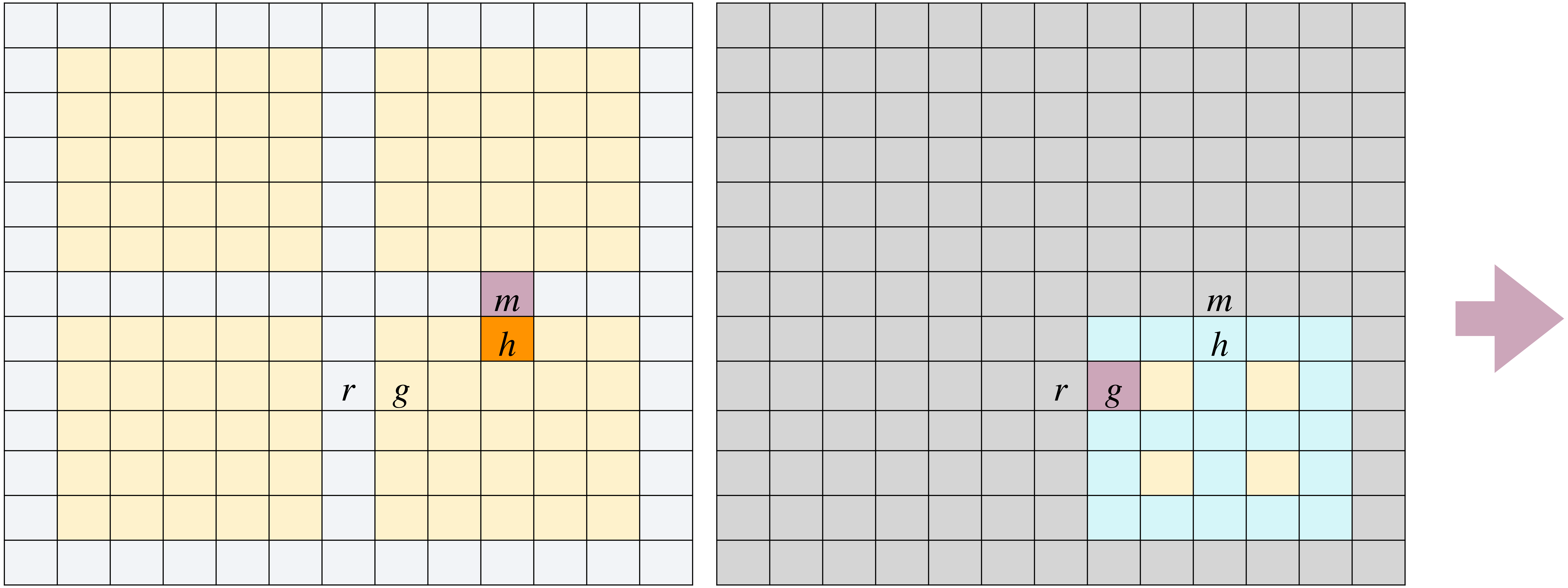




Peak finding in 2D, Algorithm III

"cross and boundary" (i.e., a "window frame")

- Scan the "cross" and find max element m.
- If m is a peak then return it, and we are done.
- Otherwise, some neighbor of m is bigger than m. Recurse into that quadrant.

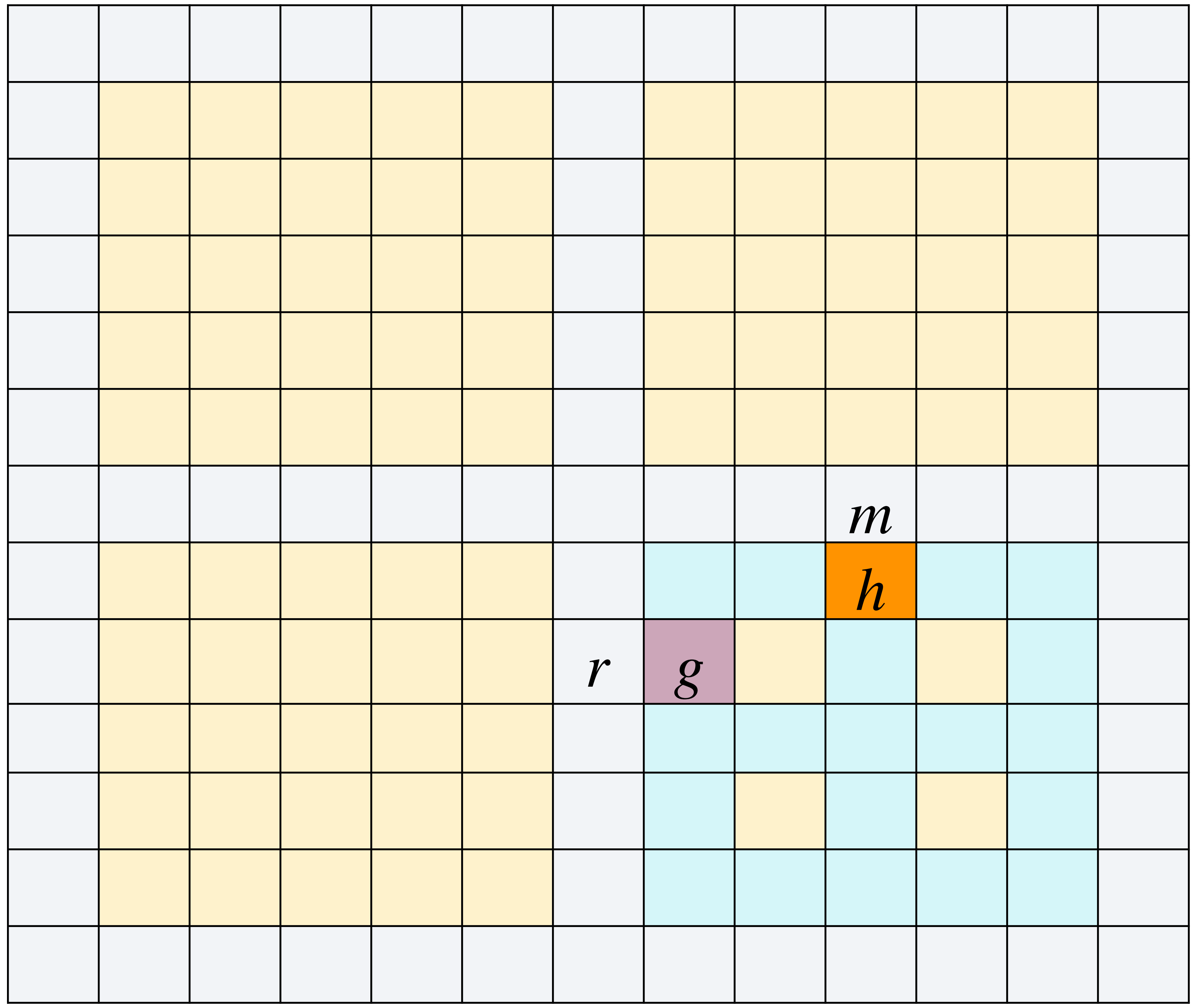


g is peak in this window
g is peak in the original matrix?

Peak finding in 2D, Algorithm III

"cross and boundary" (i.e., a "window frame")

- Scan the "cross" and find max element m.
- If m is a peak then return it, and we are done.
- Otherwise, some neighbor of m is bigger than m. Recurse into that quadrant.



Claim: A peak (found by the algorithm) in the quadrant containing the large neighbor is also a peak in the original matrix.

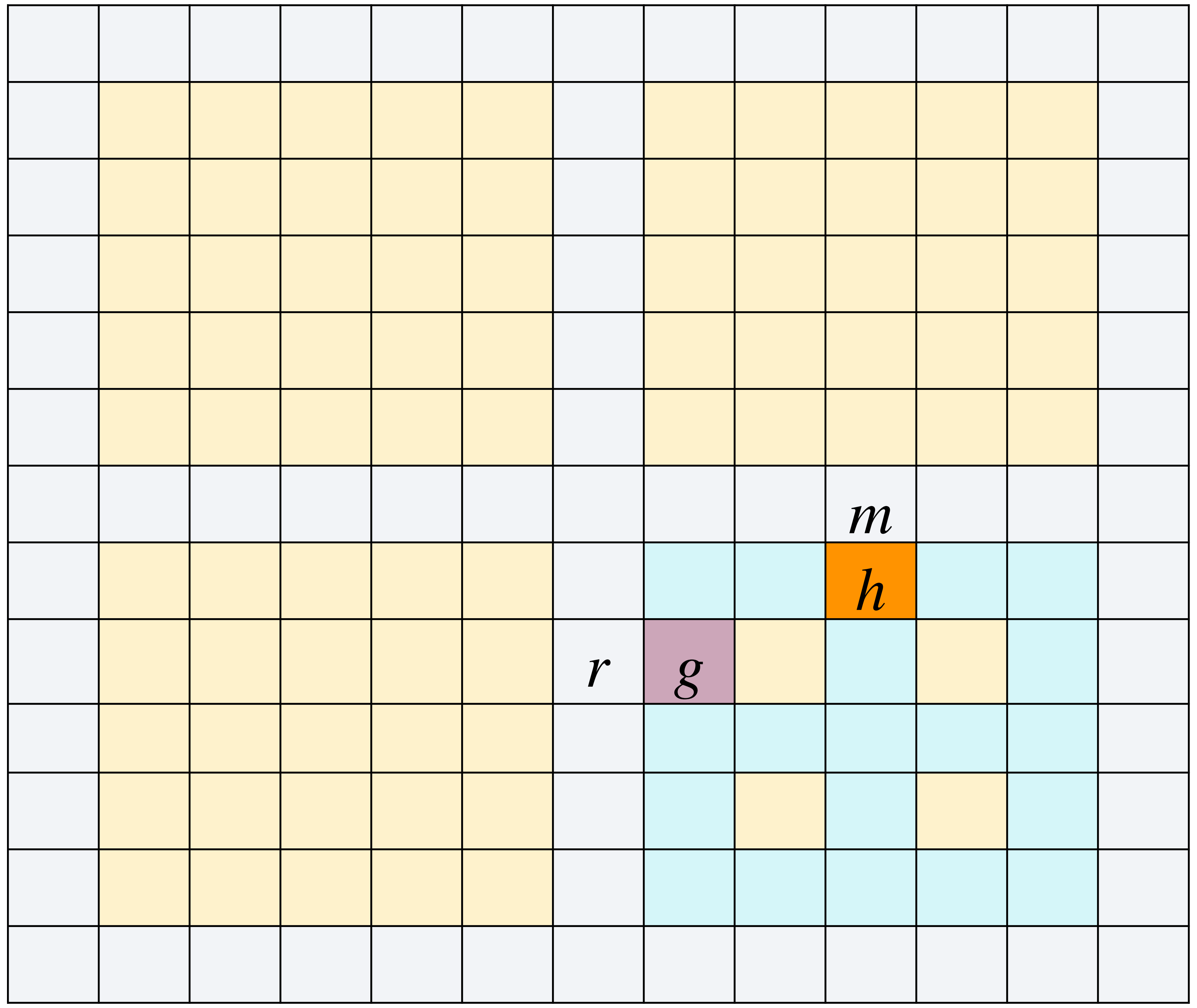
Proof:

- If the peak found by the algorithm in the quadrant is not on the boundary of the quadrant, then clearly it's a peak in the original matrix.
- Otherwise, the peak found by the algorithm in the quadrant is on the boundary of the quadrant (say g); and it's also a peak in the original matrix (since $g \geq h \geq m \geq r$).

Peak finding in 2D, Algorithm III

"cross and boundary" (i.e., a "window frame")

- Scan the "cross" and find max element m.
- If m is a peak then return it, and we are done.
- Otherwise, some neighbor of m is bigger than m. Recurse into that quadrant.



Runtime of this algorithm

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

Further reading

- [CLRS] Ch.2 (2.3), Ch.4
- [Erickson] Ch.1 (excluding 1.5 and 1.8)