Divide-and-Conquer

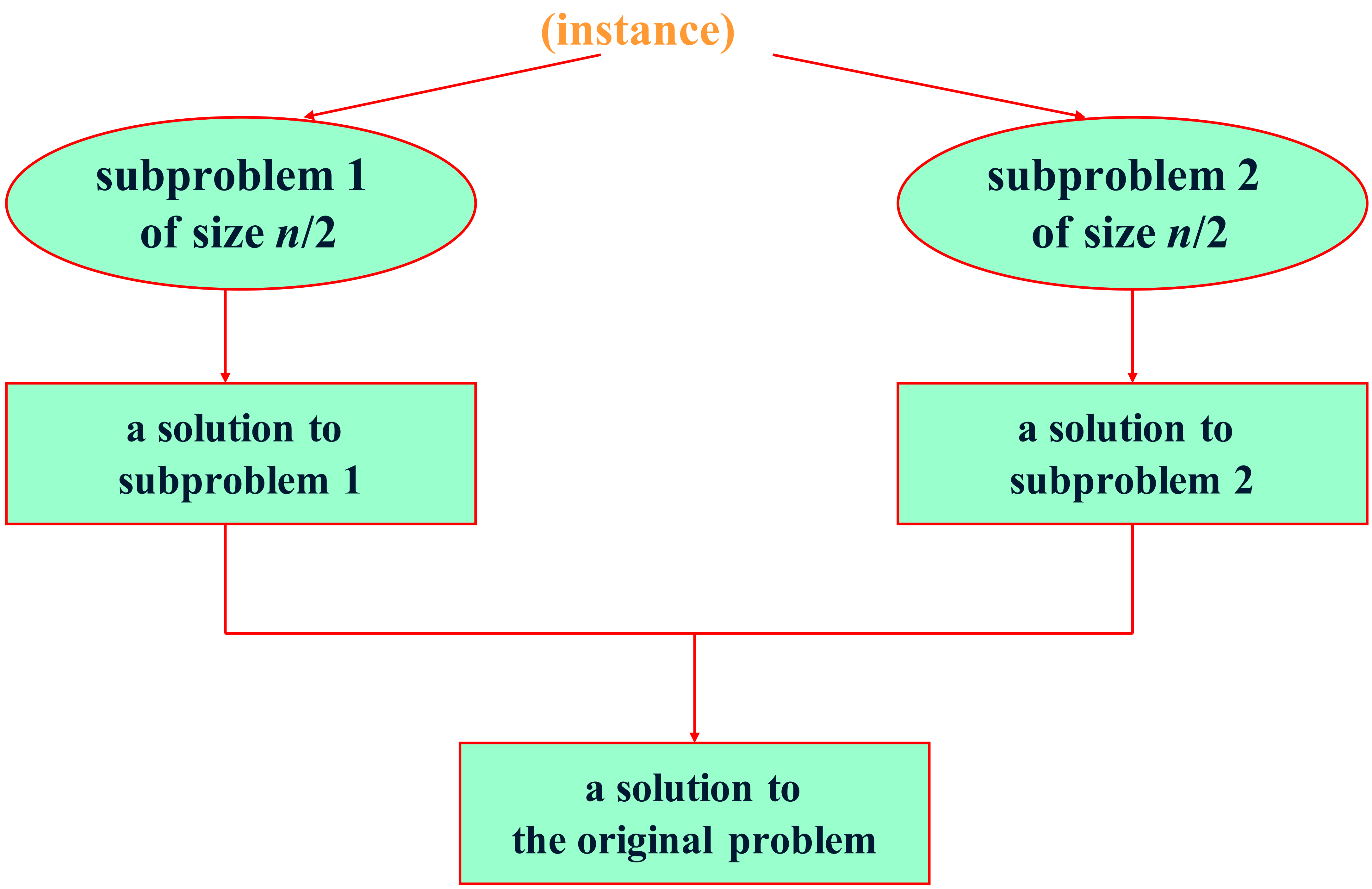
Natasha Dejdumrong

CPE 212 Algorithm Design

Topics

- Merge sort
- Quick sort

General Concept



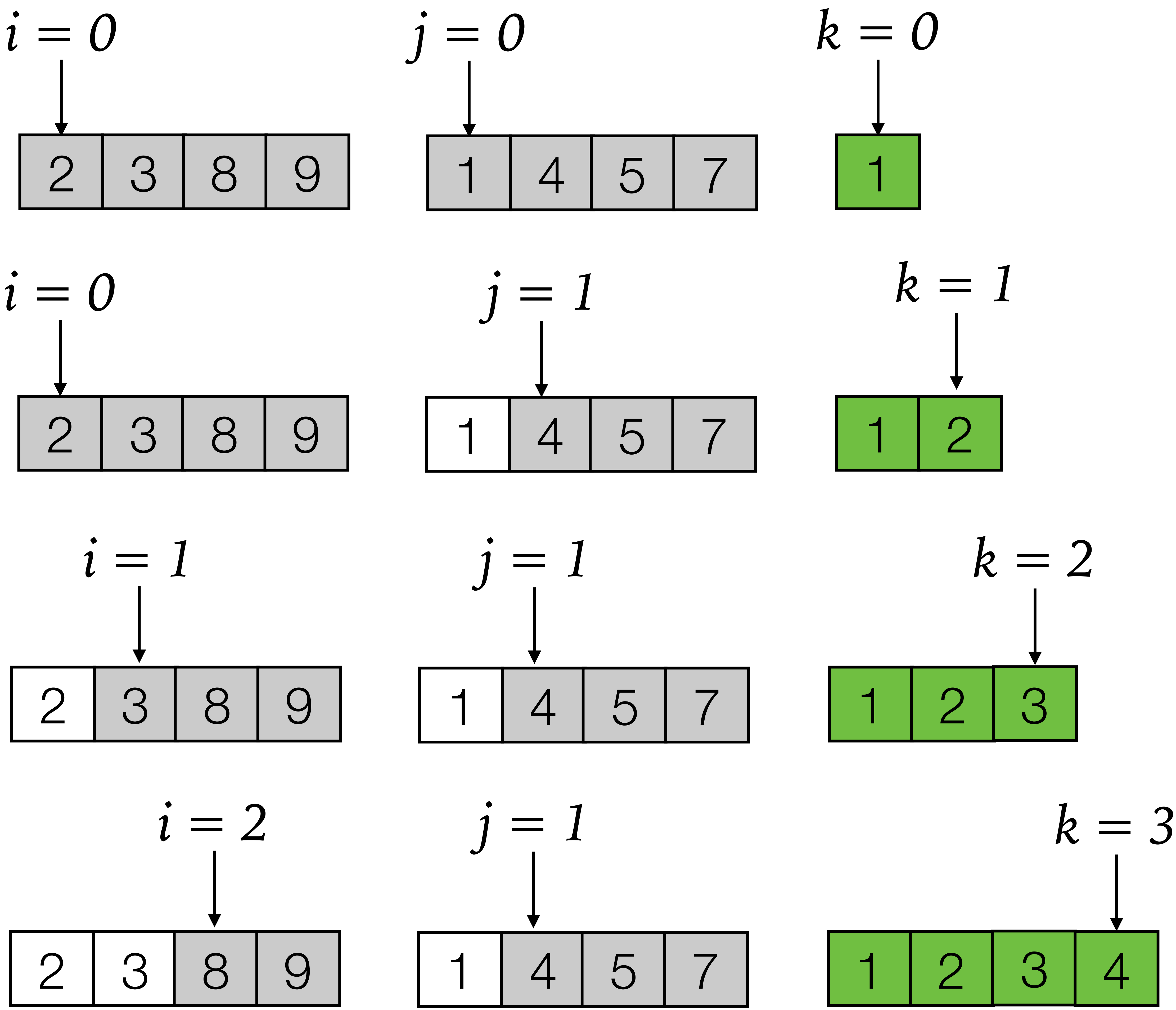
Merge Sort

8	3	2	9	7	1	5	4

Algorithm MergeSort(A)

Input: An array of $A[l \ldots r]$ of orderable elements
Output: Array $A[l \ldots r]$ sorted in non-decreasing order

if A has more than one element **then**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  MergeSort($A[l \ldots m]$) ▷ Sort 1^{st}half of A
  MergeSort($A[m+1 \ldots r]$) ▷ Sort 2^{nd}half of A
  Merge(A, l, m, r) ▷ Merge two halves



i = 0
j = 0
k = 0
2 3 8 9
1 4 5 7
1
i = 0
j = 1
k = 1
2 3 8 9
1 4 5 7
1 2
i = 1
j = 1
k = 2
2 3 8 9
1 4 5 7
1 2 3
i = 2
j = 1
k = 3
2 3 8 9
1 4 5 7
1 2 3 4

Algorithm Merge(A, l, m, r)

Input: Two arrays defined by $A[l \dots m]$, $A[m+1 \dots r]$
Output: Sorted array $A[l \dots r]$

$n_1 \leftarrow$ Size($A[l \dots m]$)
$n_2 \leftarrow$ Size($A[m+1 \dots r]$)
$i, j, k \leftarrow 0$

while $i < n_1$ and $j < n_2$ **do**
    if $A[l+i] < A[m+1+j]$ **then**
        $B[k] \leftarrow A[l+i]$
        $i \leftarrow i+1$
    else
        $B[k] \leftarrow A[m+1+j]$
        $j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = n_1$ **then** ▷ Left array empty first
    Copy $A[j \dots n_2 - 1]$ to $B[k \dots n_1 + n_2 - 1]$
else ▷ Right array empty first
    Copy $A[i \dots n_1 - 1]$ to $B[k \dots n_1 + n_2 - 1]$
Copy B to $A[l \dots r]$

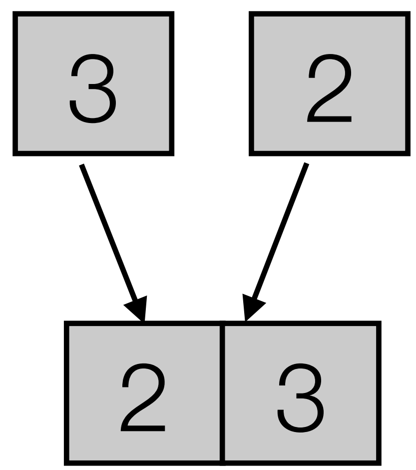
Efficiency of Merge Sort

- Basic operation is the key comparison
  - Split the list into two halfs
  - Sort each half
  - Merge both halfs
- C(n) = # key comparisons for a list of size n

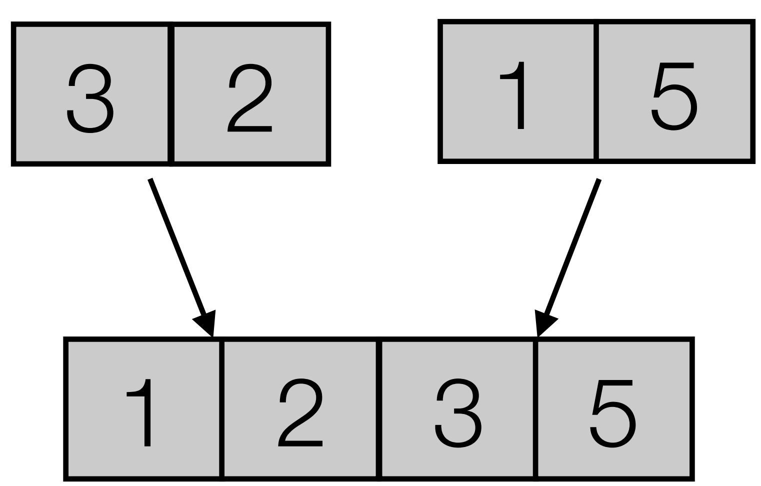
$$C(n) = \begin{cases} 0 & n = 1 \\ 2C(n/2) + C_{\textsf{merge}}(n) & n > 1 \end{cases}$$

- How many worst-case comparions to merge a list of size n?

$n = 2$



$n = 4$



- The worst-case number of comparisons in the merge sort is

$$C_{\text{worst}}(n) = \begin{cases} 0 & n = 1 \\ 2C_{\text{worst}}(n/2) + n - 1 & n > 1 \end{cases}$$

  - Solved by backward substitution
  - Alternatively from Master theorem

General Divide-and-Conquer

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

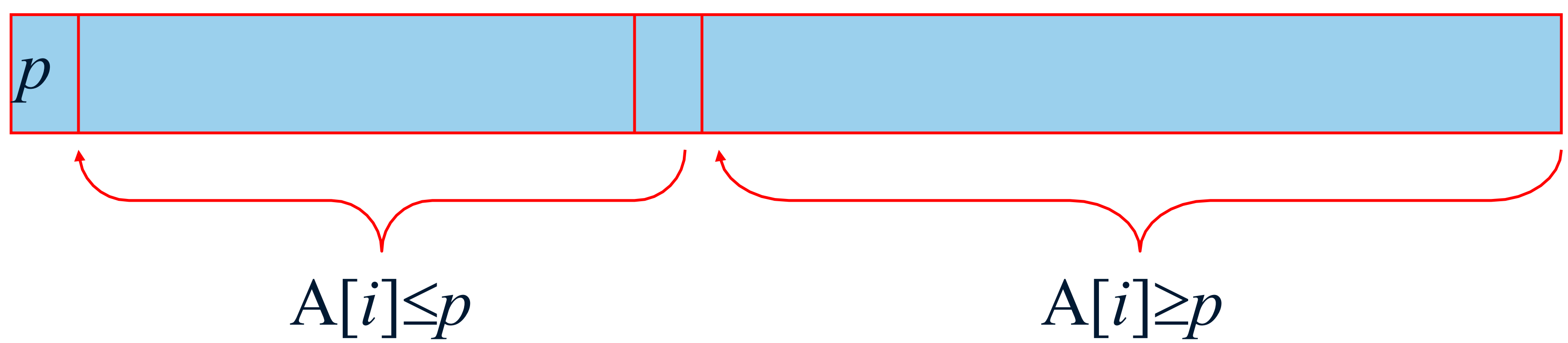
Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

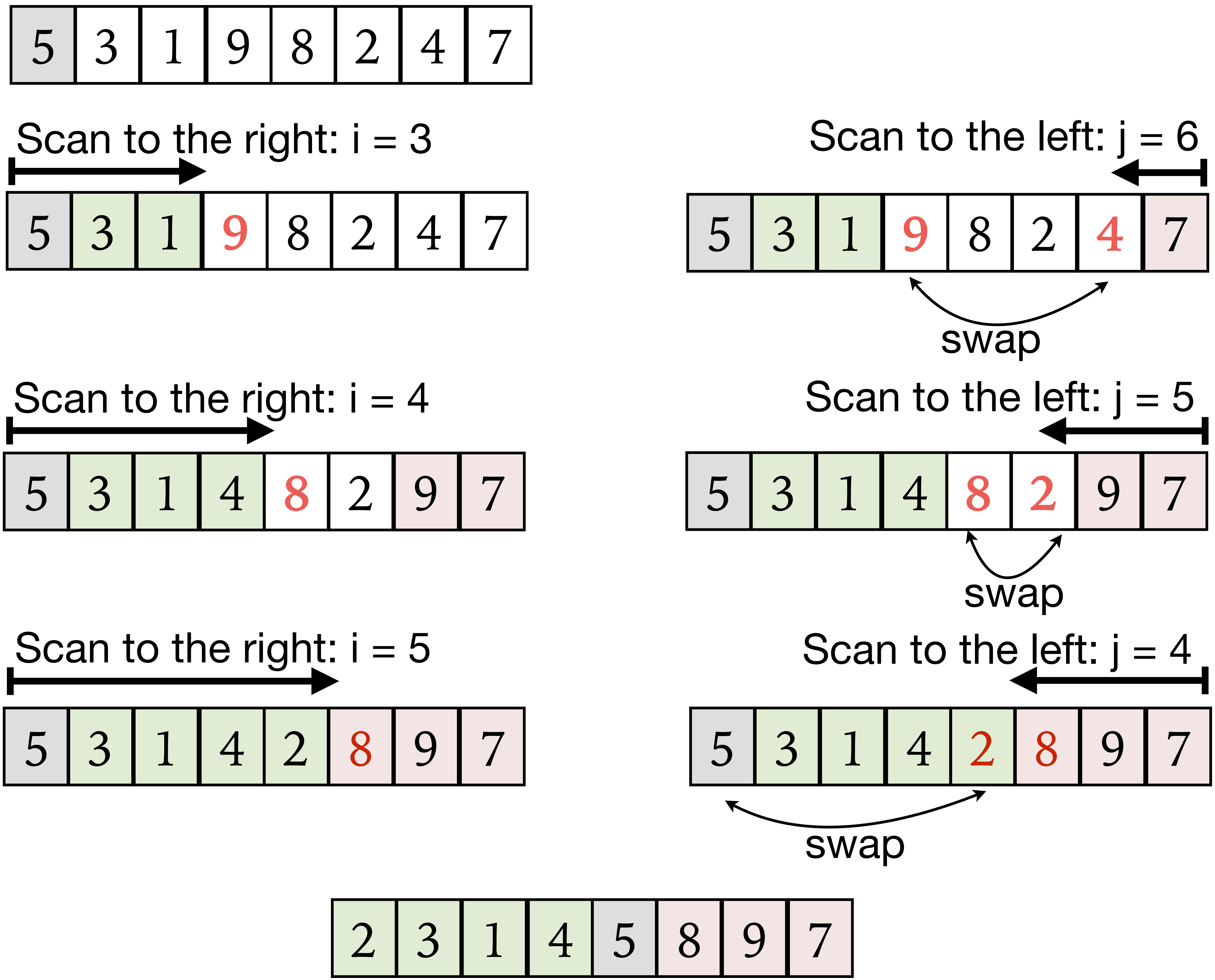
$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

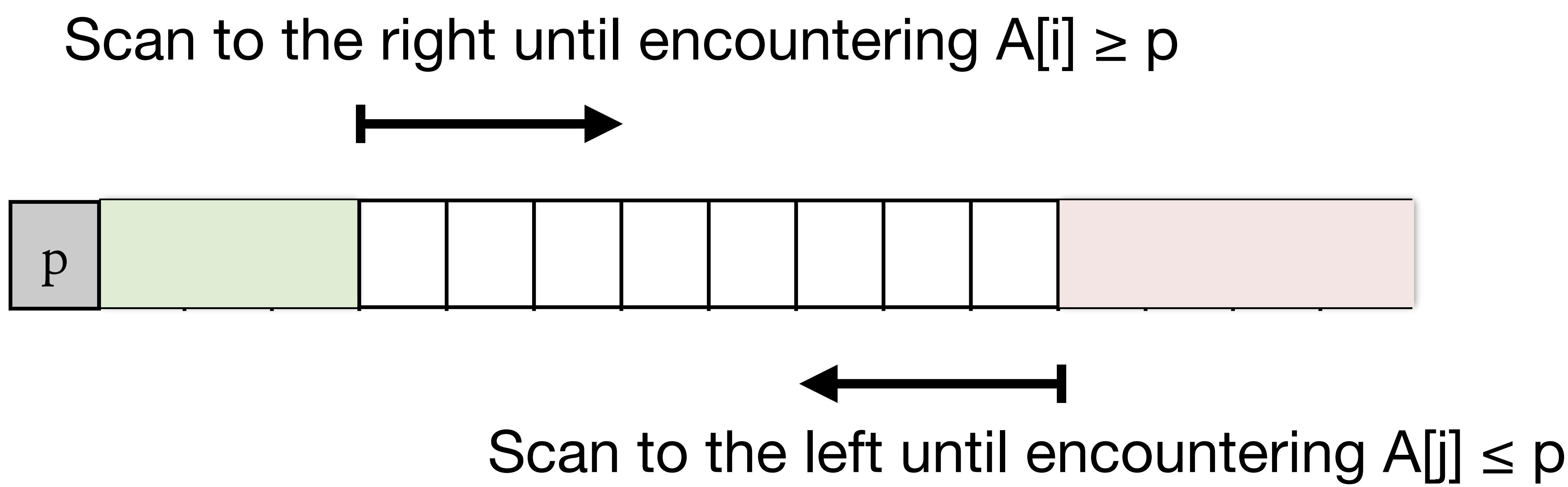
$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

Quick Sort

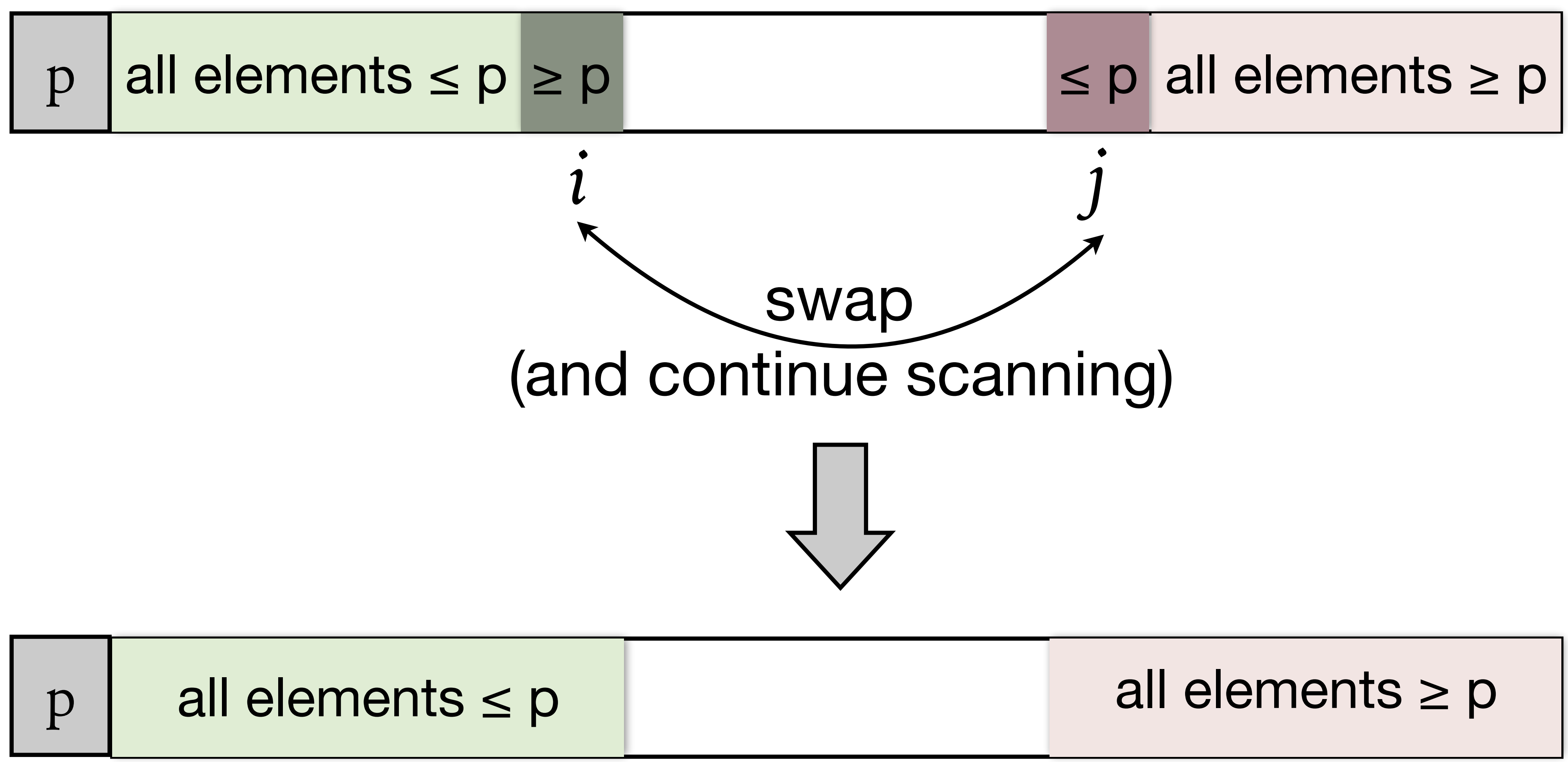


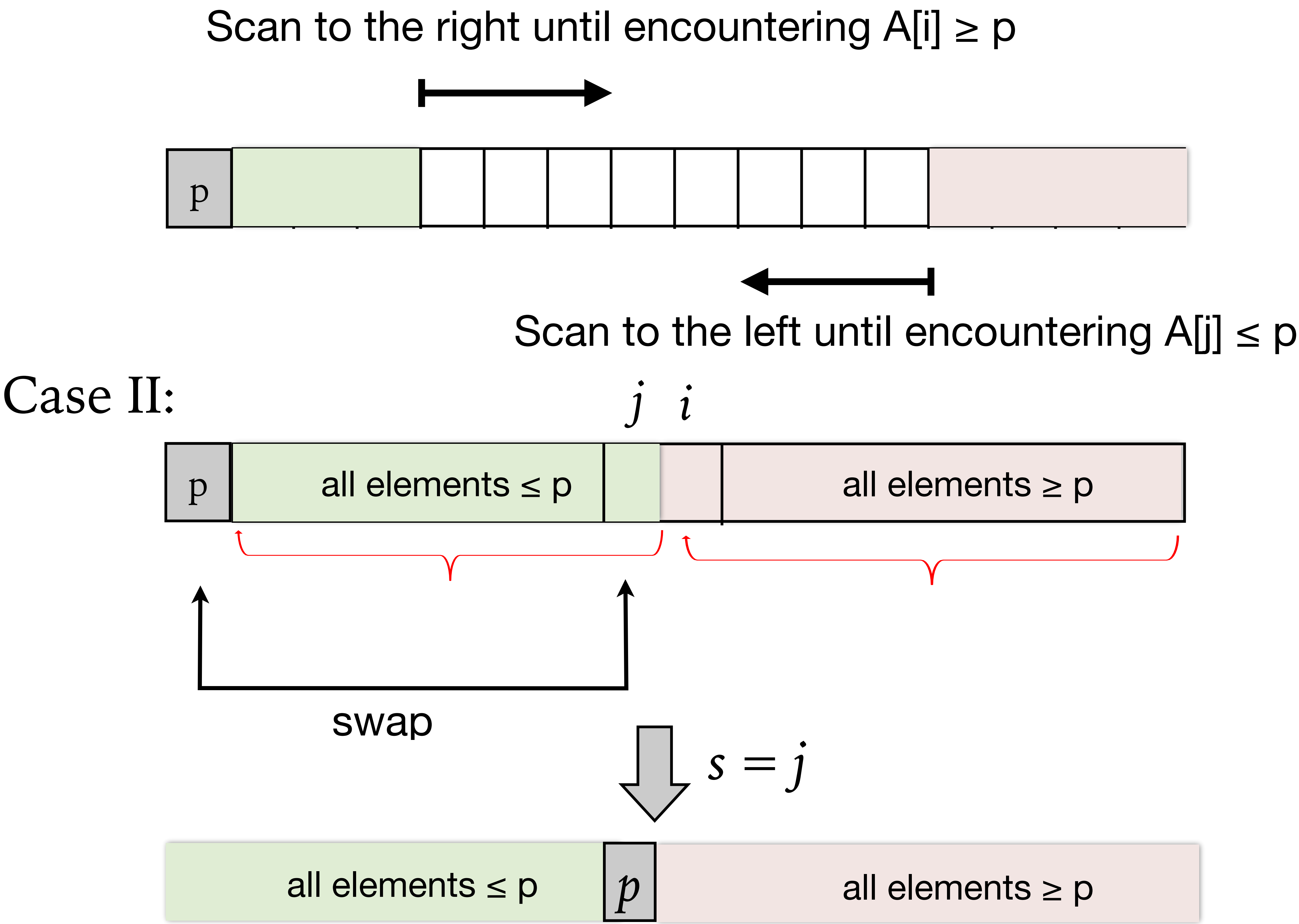
- Select pivot element p (first one)
- Rearrange the list into two subarrays L, R such that
  - L has s elements less than equal to p
  - R has remaining elements greater than equal to p
- Exchange the pivot element with the last element in L
- Sort L and R recursively


5 3 1 9 8 2 4 7
Scan to the right: i = 3
5 3 1 9 8 2 4 7
Scan to the left: j = 6
5 3 1 9 8 2 4 7
swap
Scan to the right: i = 4
5 3 1 4 8 2 9 7
Scan to the left: j = 5
5 3 1 4 8 2 9 7
swap
Scan to the right: i = 5
5 3 1 4 2 8 9 7
Scan to the left: j = 4
5 3 1 4 2 8 9 7
swap
2 3 1 4 5 8 9 7

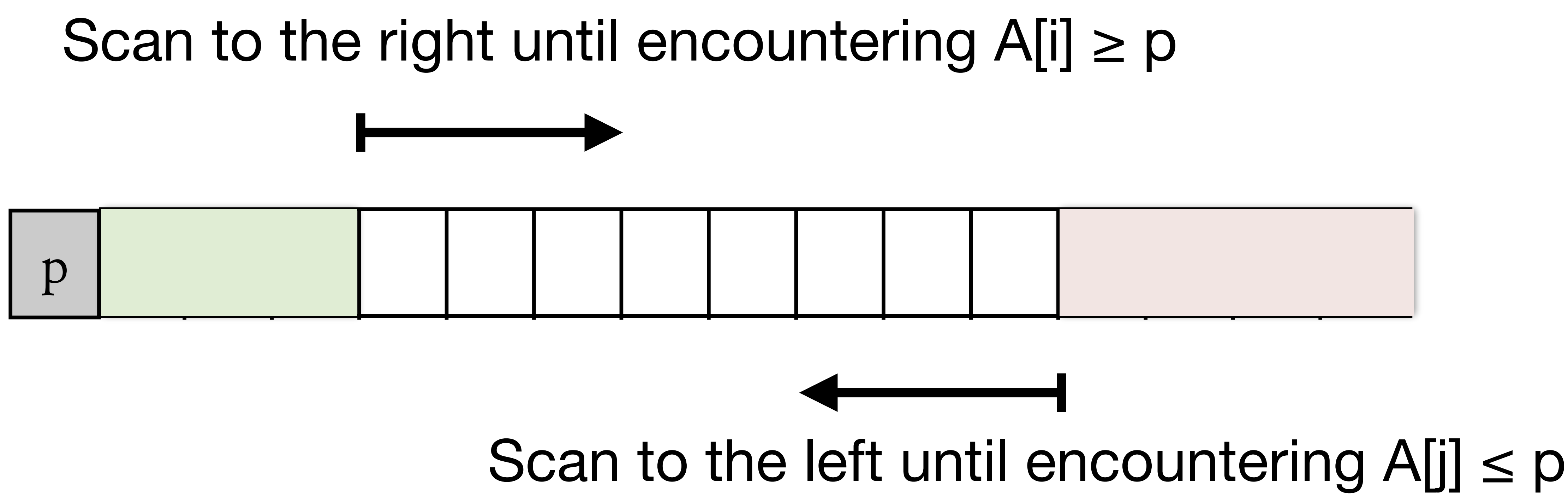

Scan to the right until encountering A[i] ≥ p
p
Scan to the left until encountering A[j] ≤ p

Case I:

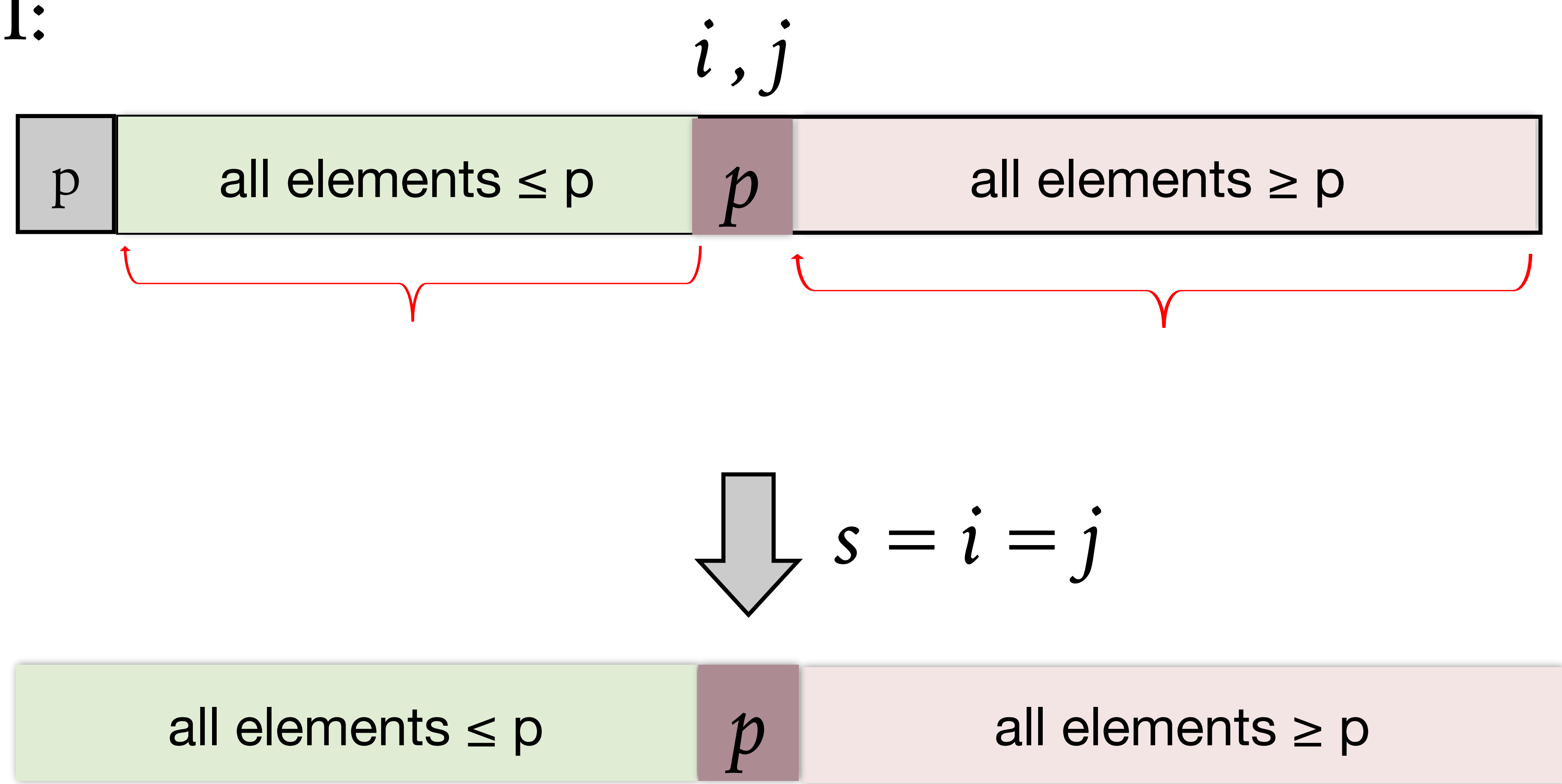

p
all elements ≤ p
≥ p
≤ p
all elements ≥ p
i
j
swap
(and continue scanning)
p
all elements ≤ p
all elements ≥ p



Scan to the right until encountering A[i] ≥ p
p
Scan to the left until encountering A[j] ≤ p
Case II:
j
i
p
all elements ≤ p
all elements ≥ p
swap
s = j
all elements ≤ p
p
all elements ≥ p


Scan to the right until encountering A[i] ≥ p
p
Scan to the left until encountering A[j] ≤ p

Case III:

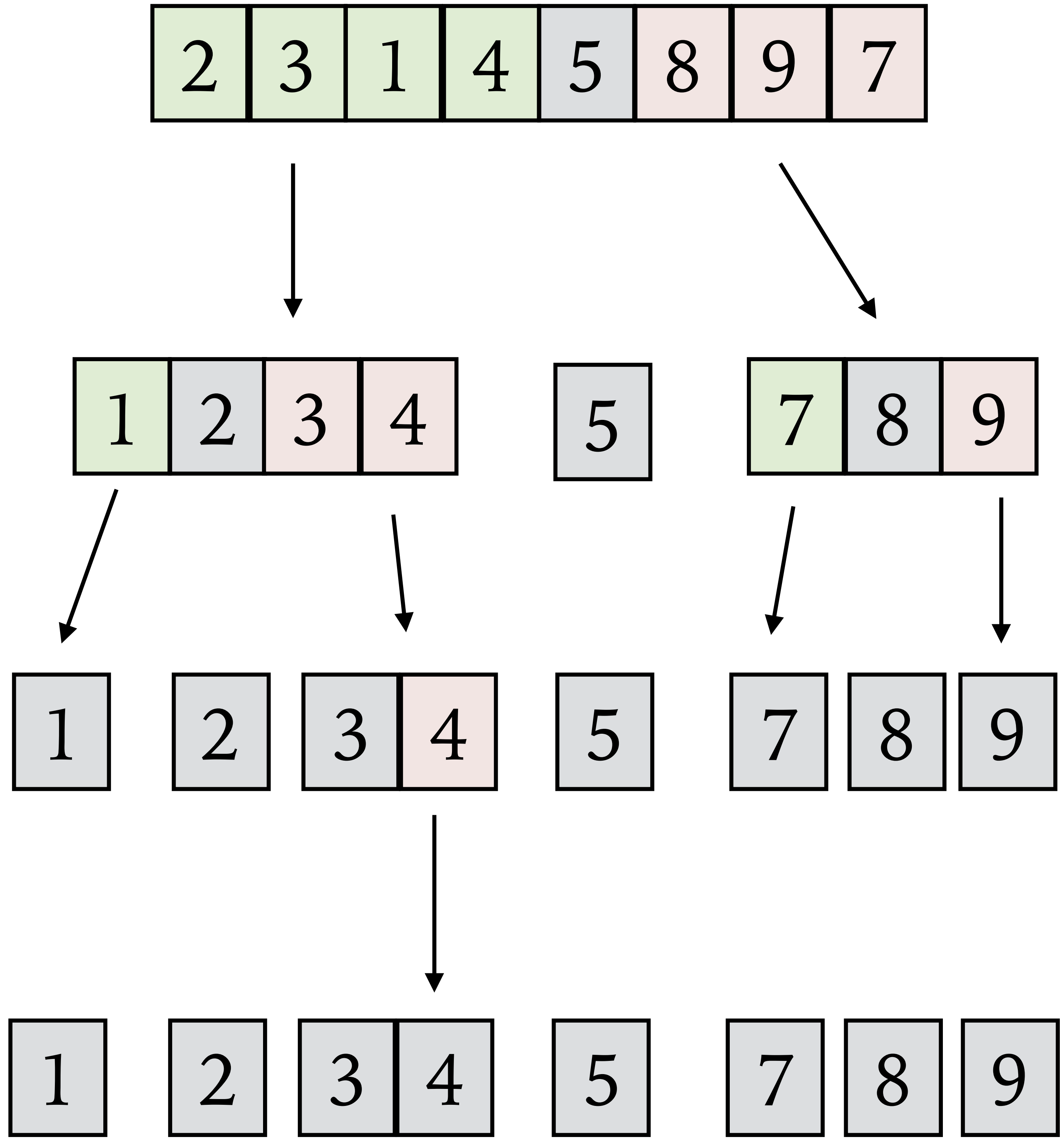

i , j
p
all elements ≤ p
p
all elements ≥ p
s = i = j
all elements ≤ p
p
all elements ≥ p

Algorithm HoarePartition

Input: Array $A[l \ldots r]$ of orderable elements
Output: A partition of $A[l \ldots r]$ with the split position as return value

$p \leftarrow A[l]$
$i \leftarrow l$
$j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$
    repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$
    swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ $\triangleright$ Undo last swap when $i \geq j$
swap$(A[l], A[j])$
Return j

After partitioning with 5



Algorithm QuickSort

Input: Array $A[l \dots r]$ of orderable elements

Output: Array $A[l \dots r]$ in non-decreasing order

if $l < r$ **then**

  $s \leftarrow \text{Partition}(A[l \dots r]))$

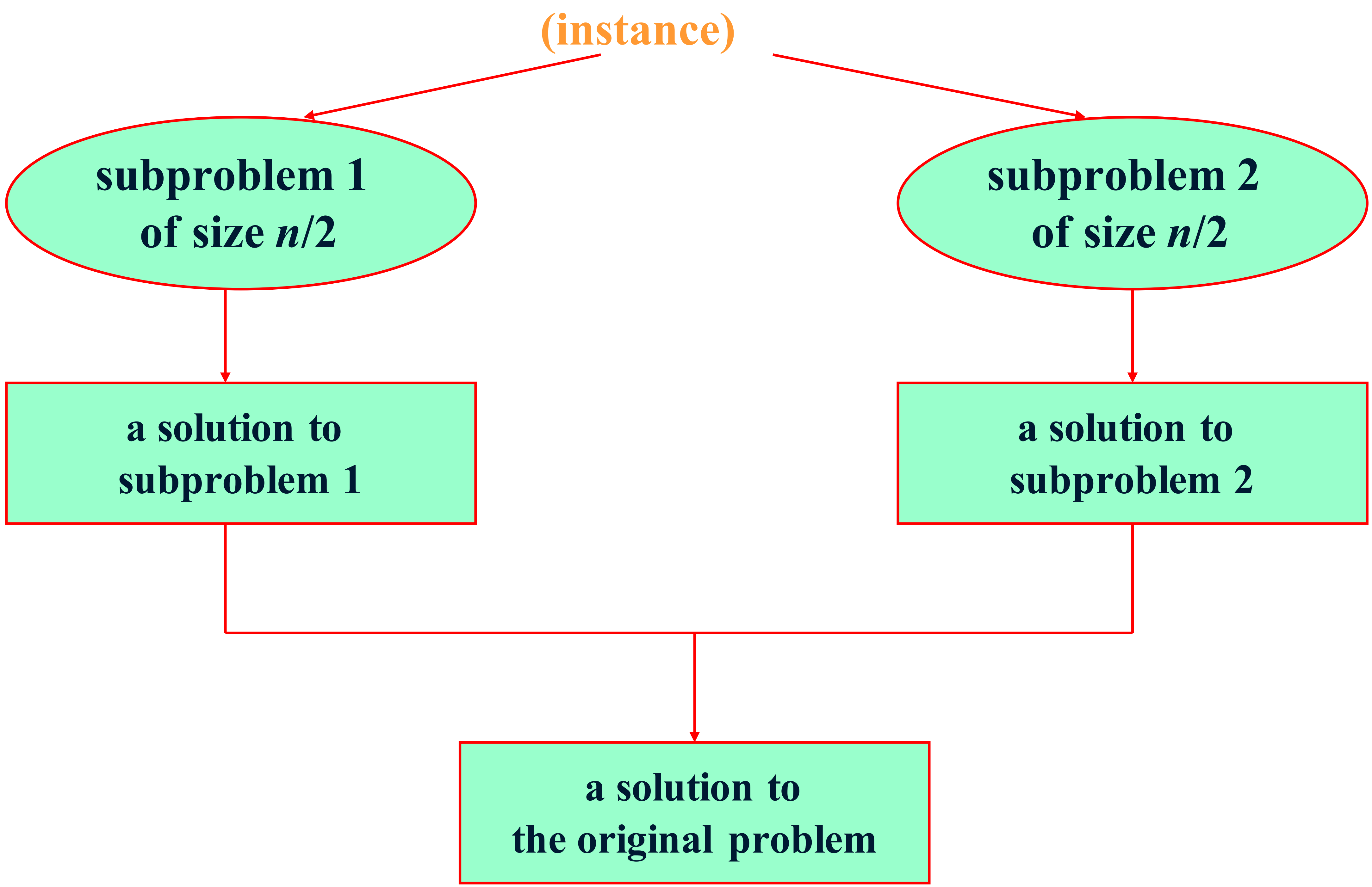
  $\text{QuickSort}(A[l \dots s-1])$

  $\text{QuickSort}(A[s+1 \dots r])$

Efficiency of Quick Sort

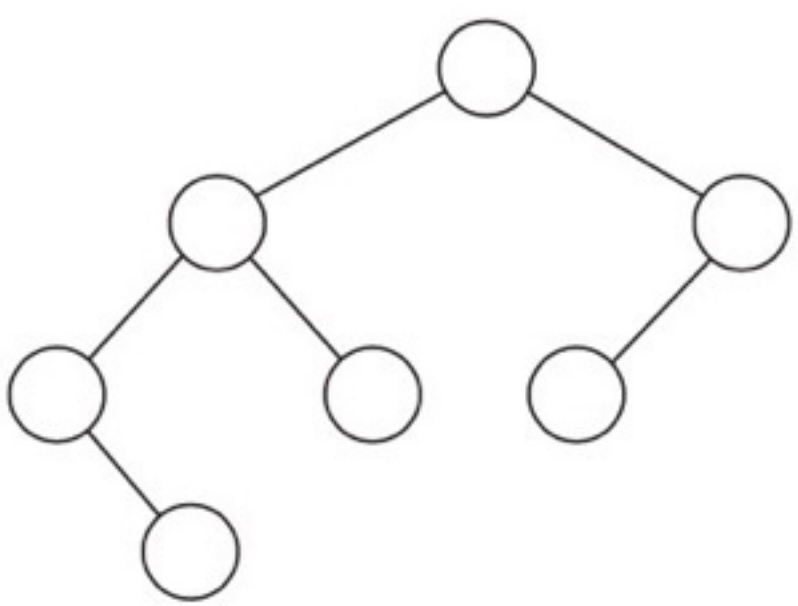
- Best case $\Theta(n \log_2 n)$
  - All the splits happen in the middle of subarrays
- Worst case $\Theta(n^2)$
  - Input is a strictly increasing array (problem already solved).
  - One of the two subarrays after splitting is always empty.
- Average case $\Theta(n \log_2 n)$
  - Split position is equally likely.

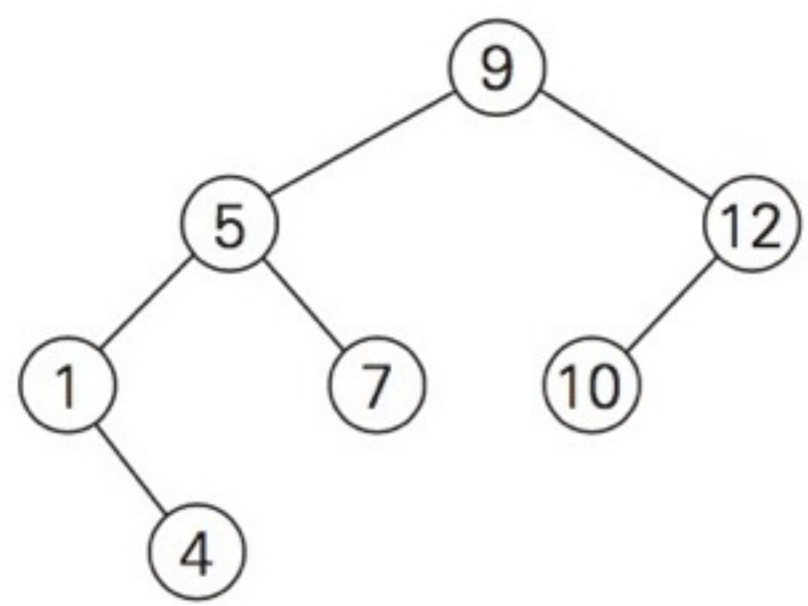
General Concept



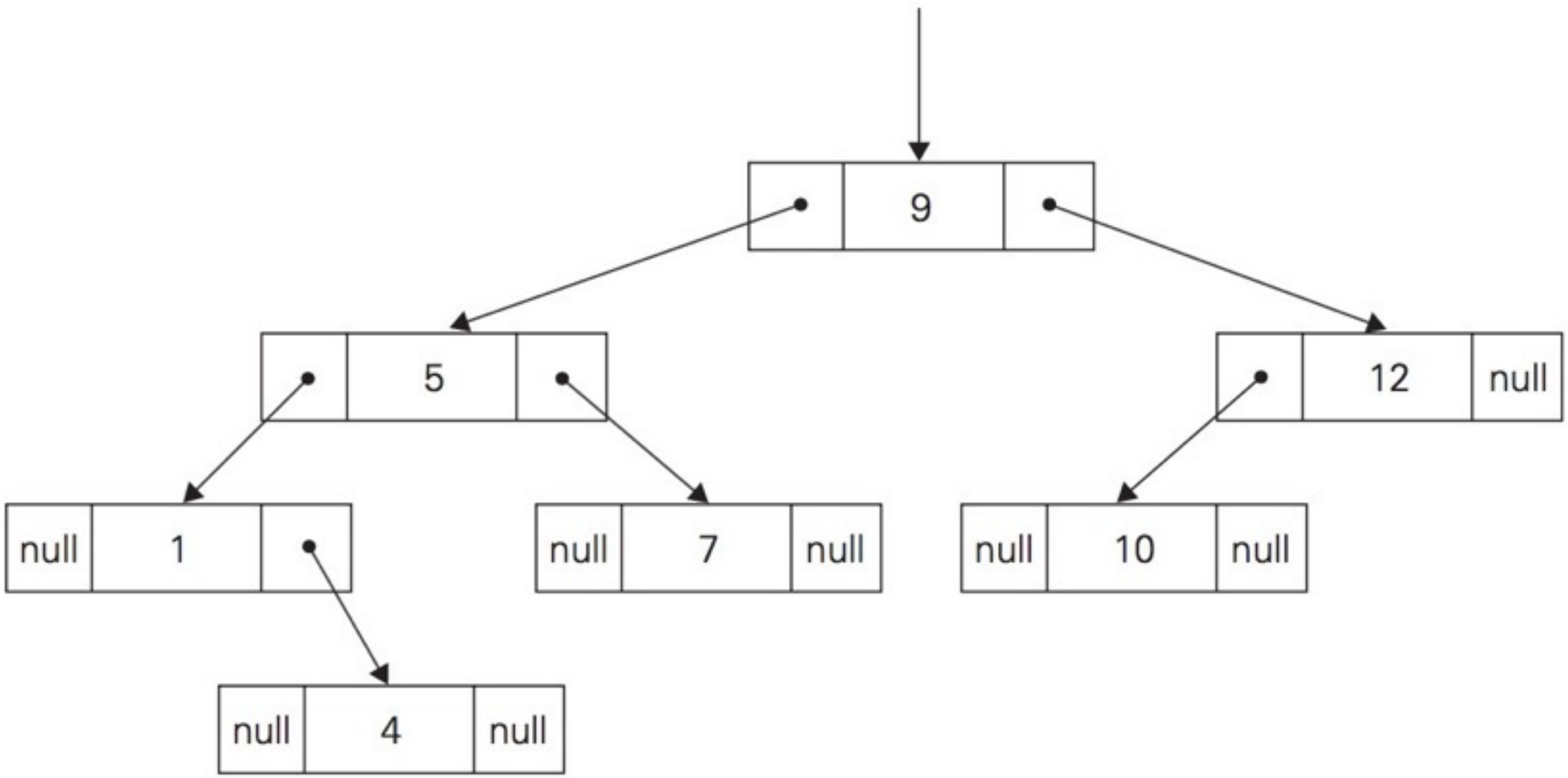
Binary Tree

- Ordered tree = Rooted tree in which all the children of each vertex are ordered.
- Binary tree = Ordered tree where vertex has at most two children (left child, right child)



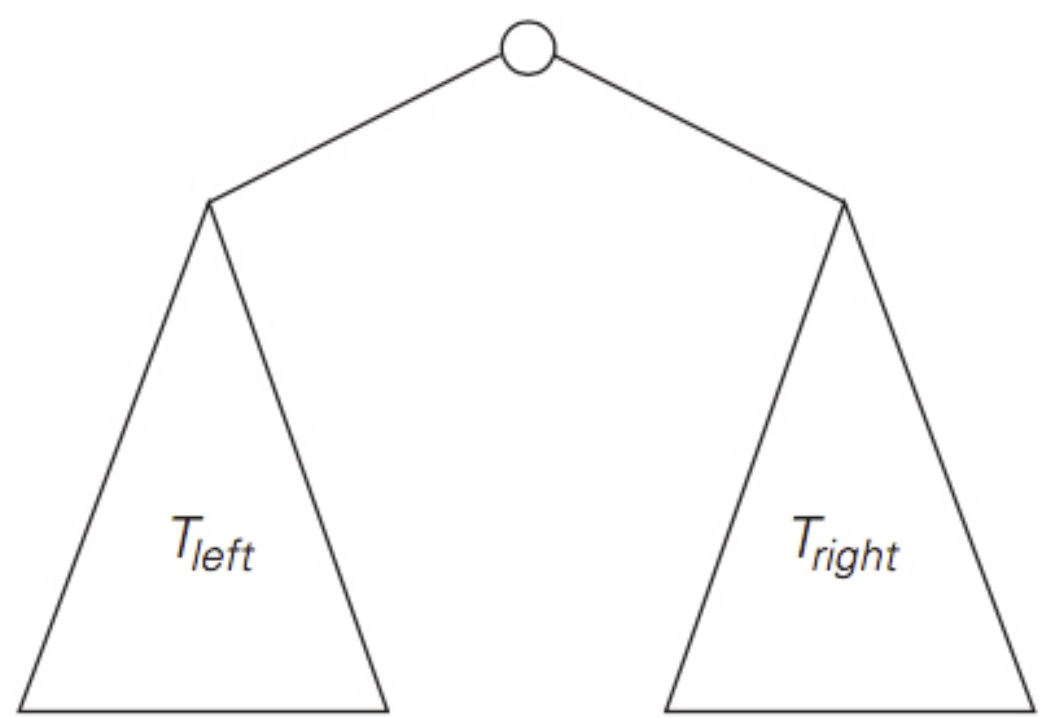


Implementation of Binary Tree



Computing Height of a Binary Tree

- Length of the longest path from the root to the leaf.
- Computed as max of the heights of the root's left and right subtrees plus 1.



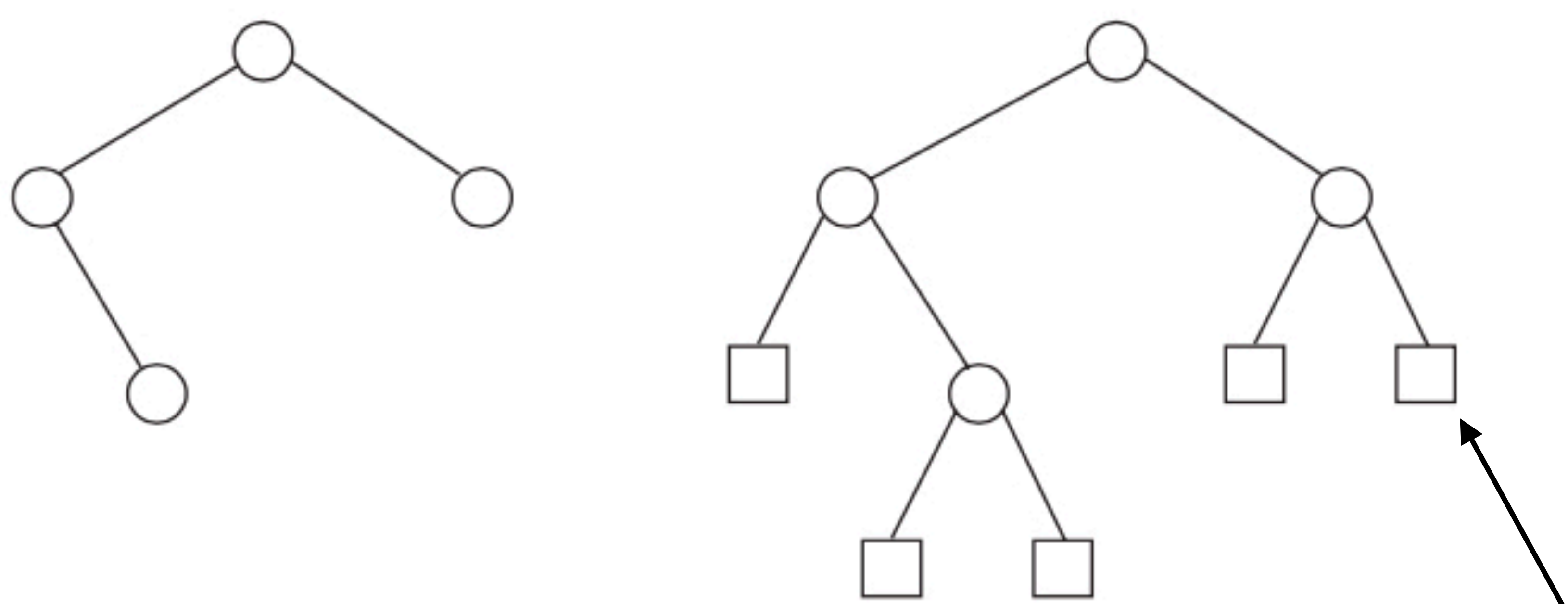
Algorithm Height(T)

```
Input: A binary tree T
Output: The height of T

if T = ∅ then
    return -1
else
    return max{ Height(T_left), Height(T_right) } +1
```

Efficiency of Height()

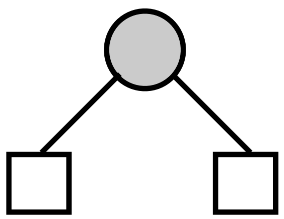
- Basic operation (executed in every call) is to check whether the tree if empty.
- # calls made = # internal nodes + # external nodes



(a) (b)

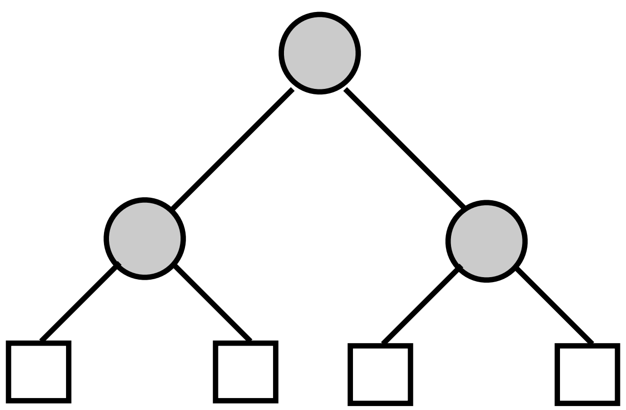
External node to trace
basic operations

- How many external nodes (x) for n internal nodes ?

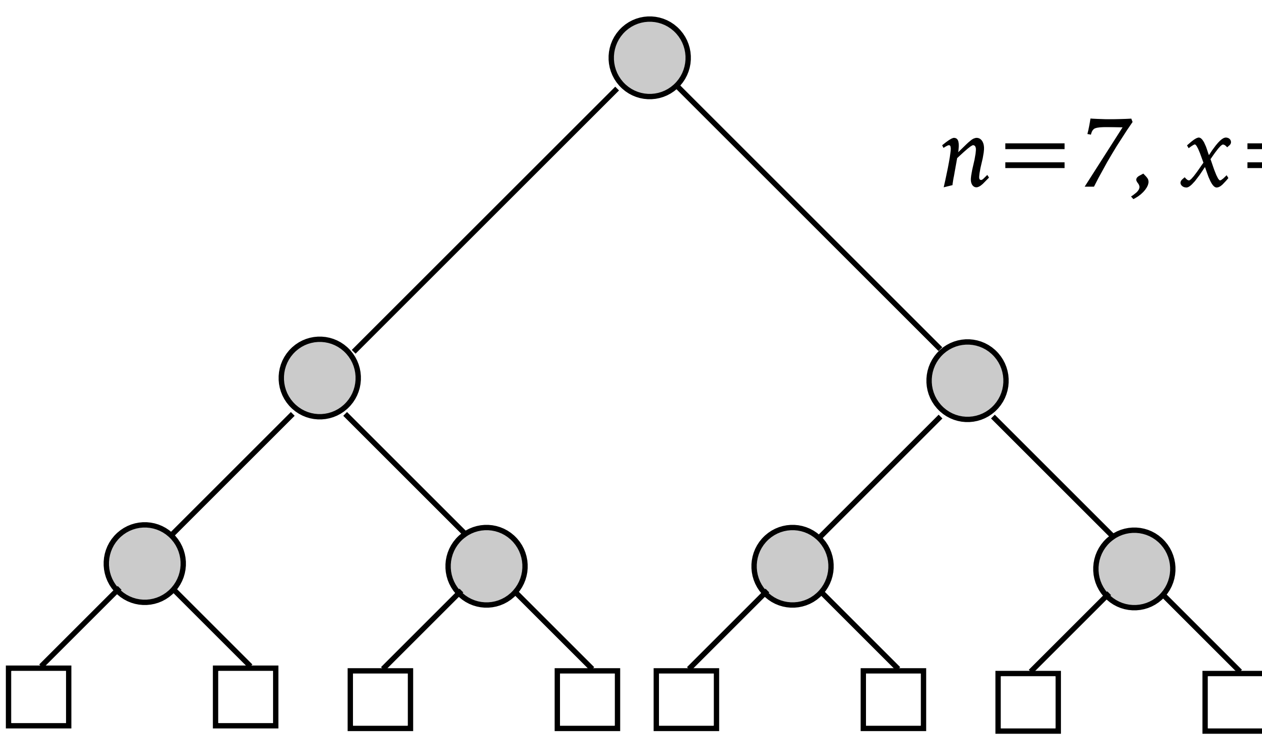


$n=1, x=2$

Full binary tree only
(zero or two children)



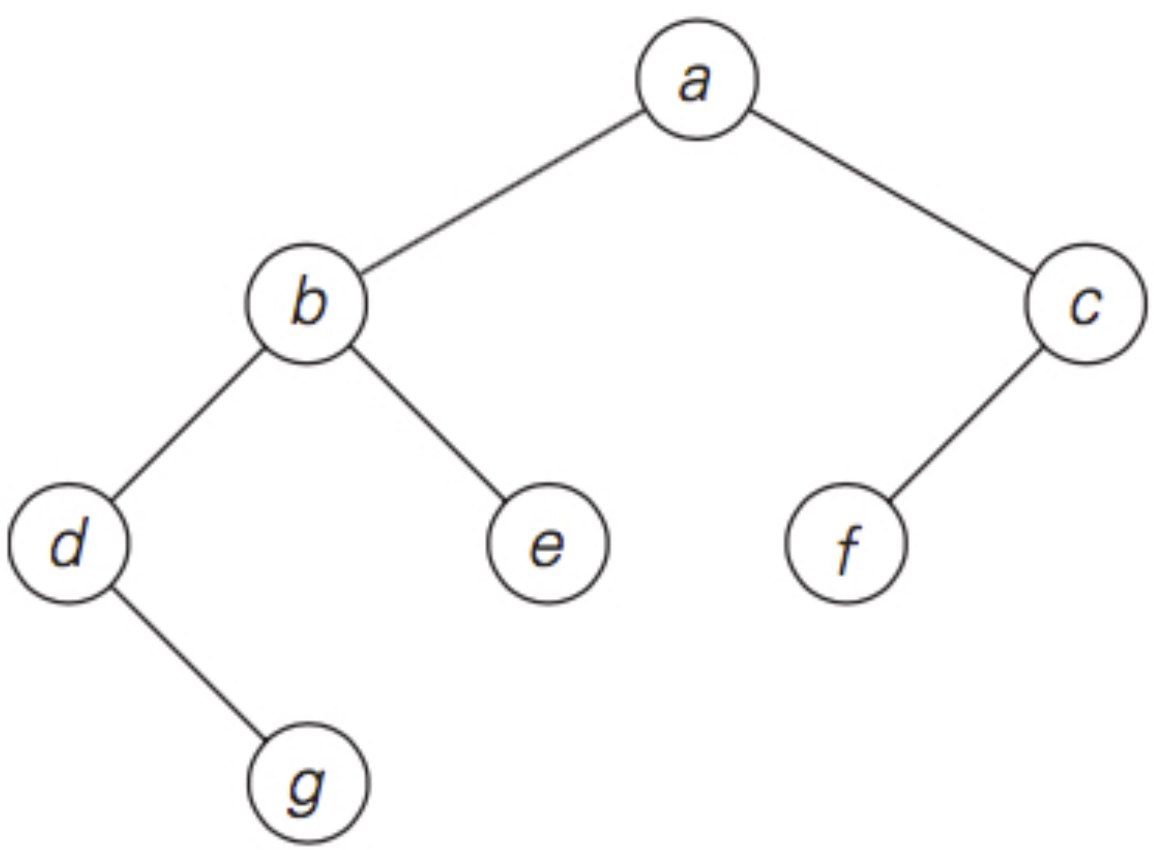
$n=3, x=4$



$n=7, x=8$

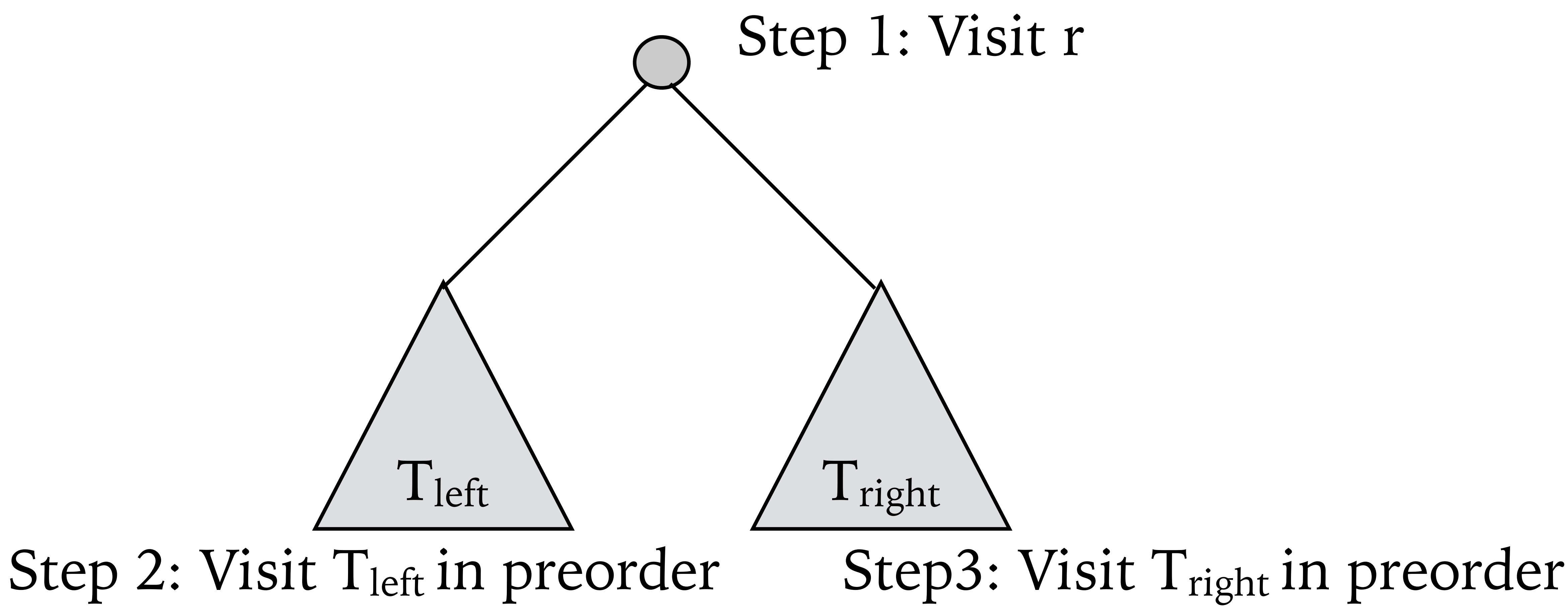
Binary Tree Traversal

- **Preorder:** Root before left and right subtrees.
- **Inorder:** Root after left subtree but before right subtree.
- **Postorder:** Root after left and right subtrees.



preorder: *a, b, d, g, e, c, f*
inorder: *d, g, b, e, a, f, c*
postorder: *g, d, e, b, f, c, a*

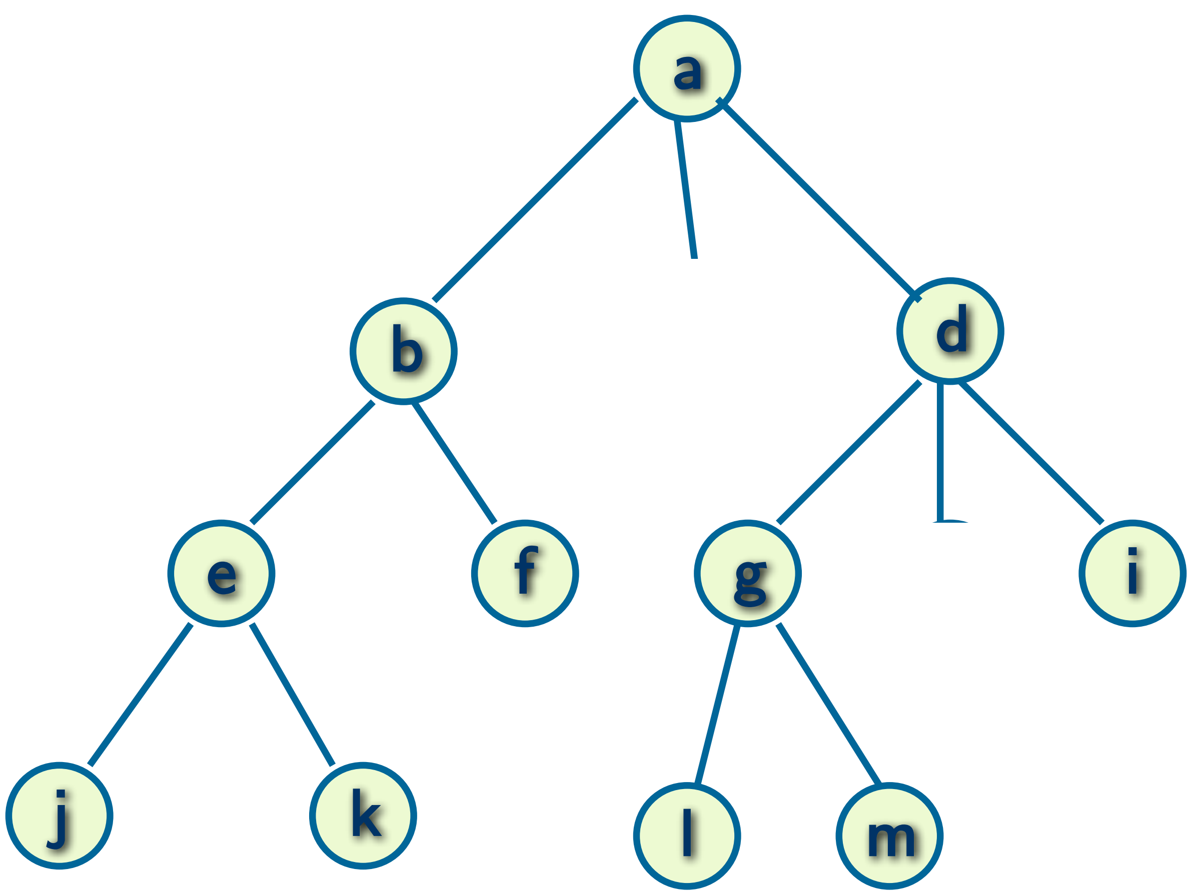
Preorder Traversal



Algorithm Preorder(T)

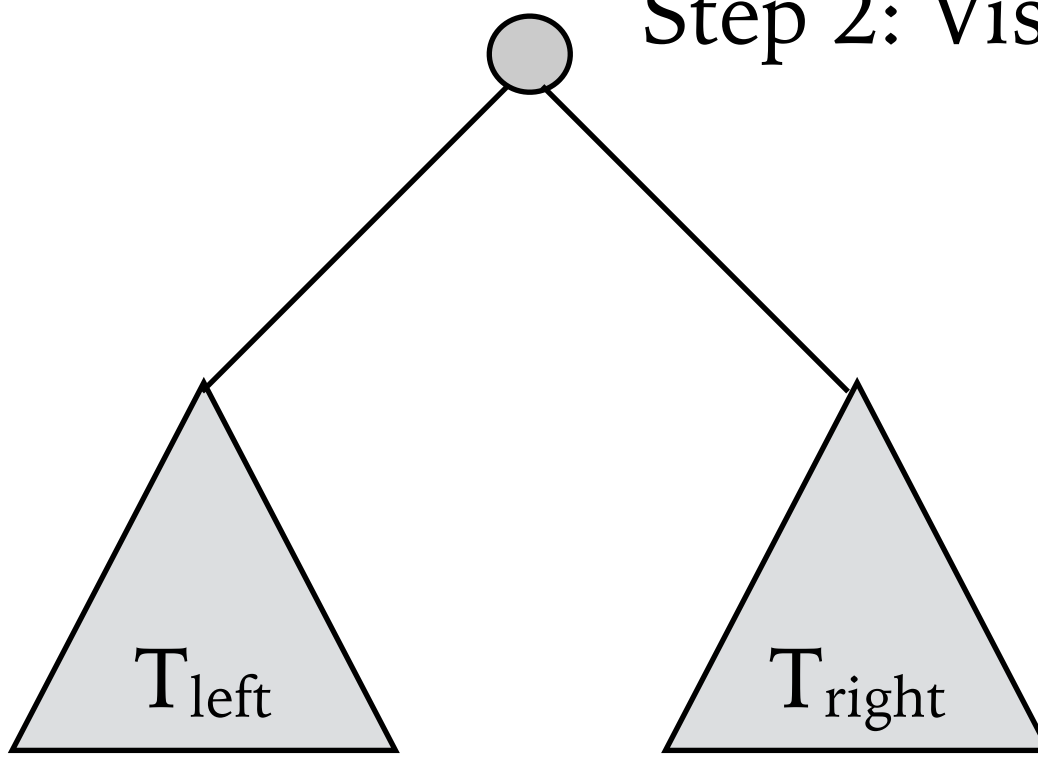
```
Input: A binary tree T
Output: The sequence of nodes from preorder traversal of T

if T ≠ ∅ then
    Visit root of T
    Preorder(T_left)
    Preorder(T_right)
```



Inorder Traversal

Step 2: Visit r



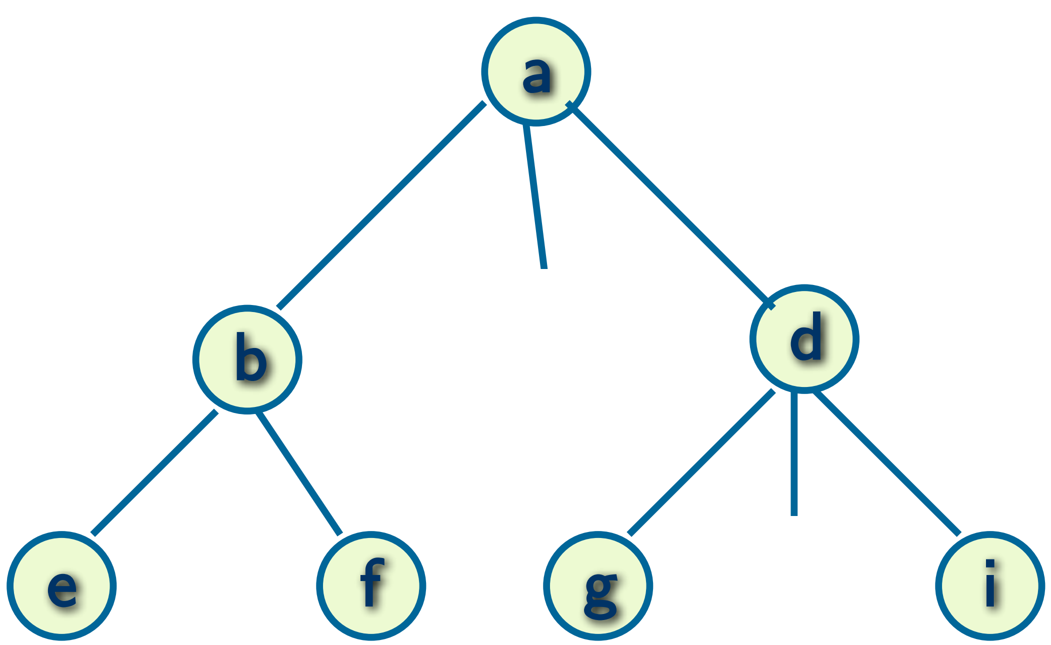
Step 1: Visit T_{left} in inorder

Step3: Visit T_{right} in inorder

Algorithm Inorder(T)

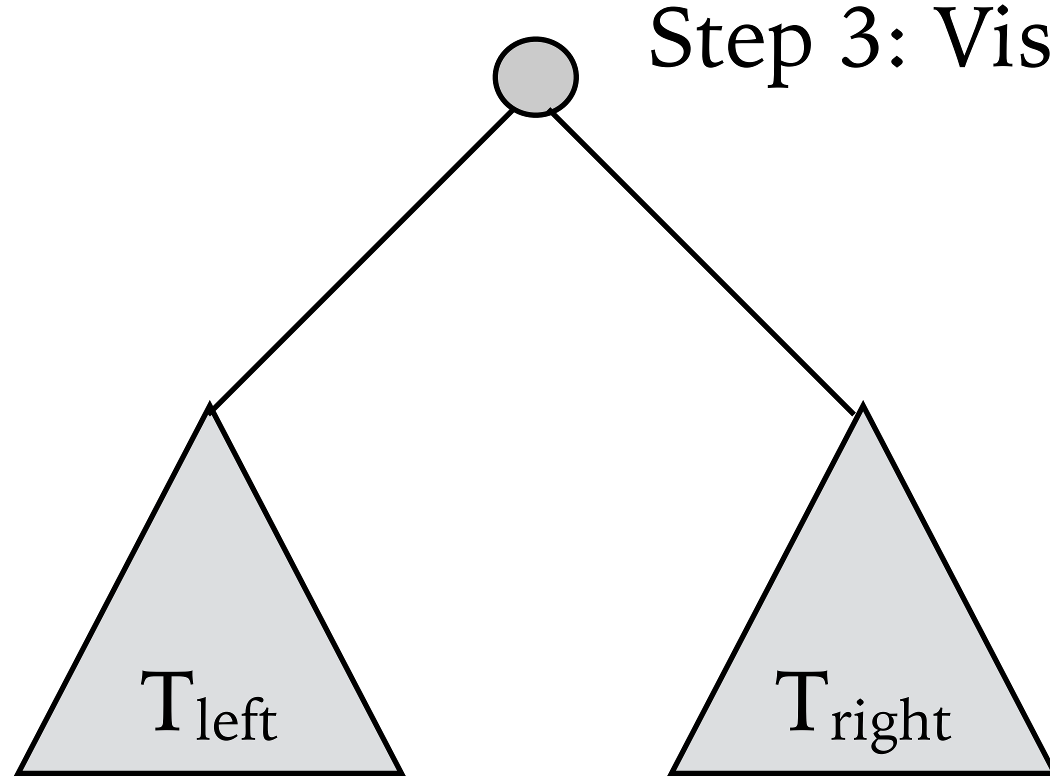
```
Input: A binary tree T
Output: The sequence of nodes from inorder traversal of T

if T_left ≠ ∅ then
    Inorder(T_left)
Visit root of T
if T_right ≠ ∅ then
    Inorder(T_right)
```



Postorder Traversal

Step 3: Visit r

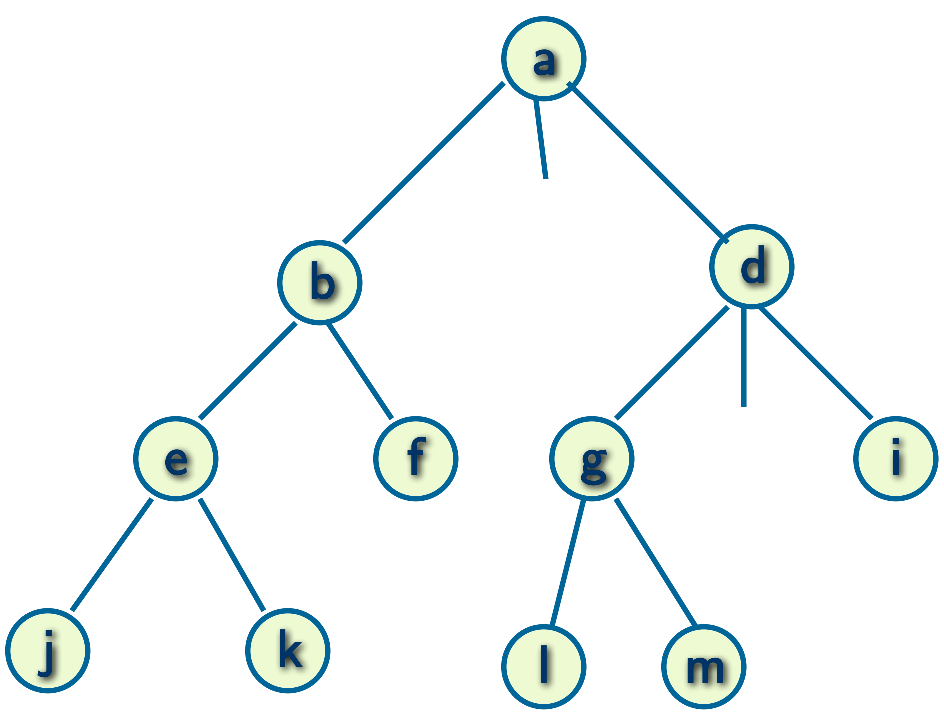


Step 1: Visit T_{left} in postorder   Step 2: Visit T_{right} in postorder

Algorithm Postorder(T)

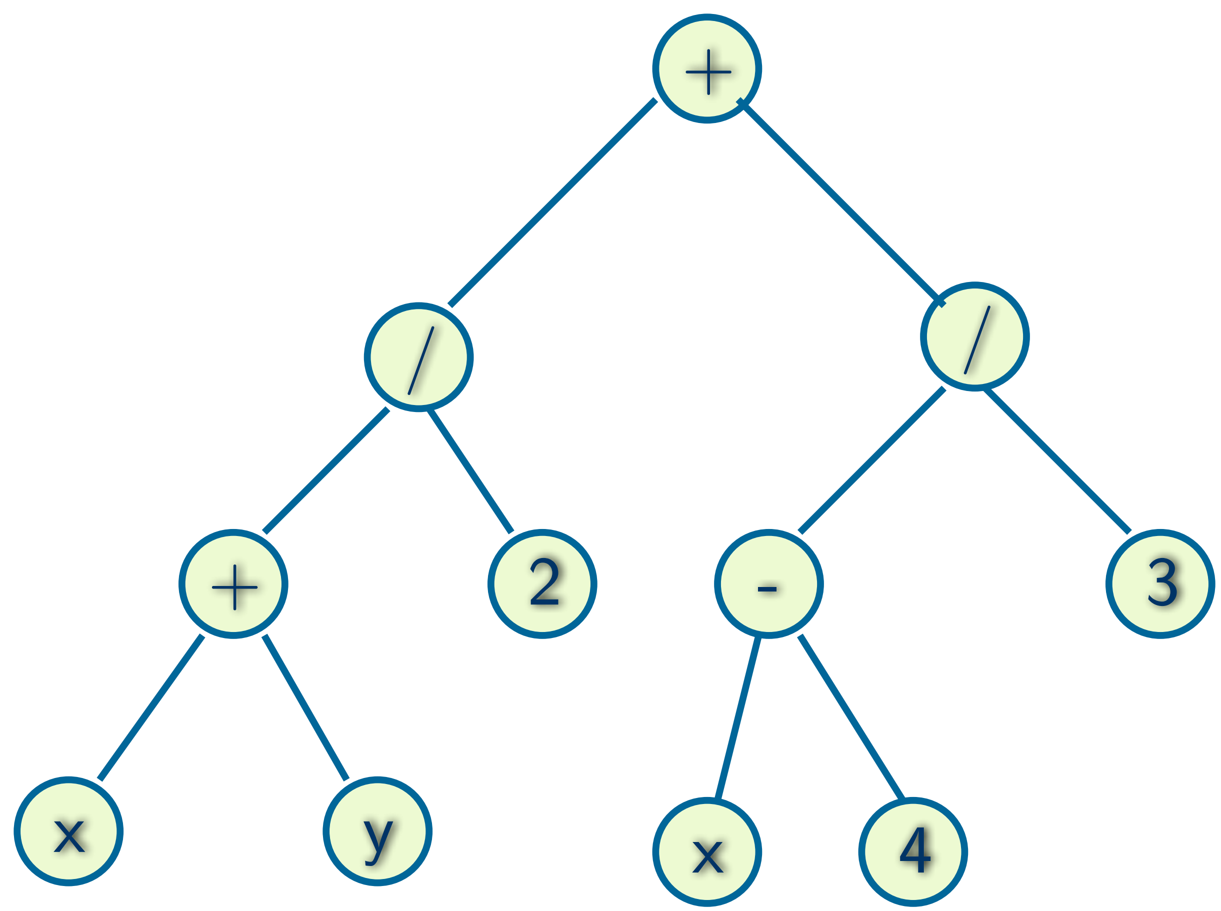
```
Input: A binary tree T
Output: The sequence of nodes from postorder traversal of T

if T_left ≠ ∅ then
    Postorder(T_left)
if T_right ≠ ∅ then
    Postorder(T_right)
Visit root of T
```



Ex: Algebraic Expression

- Consider an expression ((x+y)/2)+((x-4)/3))
- What kind of traversal must be applied to evaluate the expression?



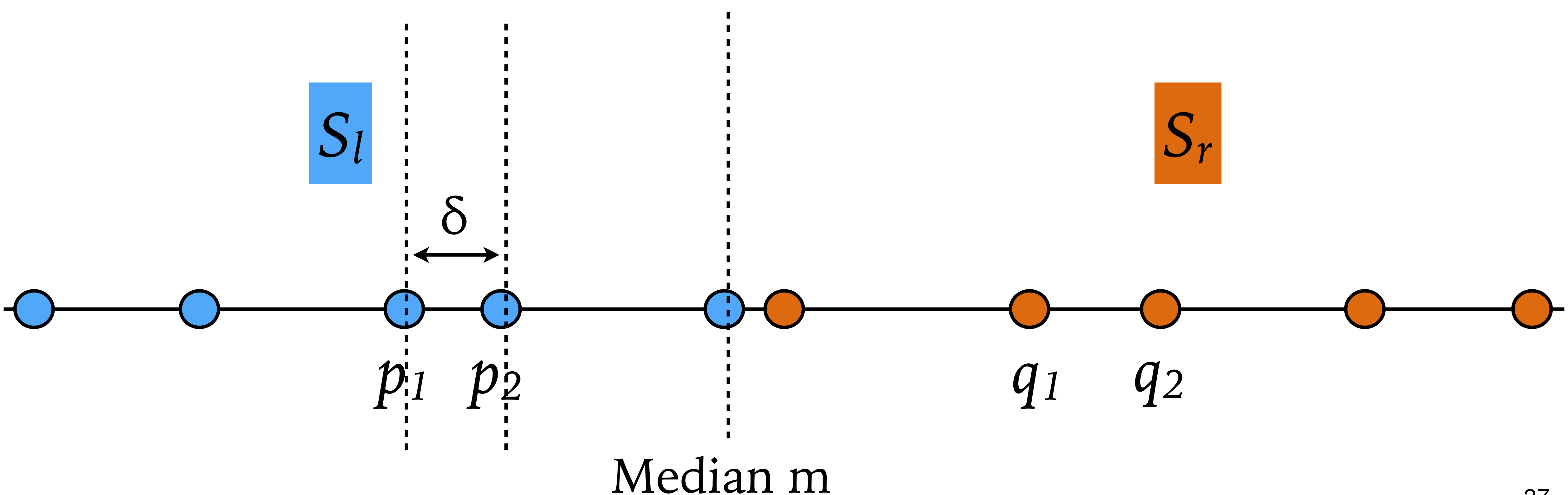
Closest-Pair Problem

ALGORITHM *BruteForceClosestPair*(P)

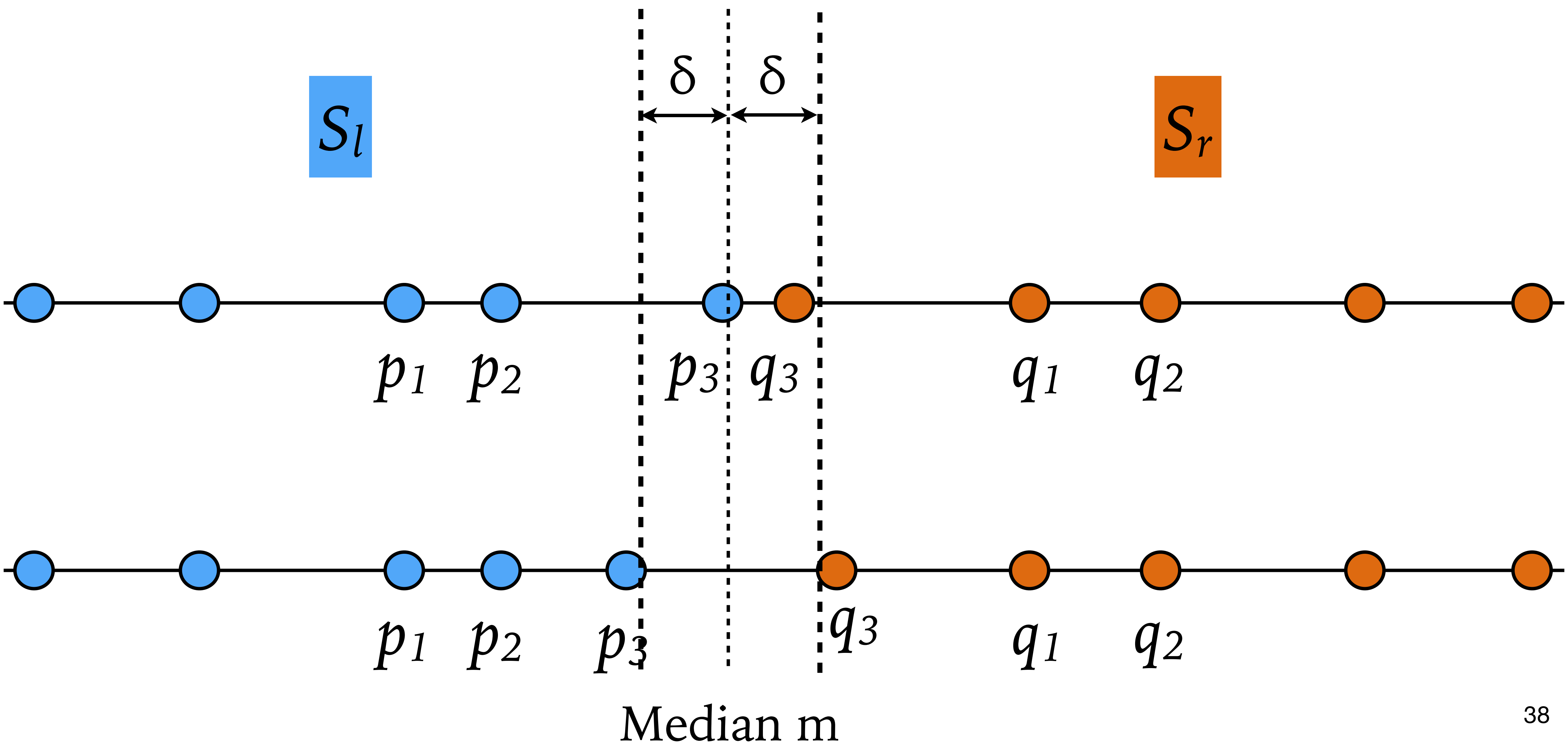
//Finds distance between two closest points in the plane by brute force
//Input: A list P of n ($n \geq 2$) points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points
$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root
return d

1D Closest Pair by Divide-and-

- Divide S into two sets S_l and S_r by the median
- Recursively compute closest pair (p_1,p_2) in S_l and (q_1,q_2) in S_r
- Let $\delta = \min\{dist(p_1,p_2), dist(q_1,q_2)\}$



- Closest pair is (p_1,p_2), or (q_1,q_2), or some (p_3,q_3)
- If (p_3,q_3) is the closest pair, it must be within δ of m
  - In S_1, at most one point can be in (m-δ, m]. Why ?
  - In S_2, at most one point can be in (m, m-δ)



Algorithm 1D-EffClosestPair(S)

Input: Set of sorted points S in 1D

Output: Closest distance $\delta_{\min}$ of two points in S

if $|S| \leq 3$ **return** δ computed by brute force

$m \leftarrow \text{median}(S)$

Split S to S_l and S_r by the median m

$\delta_l \leftarrow \text{1D-EffClosestPair}(S_l)$

$\delta_r \leftarrow \text{1D-EffClosestPair}(S_r)$

$\delta \leftarrow \min(\delta_l, \delta_r)$

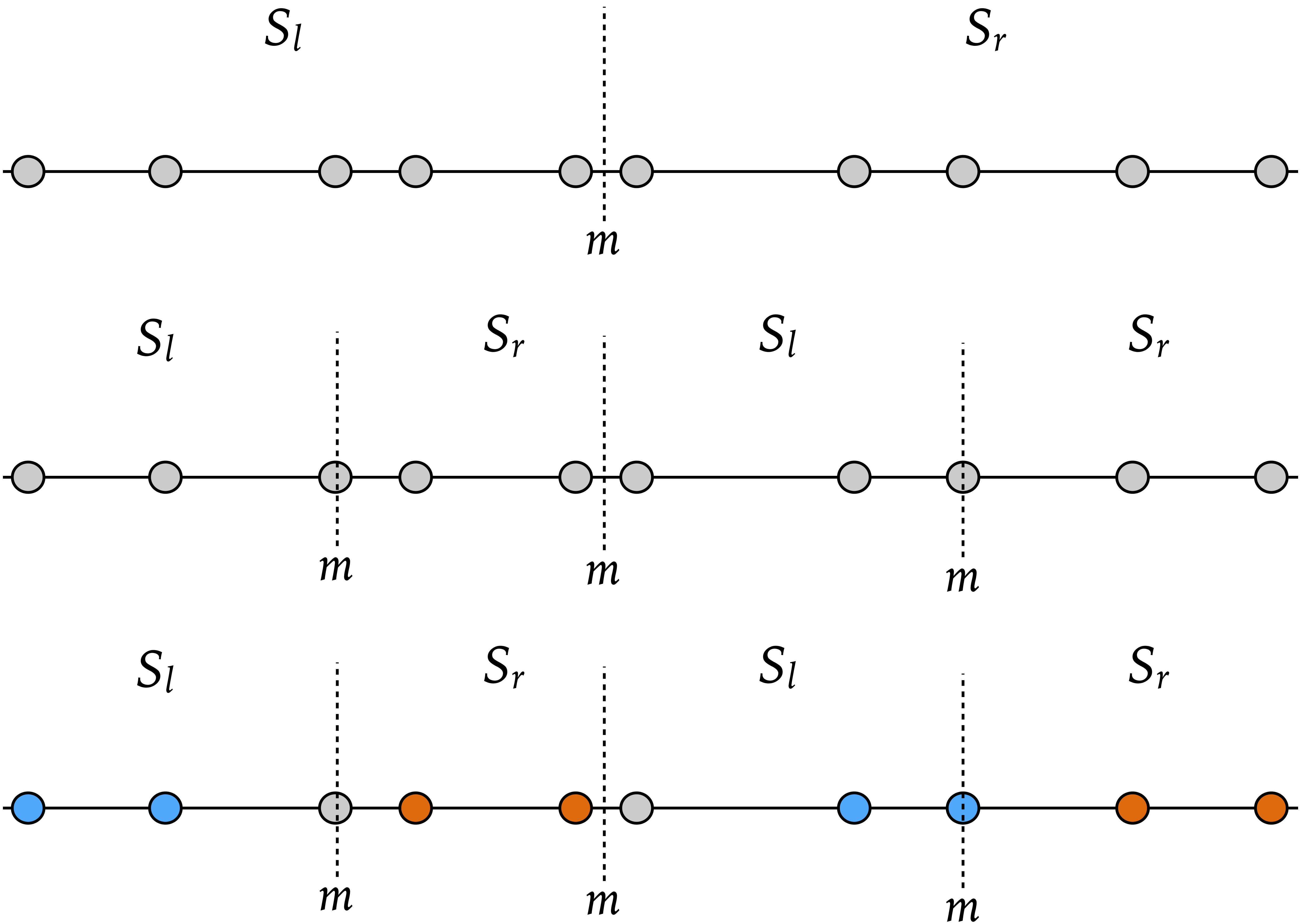
Get a point p_3 in S_l within $m - \delta$ from m

Get a point q_3 in S_r within $m + \delta$ from m

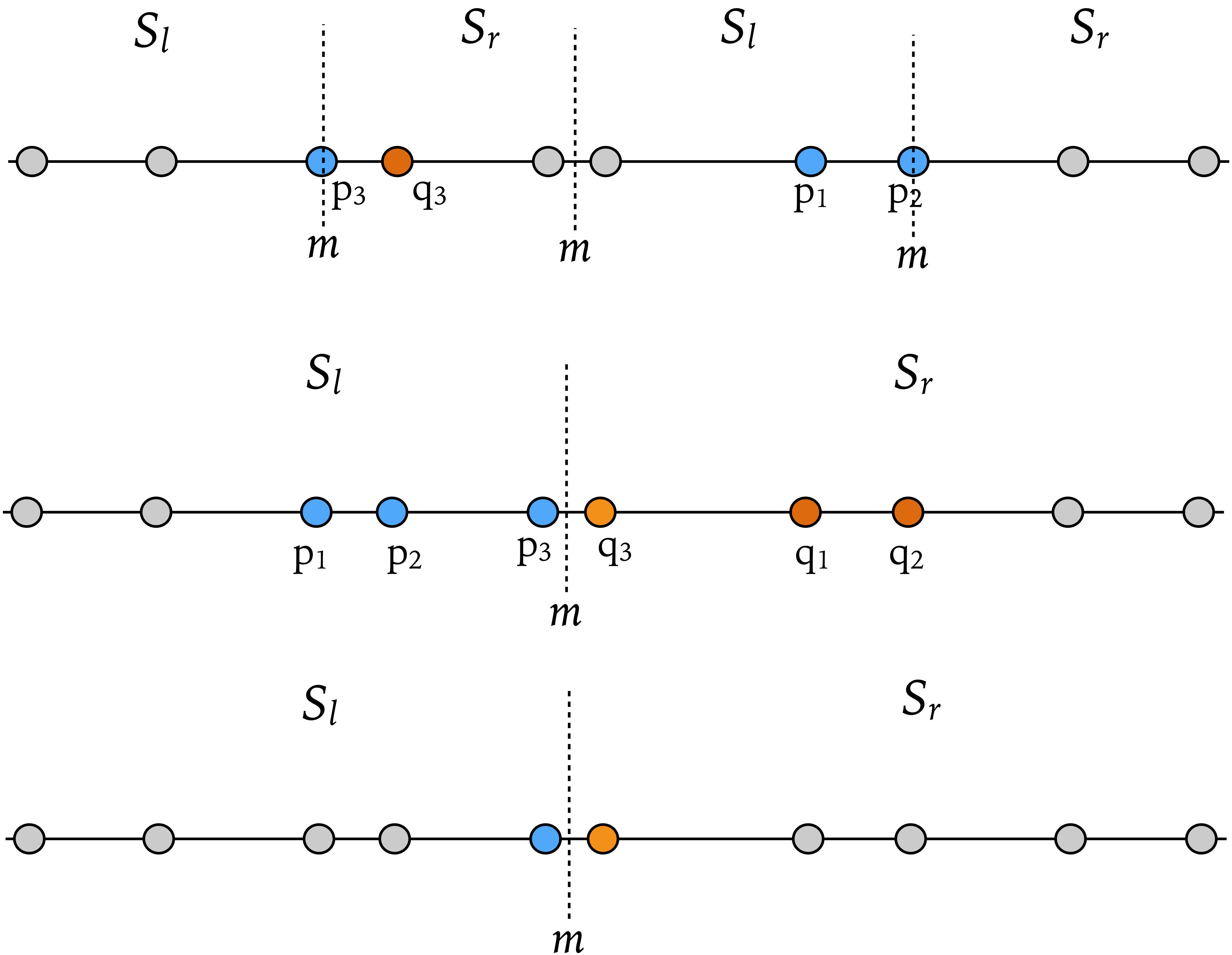
$\delta_{\min} \leftarrow \text{dist}(p_3, q_3)$

$\delta_{\min} \leftarrow \min\{\delta_{\min}, \delta\}$

return $\delta_{\min}$



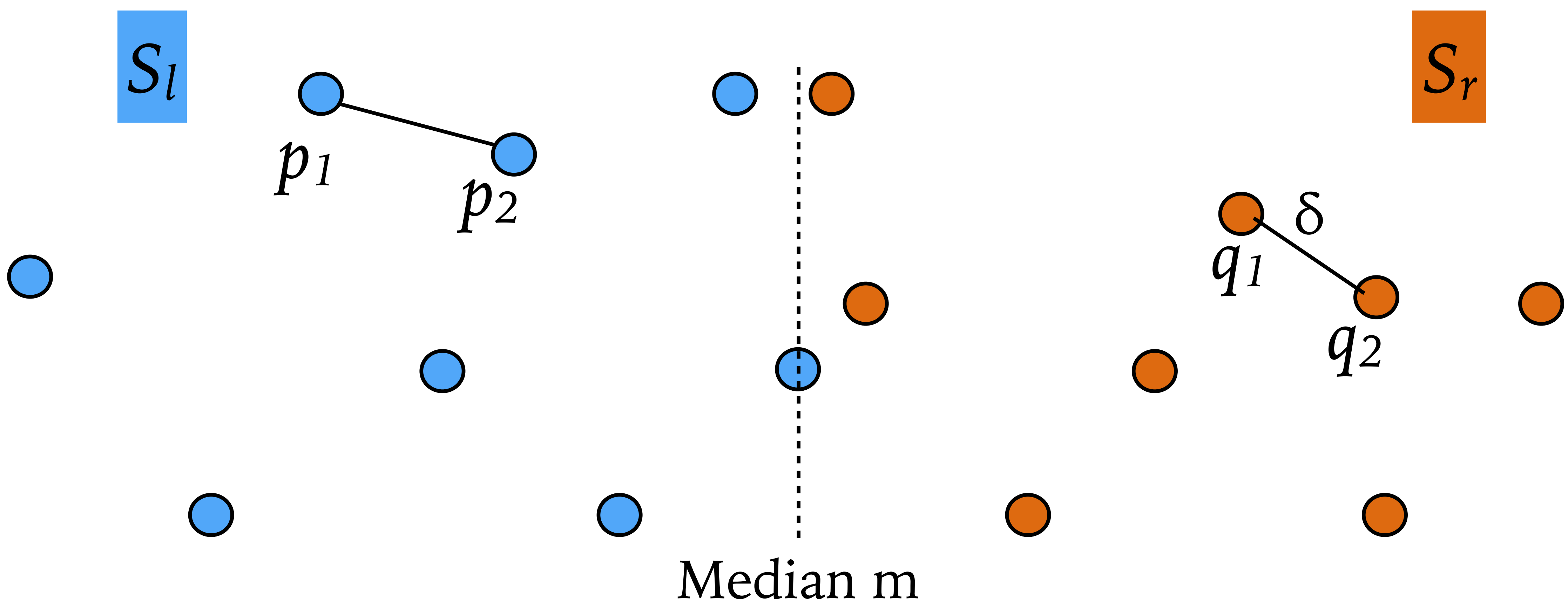
S_l
S_r
m
S_l
S_r
S_l
S_r
m
m
m
S_l
S_r
S_l
S_r
m
m
m



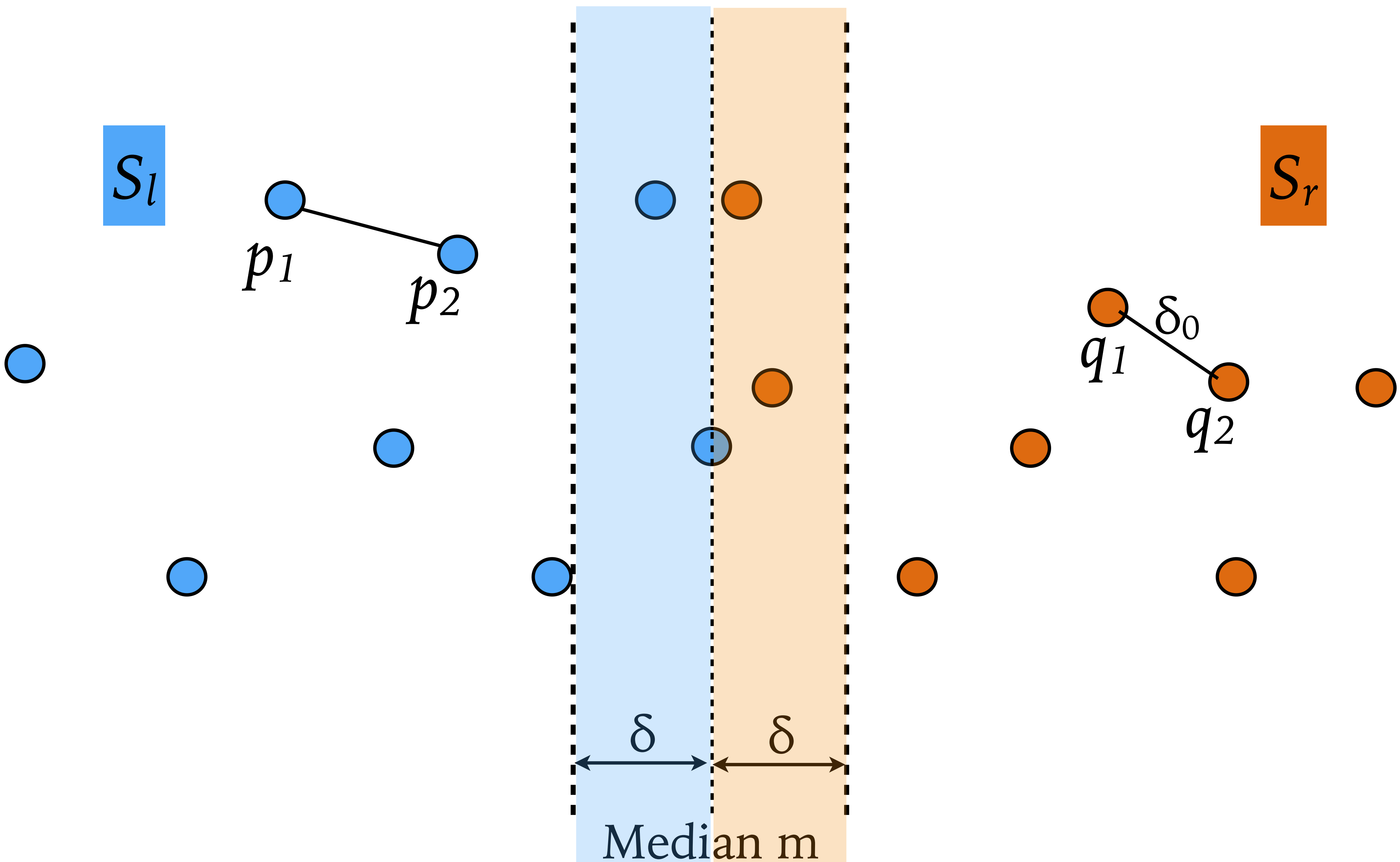
S_l
S_r
S_l
S_r
p_3
q_3
p_1
p_2
m
m
m
S_l
S_r
p_1
p_2
p_3
q_3
q_1
q_2
m
S_l
S_r
m

2D Closest Pair by Divide-and-

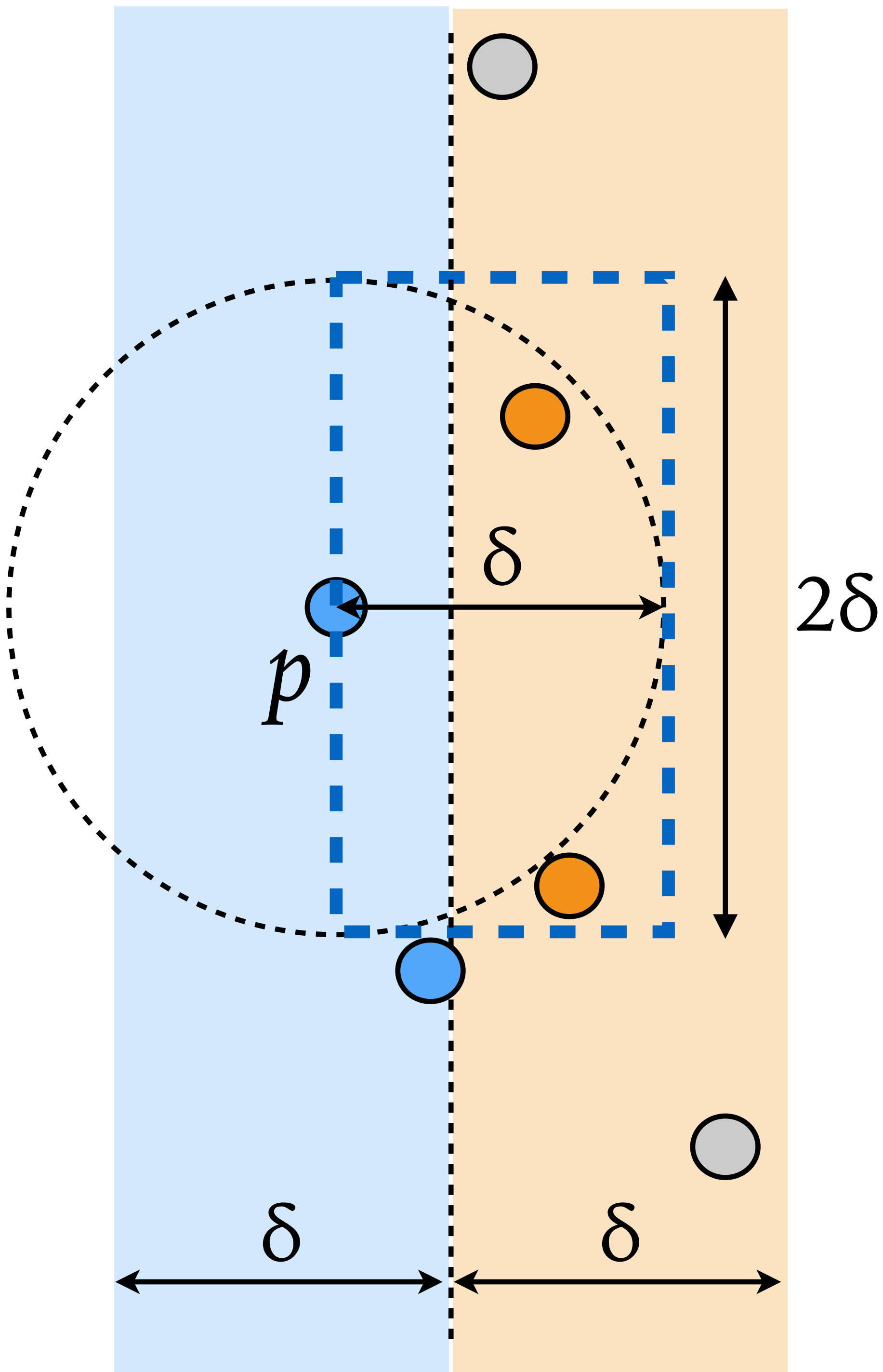
- Divide S into two sets S_l and S_r by the median of x
  - Recursively compute closest pair (p_1,p_2) in S_l and (q_1,q_2) in S_r
  - Let $\delta = \min\{dist(p_1,p_2), dist(q_1,q_2)\}$



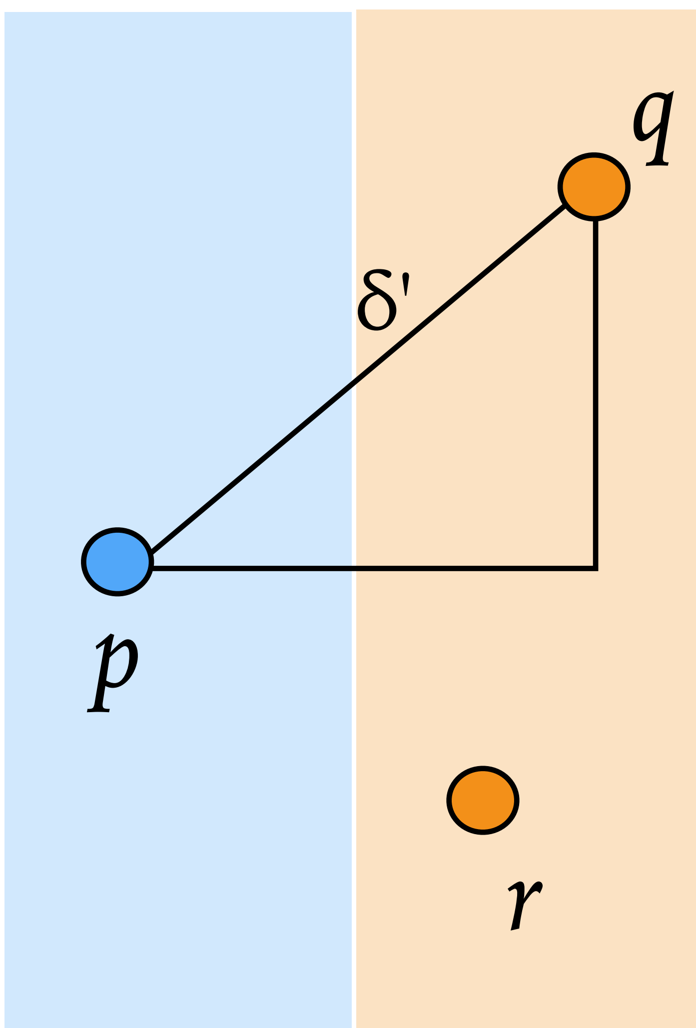
- Find the closest pair from all pairs of points whose x-coordinates within (m-δ, m+δ)

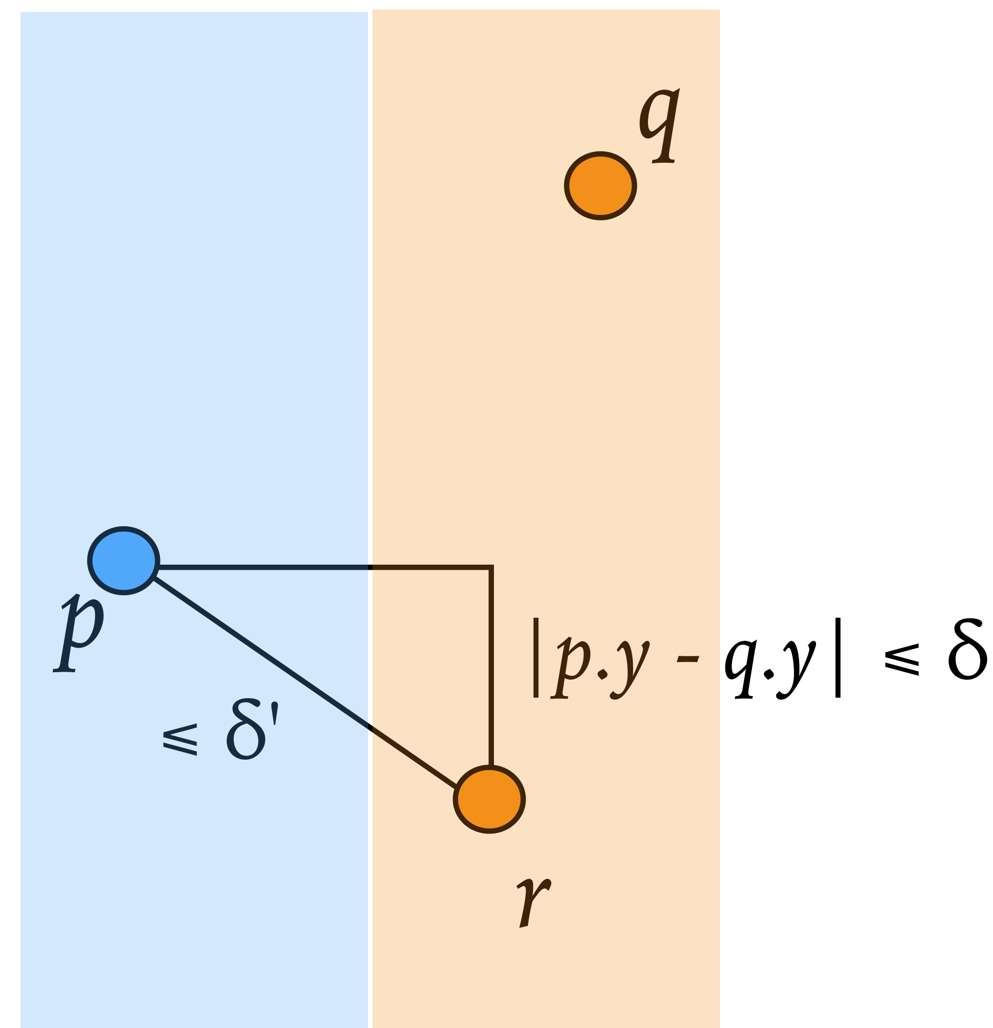


- For each point p in the strip, all neighbors within distance δ_0 must also be in the circle R

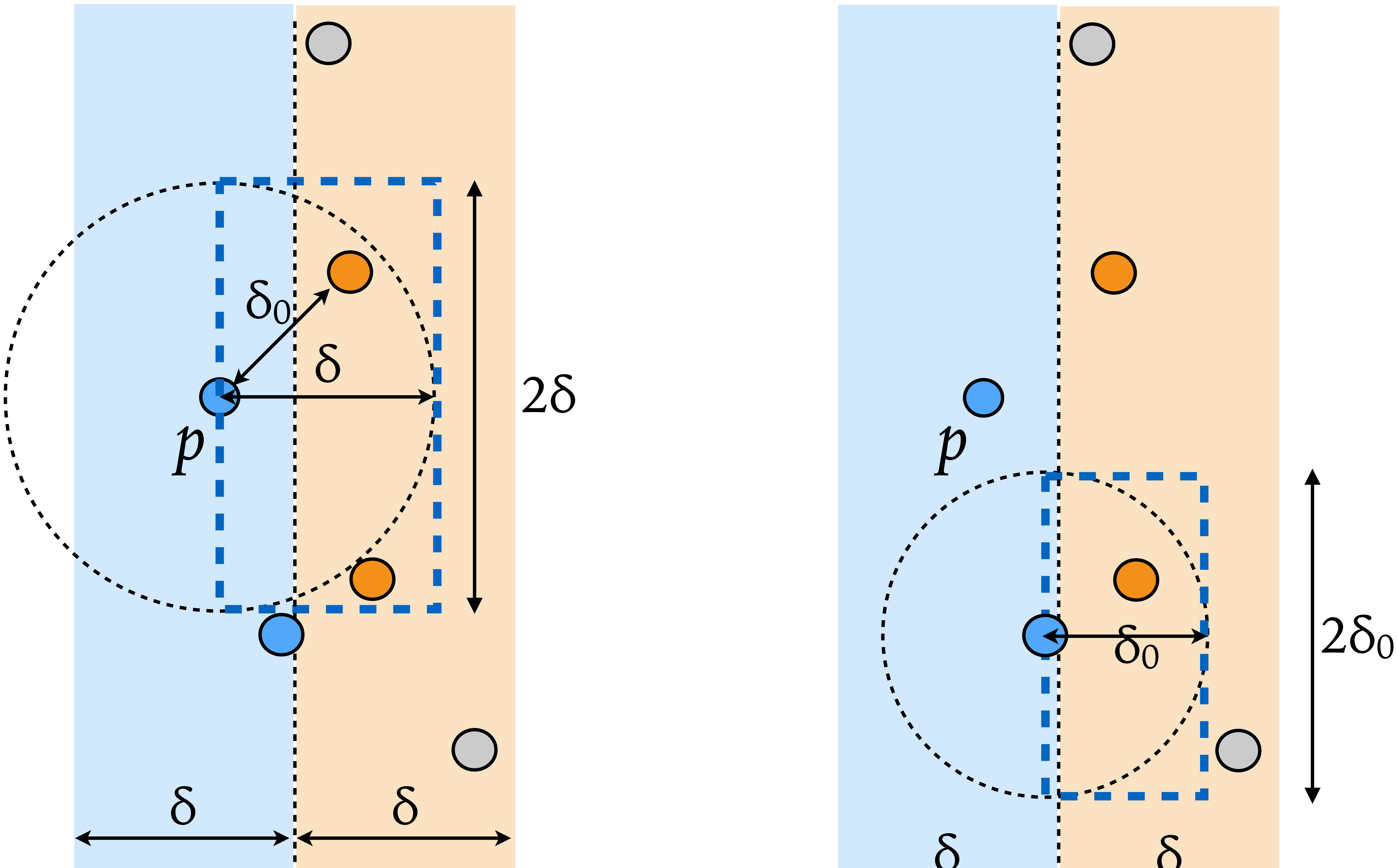


- If dist$(p,q) = \delta'$, the y-coord difference must be less than or equal to δ'.
- So, any other point r with dist$(p,r) \leqslant \delta'$ must have its y-coordinate difference less than δ'





- Can subsequently consider only points with y-coord difference less than δ_0
- Update δ_0 if encoutering a closer point to p.



Algorithm 2D-EffClosestPair(X, Y)

Inputs: Set of points X with size n sorted by x-coordinates
    Same set of points Y sorted by y-coordinates
Output: Distance $\delta_{\min}$ of two closest points

if $n \leq 3$ **return** $\delta_{\min}$ computed by brute force

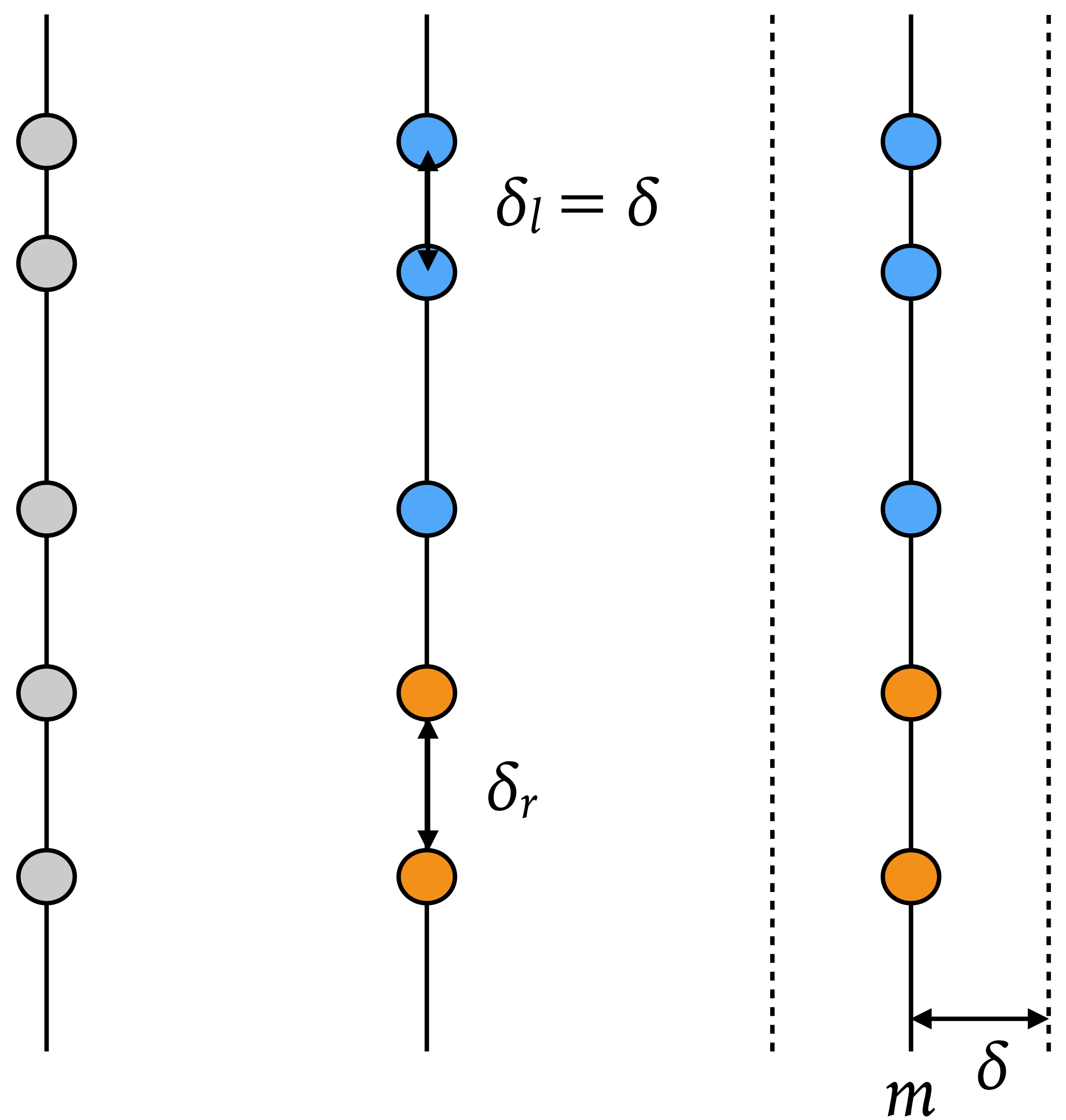
Copy first $\lceil n/2 \rceil$ points in X to X_l
Copy the same points above in Y to Y_l
$\delta_l \leftarrow$ 2D-EffClosestPair(X_l, Y_l)

Copy the remaining $\lfloor n/2 \rfloor$ points in X to X_r
Copy the same points above in Y to Y_r
$\delta_r \leftarrow$ 2D-EffClosestPair(X_r, Y_r)
$\delta \leftarrow \min\{\delta_l, \delta_r\}$

$m \leftarrow X[\lceil n/2 \rceil - 1].x$ ▷ Get the median by x-coordinates
Copy all points in Y whose x-coordinates are within $m - \delta$ to array P
Copy all points in Y whose x-coordinates are within $m + \delta$ to array Q
$\delta_{\min} \leftarrow \delta$
for each point $p \in P$ **do**
    for each point $q \in Q$ with $|p.y - q.y| < \delta_{\min}$ **do**
        $\delta_{\min} \leftarrow \min\{\delta_{\min}, \mathrm{dist}(p, q)\}$

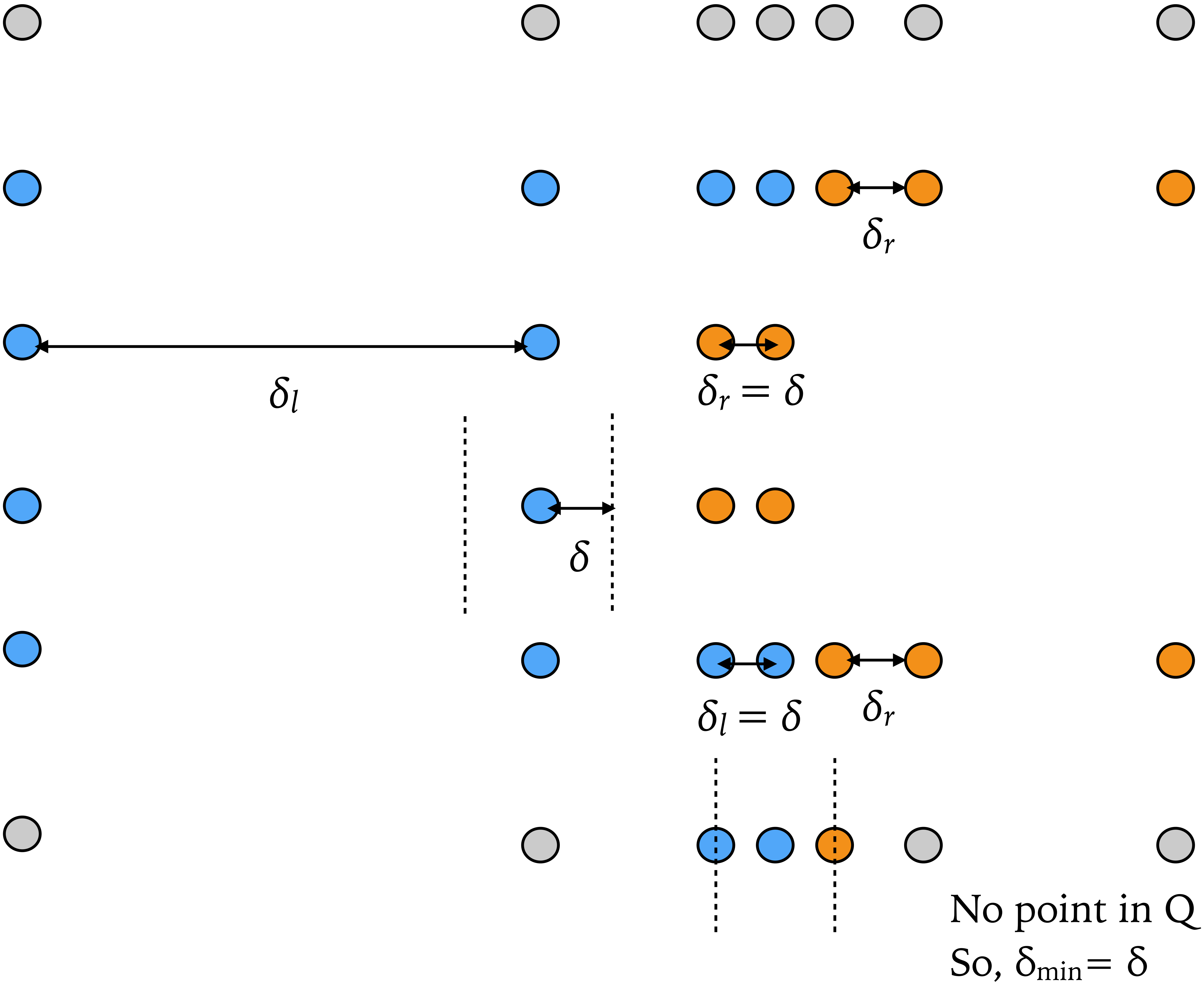
return $\delta_{\min}$

Examples



No points in Q

So, $\delta_{min}=\delta$



δ_r
δ_l
$\delta_r = \delta$
δ
$\delta_l = \delta$
δ_r
No point in Q
So, $\delta_{\min} = \delta$

Summary

- Divide-and-Conquer design
  - Problem recursively divided into equal-sized subproblems
  - Combine solutions of subproblems to get the solution of the original problem.
- Running time typically satisfies $T(n) = aT(n/b) + f(n)$ and solved from Master theorem.