

# CS2040S

## Data Structures and Algorithms

Welcome!

### INEFFECTIVE SORTS (XKCD: 1185)

```
DEFINE HALFHEARTEDMERGESORT(LIST):  
  IF LENGTH(LIST) < 2:  
    RETURN LIST  
  PIVOT = INT(LENGTH(LIST) / 2)  
  A = HALFHEARTEDMERGESORT(LIST[:PIVOT])  
  B = HALFHEARTEDMERGESORT(LIST[PIVOT:])  
  // UMMMMMM  
  RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):  
  // AN OPTIMIZED BOGOSORT  
  // RUNS IN  $O(N \log N)$   
  FOR N FROM 1 TO LOG(LENGTH(LIST)):  
    SHUFFLE(LIST):  
    IF ISSORTED(LIST):  
      RETURN LIST  
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBIINTERVIEWQUICKSORT(LIST):  
  OK SO YOU CHOOSE A PIVOT  
  THEN DIVIDE THE LIST IN HALF  
  FOR EACH HALF:  
    CHECK TO SEE IF IT'S SORTED  
    NO, WAIT, IT DOESN'T MATTER  
    COMPARE EACH ELEMENT TO THE PIVOT  
    THE BIGGER ONES GO IN A NEW LIST  
    THE EQUAL ONES GO INTO, UH  
    THE SECOND LIST FROM BEFORE  
  HANG ON, LET ME NAME THE LISTS  
  THIS IS LIST A  
  THE NEW ONE IS LIST B  
  PUT THE BIG ONES INTO LIST B  
  NOW TAKE THE SECOND LIST  
  CALL IT LIST, UH, A2  
  WHICH ONE WAS THE PIVOT IN?  
  SCRATCH ALL THAT  
  IT JUST RECURSIVELY CALLS ITSELF  
  UNTIL BOTH LISTS ARE EMPTY  
  RIGHT?  
  NOT EMPTY, BUT YOU KNOW WHAT I MEAN  
  AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):  
  IF ISSORTED(LIST):  
    RETURN LIST  
  FOR N FROM 1 TO 10000:  
    PIVOT = RANDOM(0, LENGTH(LIST))  
    LIST = LIST[PIVOT:] + LIST[:PIVOT]  
    IF ISSORTED(LIST):  
      RETURN LIST  
  IF ISSORTED(LIST):  
    RETURN LIST  
  IF ISSORTED(LIST): // THIS CAN'T BE HAPPENING  
    RETURN LIST  
  IF ISSORTED(LIST): // COME ON COME ON  
    RETURN LIST  
  // OH JEEZ  
  // I'M GONNA BE IN SO MUCH TROUBLE  
  LIST = []  
  SYSTEM("SHUTDOWN -H +5")  
  SYSTEM("RM -RF .")  
  SYSTEM("RM -RF ~/*")  
  SYSTEM("RM -RF /")  
  SYSTEM("RD /S /Q C:\*") // PORTABILITY  
  RETURN [1, 2, 3, 4, 5]
```

# Catch the Spies

---

There are  $N$  students in CS2040S and  $K$  of them are spies. Your job is to identify all the spies.



398 Participants

100 Correct Submissions

## CS2040S Catch the Spies Hall of Fame (Fastest)

Tee Weile Wayne	4211
Neo Wei Qing	4288
Peh Kai Min, Ryan	5136
Elizabeth Chow Ting San	5180
Tan Kel Zin	5388
Chen Yanyu	5684
Cui Langyuan	5685
Ryan Chung Yi Sheng	5928
Dai Tianle	5929
Soo Wei Kang Kelvin	6073

## CS2040S Catch the Spies Hall of Fame (Cheapest)

Tee Weile Wayne	2017105
Wong Pei Xian	2027508
Elizabeth Chow Ting San	2094285
Neo Wei Qing	2640851
Peh Kai Min, Ryan	3152289
Dai Tianle	3219550
Peh Hoe Khim Marcus	3231275
Lin Fangyuan	3243523
Tan Kel Zin	3256308
Chen Yanyu	3559568

# Sorting, Part I

---

## Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

## Properties

- Running time
- Space usage
- Stability

# Sorting, Part II

---

## QuickSort

- Divide-and-Conquer
- Partitioning

# QuickSort

---

**QuickSort**(A[1..n], n)

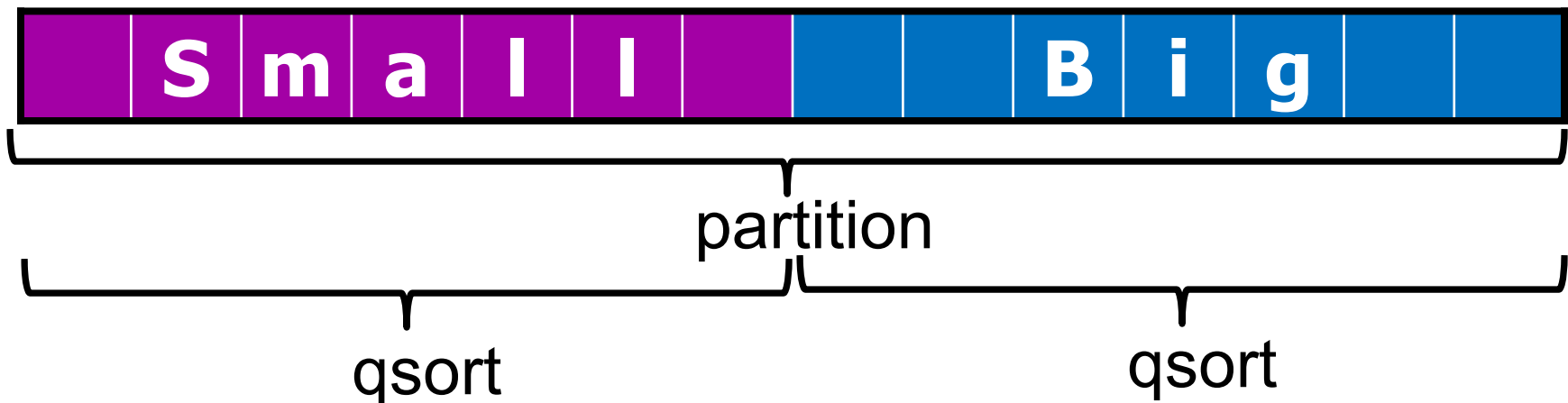
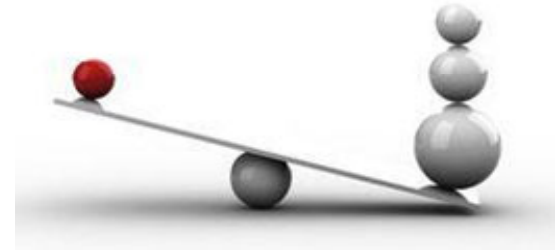
**if** (n==1) **then** return;

**else**

p = **partition**(A[1..n], n)

x = **QuickSort**(A[1..p-1], p-1)

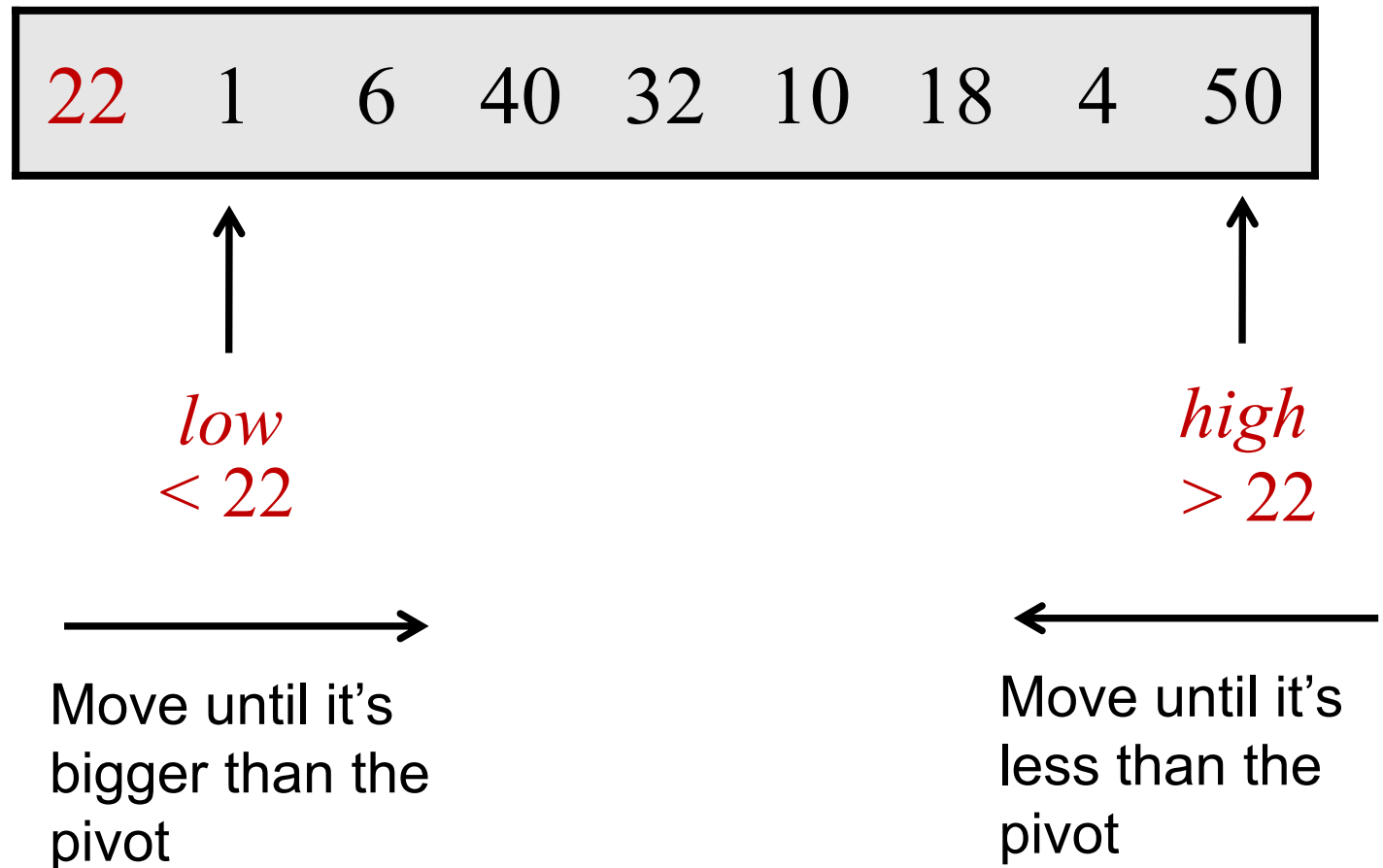
y = **QuickSort**(A[p+1..n], n-p)



# Partitioning an Array “in-place”

---

Example: partition around 22

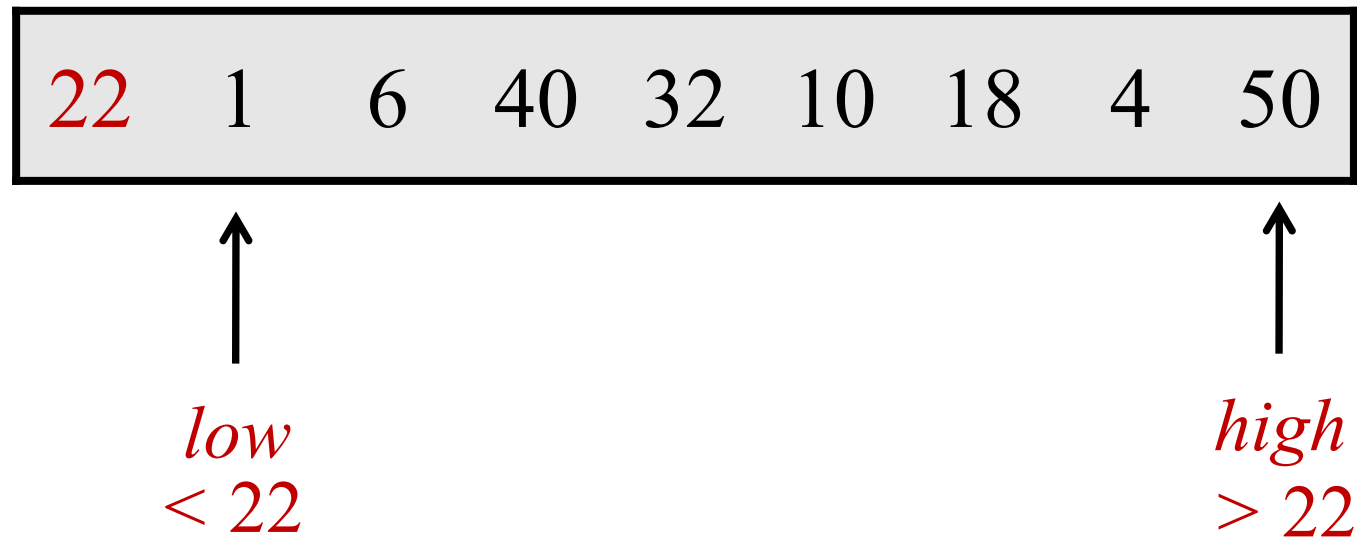




# Partitioning an Array

---

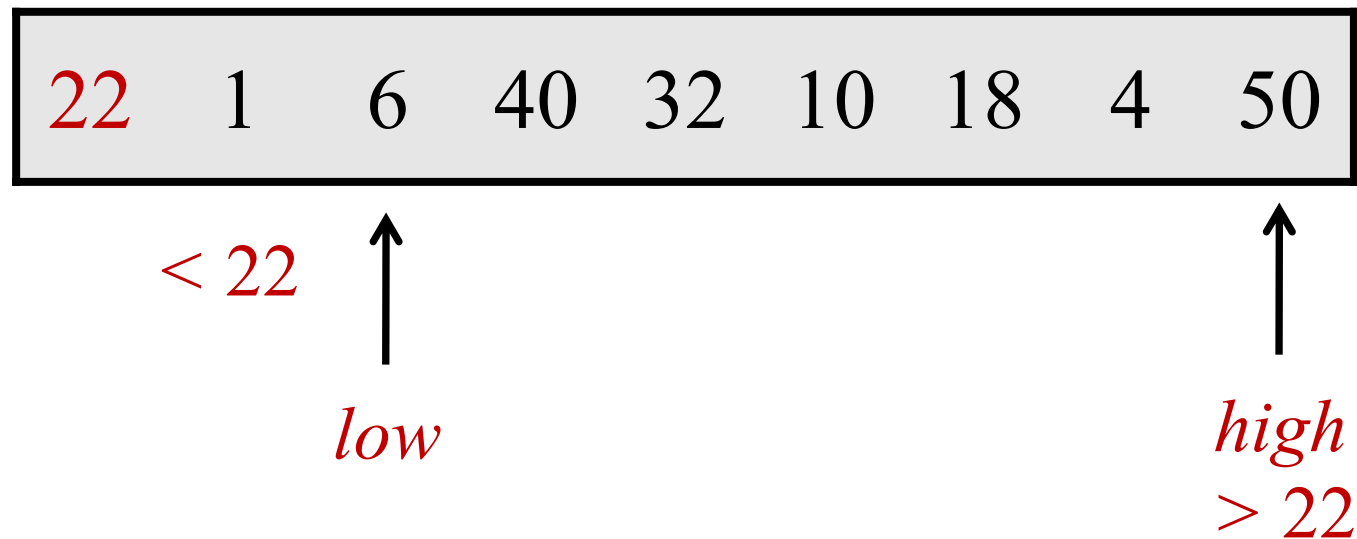
Example: partition around 22



# Partitioning an Array

---

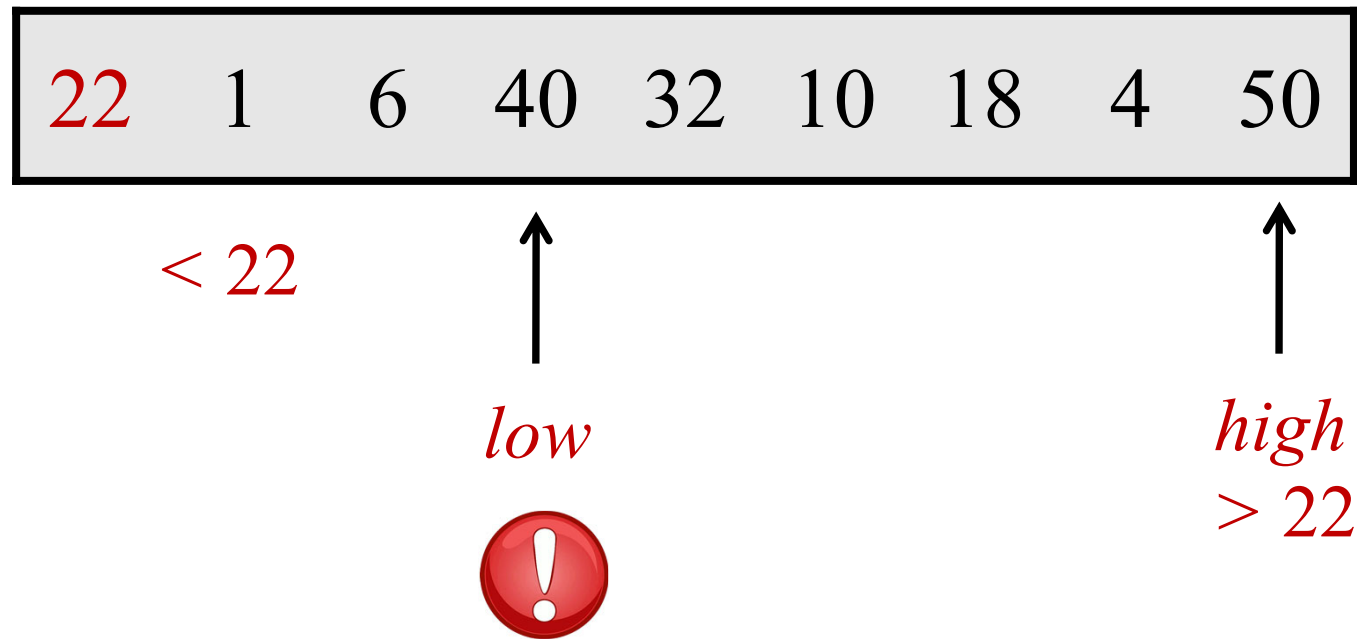
Example: partition around 22



# Partitioning an Array

---

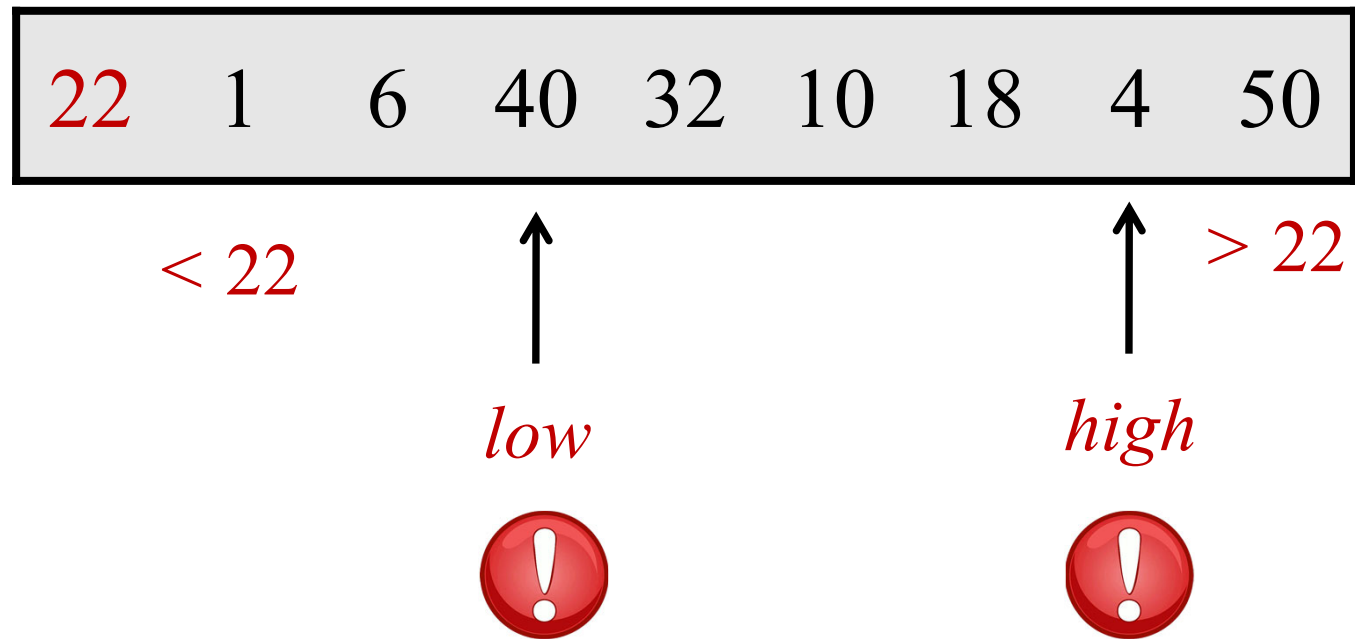
Example: partition around 22



# Partitioning an Array

---

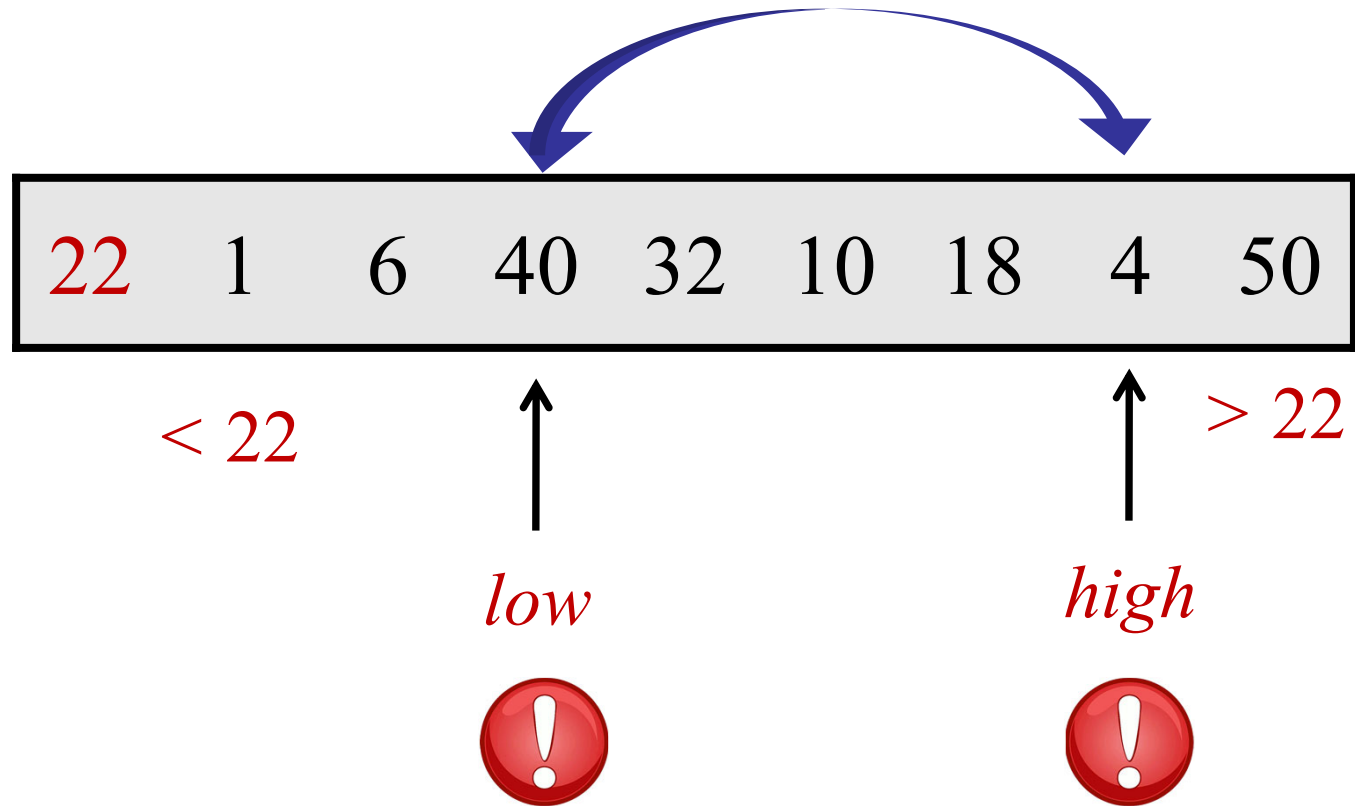
Example: partition around 22



# Partitioning an Array

---

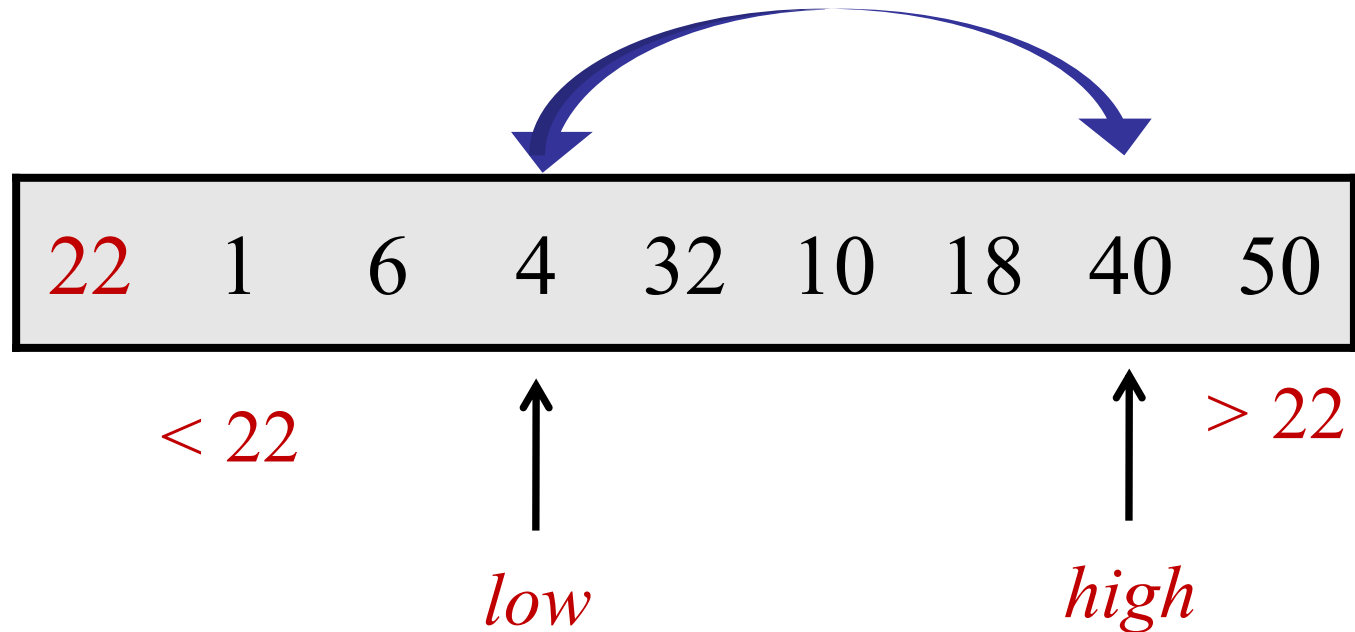
Example: partition around 22



# Partitioning an Array

---

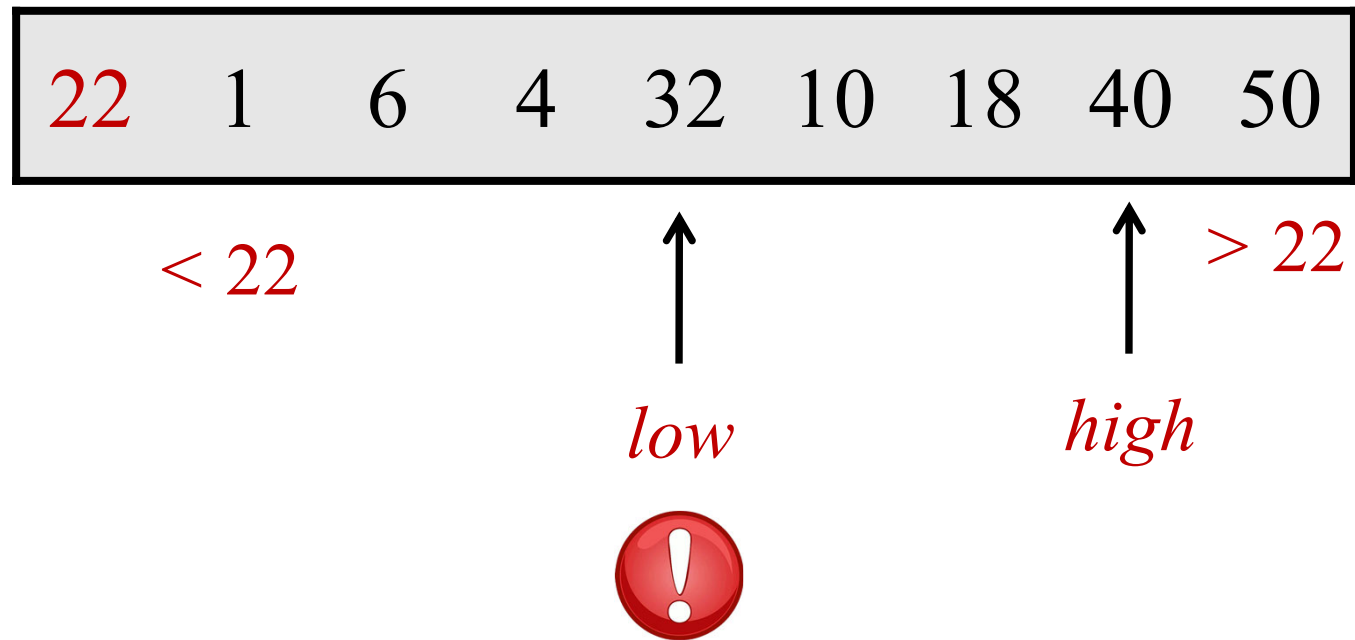
Example: partition around 22



# Partitioning an Array

---

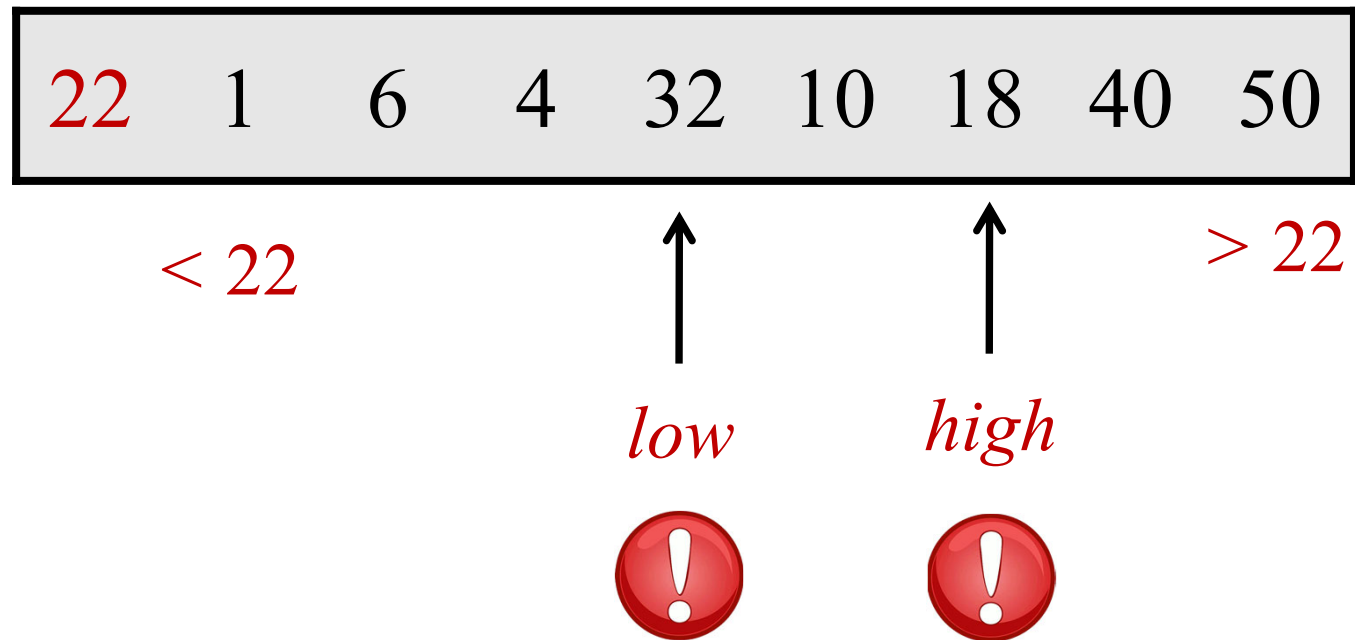
Example: partition around 22



# Partitioning an Array

---

Example: partition around 22

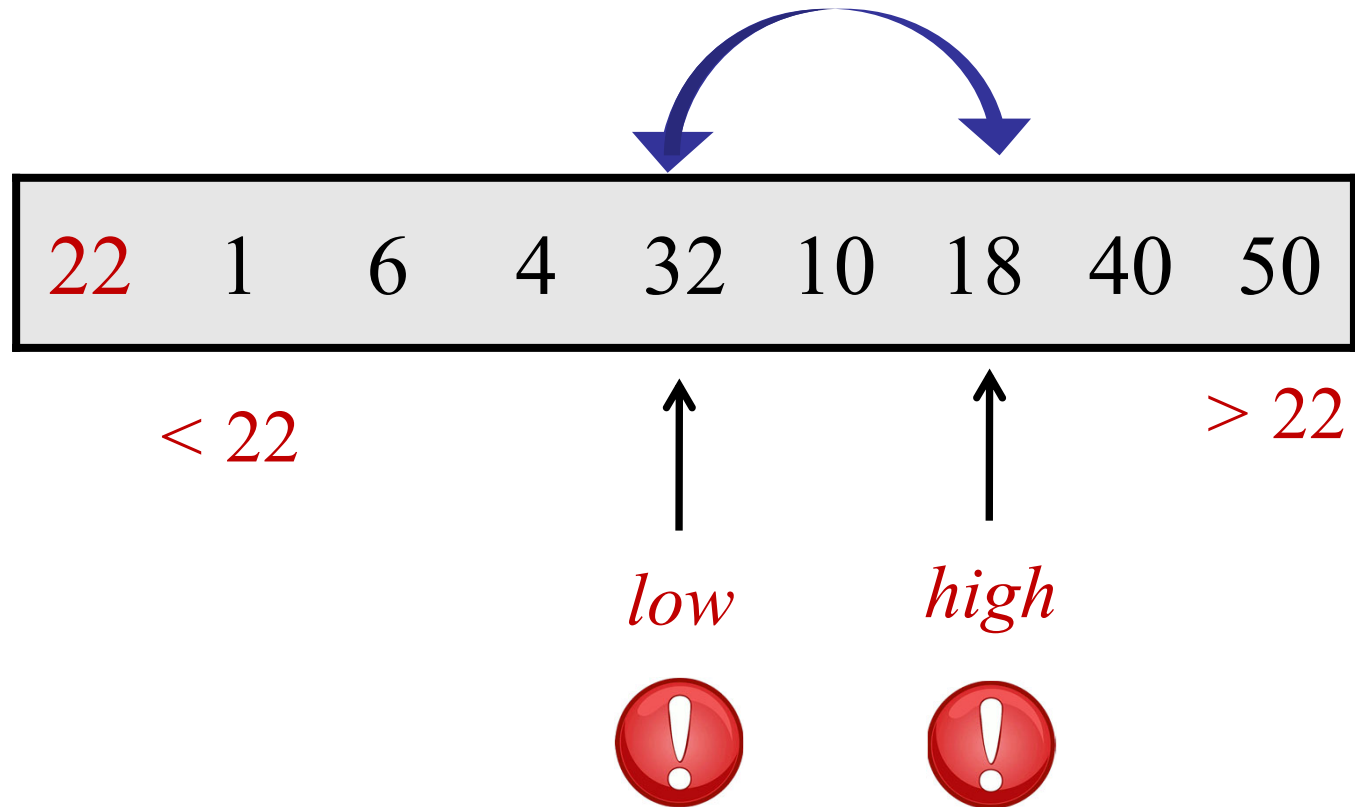




# Partitioning an Array

---

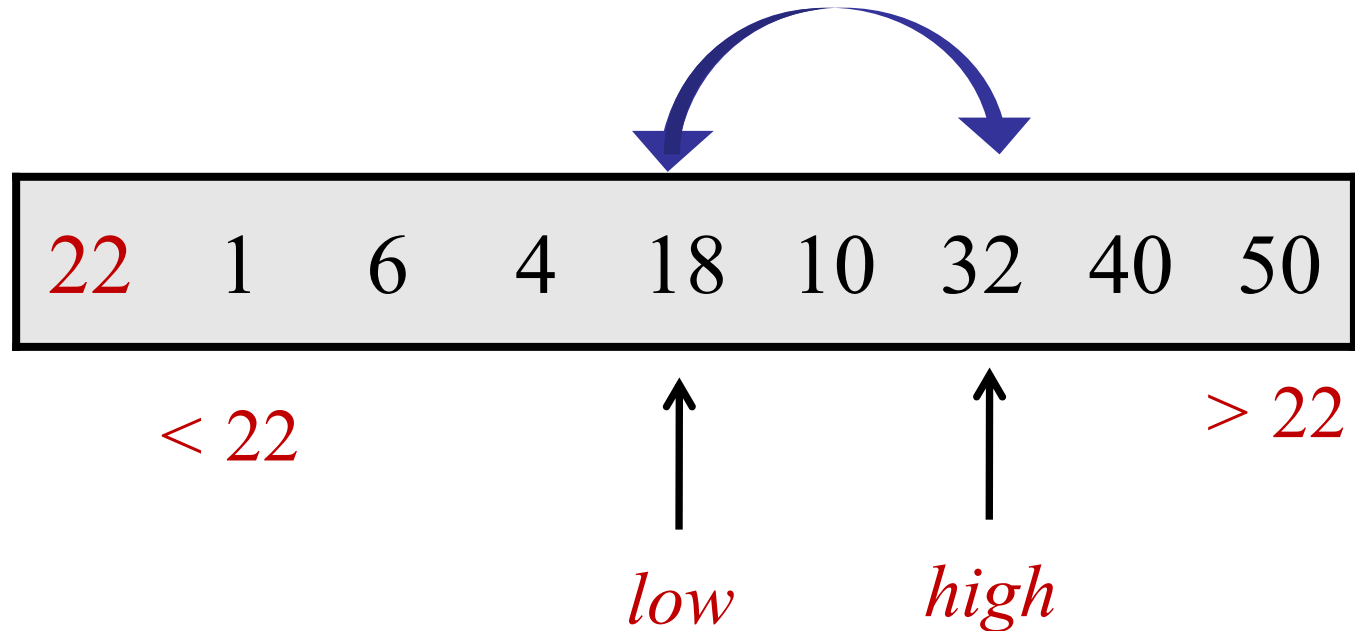
Example: partition around 22



# Partitioning an Array

---

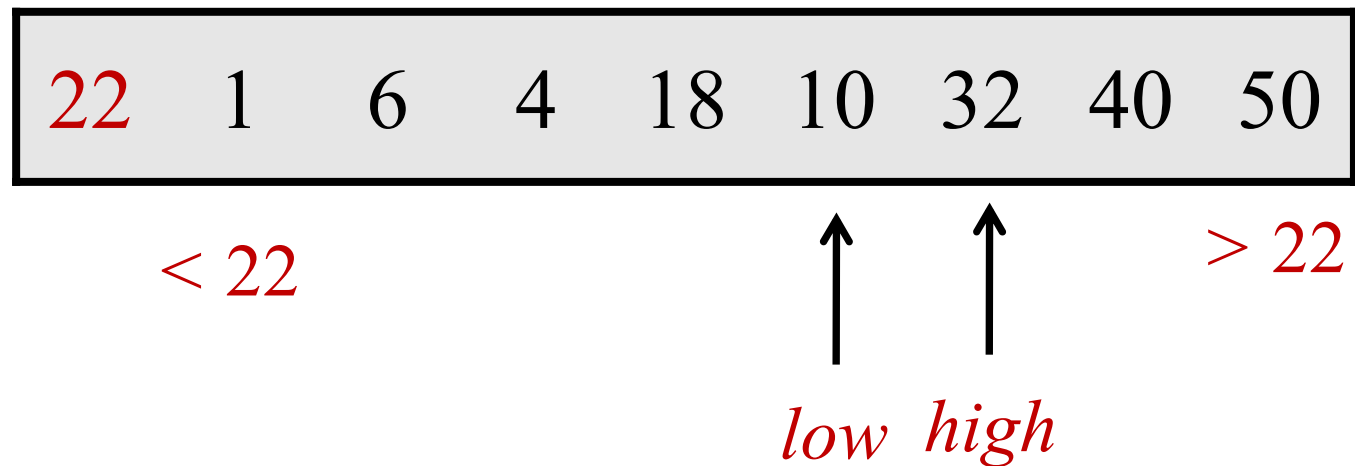
Example: partition around 22



# Partitioning an Array

---

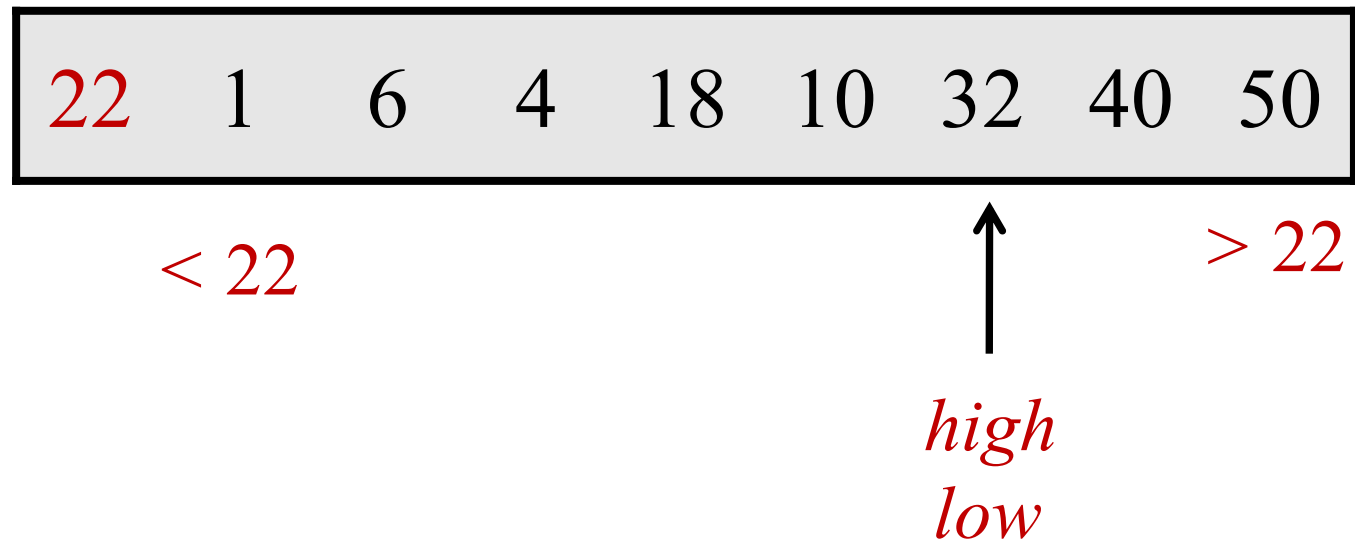
Example: partition around 22



# Partitioning an Array

---

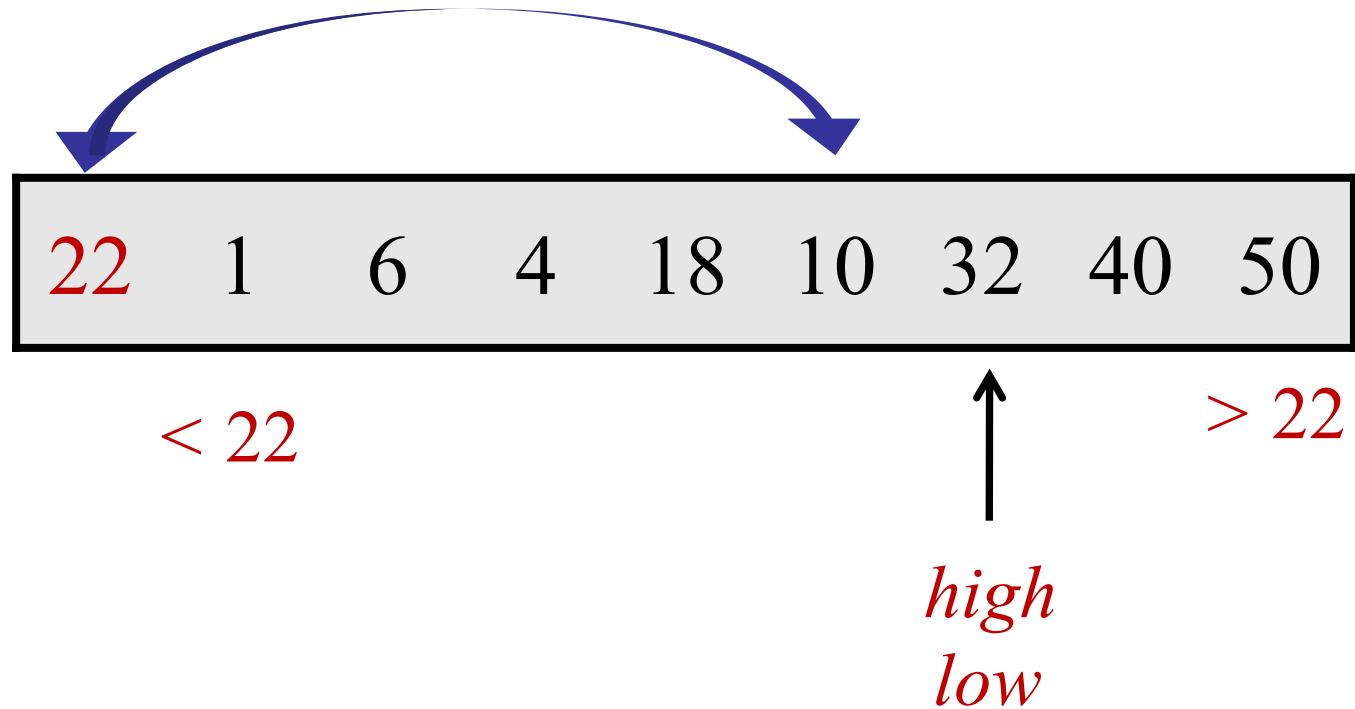
Example: partition around 22



# Partitioning an Array

---

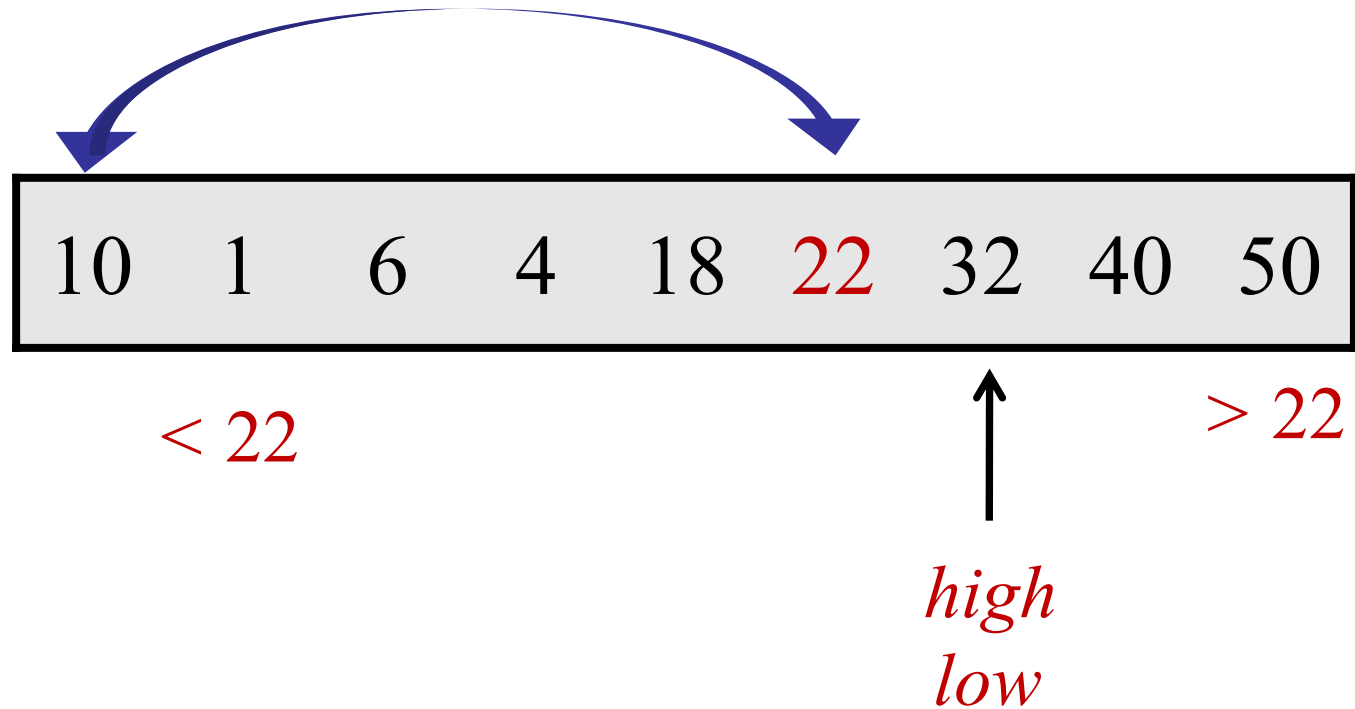
Example: partition around 22



# Partitioning an Array

---

Example: partition around 22



# Today: Sorting, Part III

---

## QuickSort

- Divide-and-Conquer
- Partitioning
- Duplicates
- Choosing a pivot
- Randomization
- Analysis

# QuickSort

---

What happens if there are duplicates?

**ARCHIPELAGO**

is open



# Duplicates

---

**QuickSort**( $A[1..n]$ ,  $n$ )

**if** ( $n==1$ ) **then** return;

**else**

Choose pivot index  $pIndex$ .

$p = \text{partition}(A[1..n], n, pIndex)$

$x = \text{QuickSort}(A[1..p-1], p-1)$

$y = \text{QuickSort}(A[p+1..n], n-p)$

$< x$

$x \quad x \quad x$

$> x$

# Quicksort

---

Example:

6 6 6 6 6 6

# Quicksort

---

Example:

6	6	6	6	6	6
6	6	6	6	6	6

# Quicksort

---

Example:

6	6	6	6	6	6
6	6	6	6	6	6
6	6	6	6	6	6

# Quicksort

---

Example:

6	6	6	6	6	6
6	6	6	6	6	6
6	6	6	6	6	6
6	6	6	6	6	6

## Example:

6 6 6 6 6 6

6 6 6 6 6 6

6 6 6 6 6 6

Diagram illustrating the concept of six using two groups of objects. The first group, enclosed in a rounded rectangle, contains six grey '6's, with the top-left one highlighted in red. The second group contains six red '6's.

## Example:

6 6 6 6 6 6

6 6 6 6 6 6

6 6 6 6 6 6

6 6 6 6 6 6

The diagram consists of two parts. On the left, a rounded rectangle contains a 2x2 grid of the number 6. The top-left 6 is red, the top-right 6 is grey, the bottom-left 6 is grey, and the bottom-right 6 is red. To the right of this rectangle is a 2x4 grid of the number 6, where all eight 6s are red.

## Example:

[illegible]



# Quicksort

---

What is the running time on the all 6's array?

6	6	6	6	6	6
6	6	6	6	6	6
6	6	6	6	6	6
6	6	6	6	6	6
6	6	6	6	6	6
6	6	6	6	6	6



Running  
time:  
 $O(n^2)$

[illegible]

<b>partition</b> ( $A[1..n]$ , $n$ , $pIndex$ )	// Assume no duplicates, $n > 1$
$pivot = A[pIndex]$ ;	// $pIndex$ is the index of pivot
<b>swap</b> ( $A[1]$ , $A[pIndex]$ );	// store pivot in $A[1]$
$low = 2$ ;	// start after pivot in $A[1]$
$high = n + 1$ ;	// Define: $A[n+1] = \infty$
<b>while</b> ( $low < high$ )	
<b>while</b> ( $A[low] < pivot$ ) <b>and</b> ( $low < high$ ) <b>do</b> $low++$ ;	
<b>while</b> ( $A[high] > pivot$ ) <b>and</b> ( $low < high$ ) <b>do</b> $high--$ ;	
<b>if</b> ( $low < high$ ) <b>then</b> <b>swap</b> ( $A[low]$ , $A[high]$ );	
<b>swap</b> ( $A[1]$ , $A[low-1]$ );	
<b>return</b> $low-1$ ;	

# Duplicates

---

**QuickSort**( $A[1..n]$ ,  $n$ )

**if** ( $n==1$ ) **then** return;

**else**

Choose pivot index  $pIndex$ .

$p = \text{partition}(A[1..n], n, pIndex)$

$x = \text{QuickSort}(A[1..p-1], p-1)$

$y = \text{QuickSort}(A[p+1..n], n-p)$



# Duplicates

---

**QuickSort**( $A[1..n]$ ,  $n$ )

**if** ( $n==1$ ) **then** return;

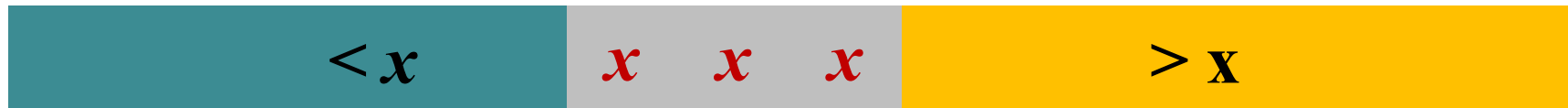
**else**

Choose pivot index  $pIndex$ .

$p = \text{partition}(A[1..n], n, pIndex)$

$x = \text{QuickSort}(A[1..p-1], p-1)$

$y = \text{QuickSort}(A[p+1..n], n-p)$



Pivot

# Duplicates

---

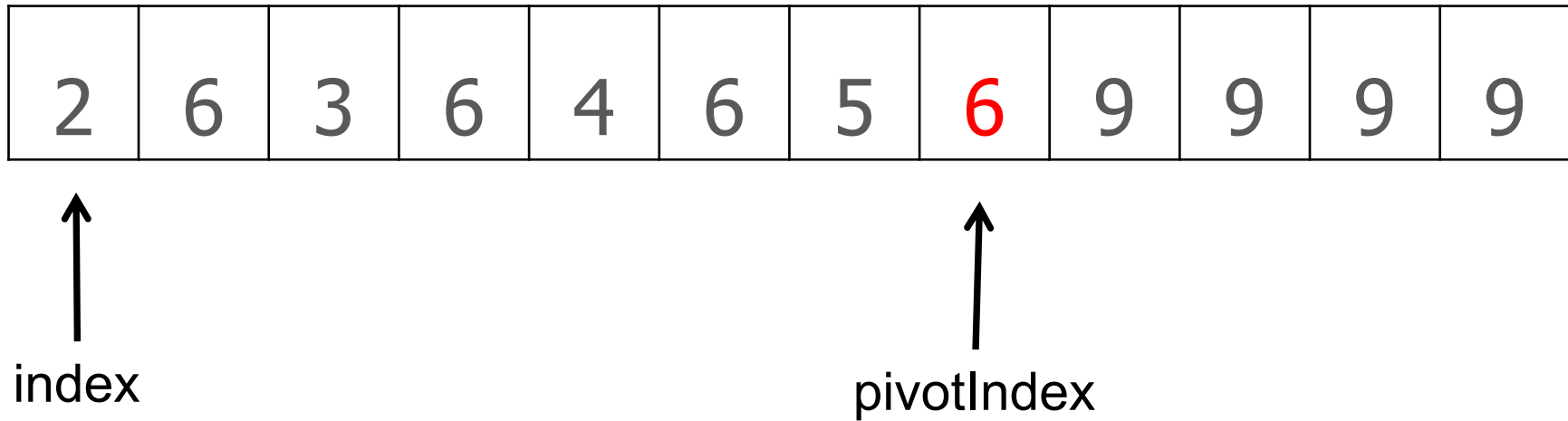
## 3-Way Partitioning

- Option 1: two pass partitioning
  1. Regular partition.
  2. Pack duplicates.

# Pack Duplicates

---

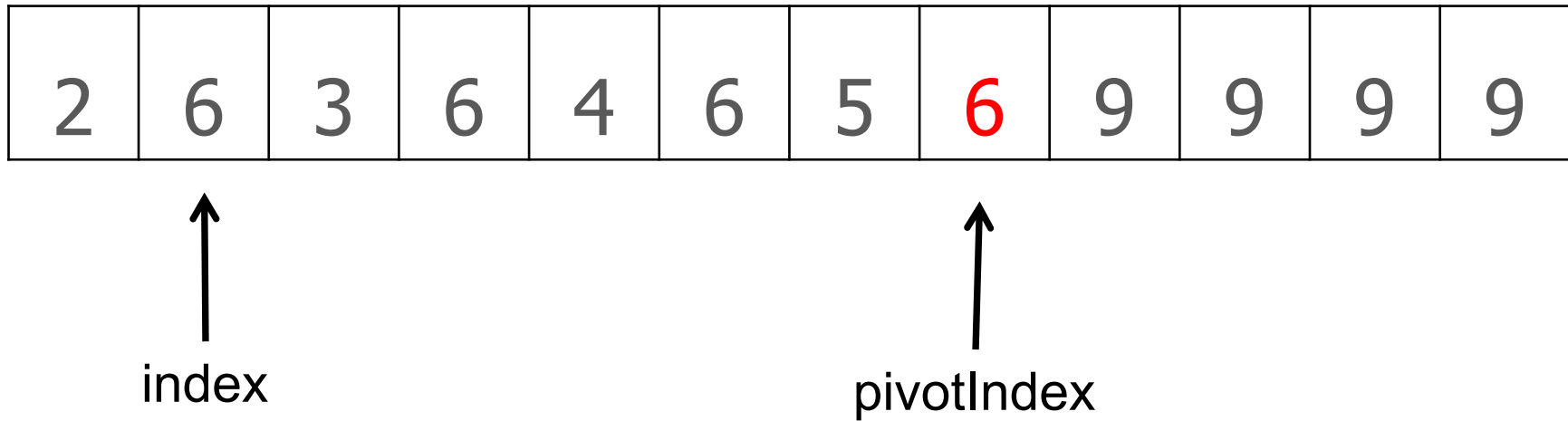
Example:



# Pack Duplicates

---

Example:

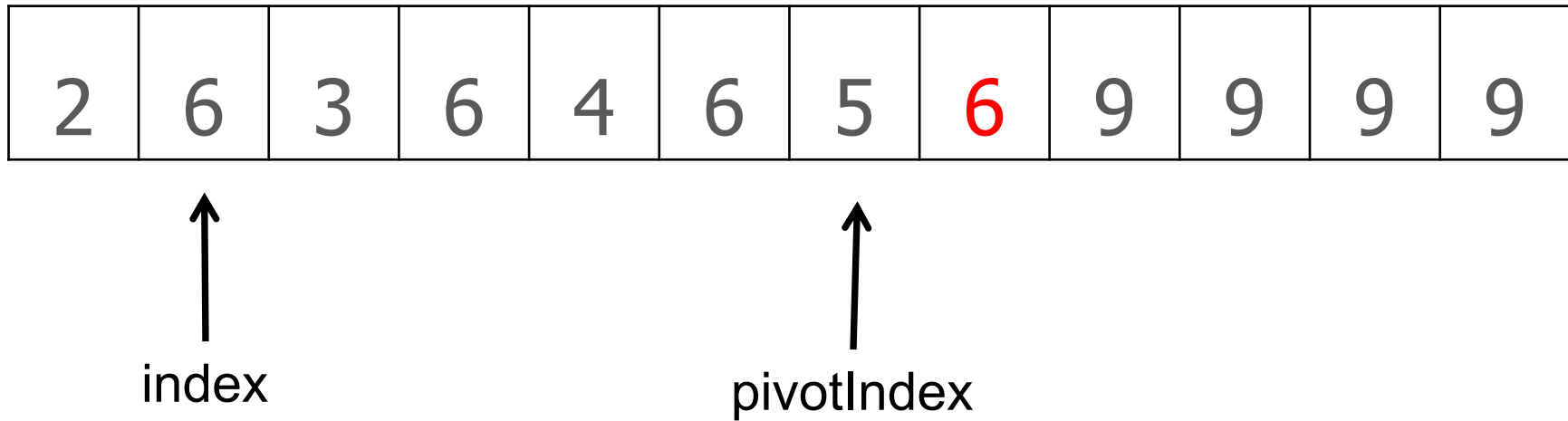




# Pack Duplicates

---

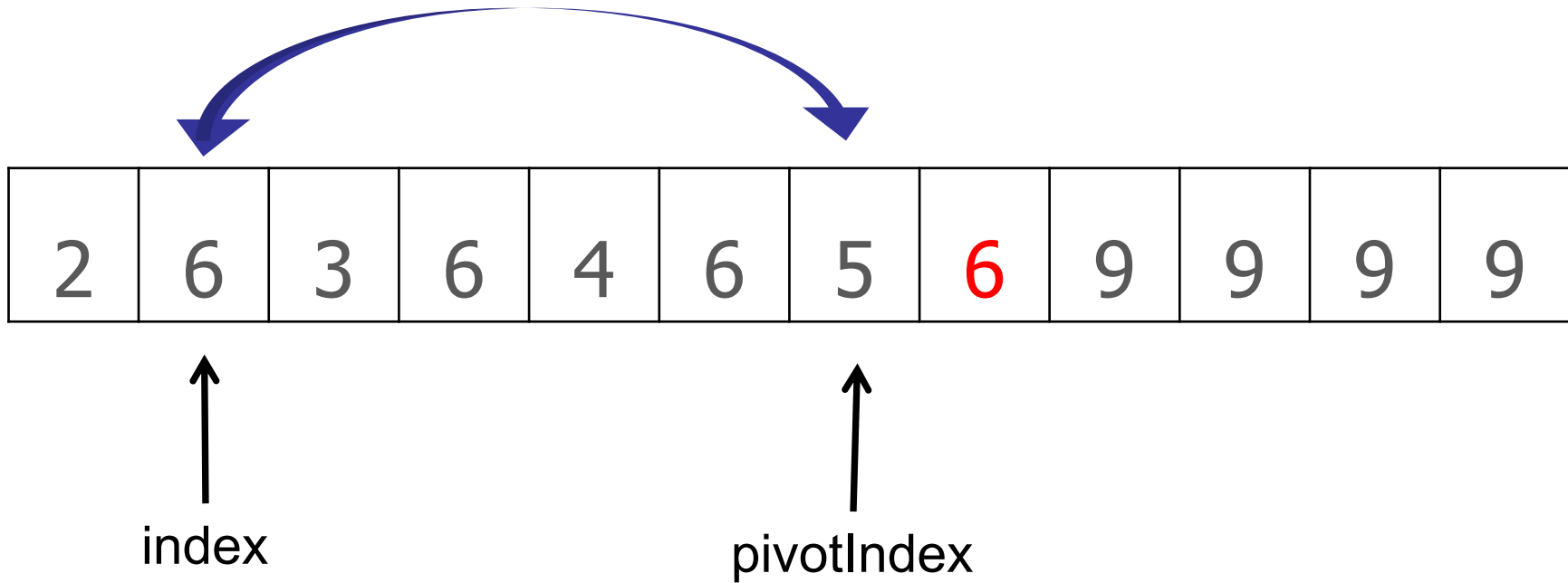
Example:



# Pack Duplicates

---

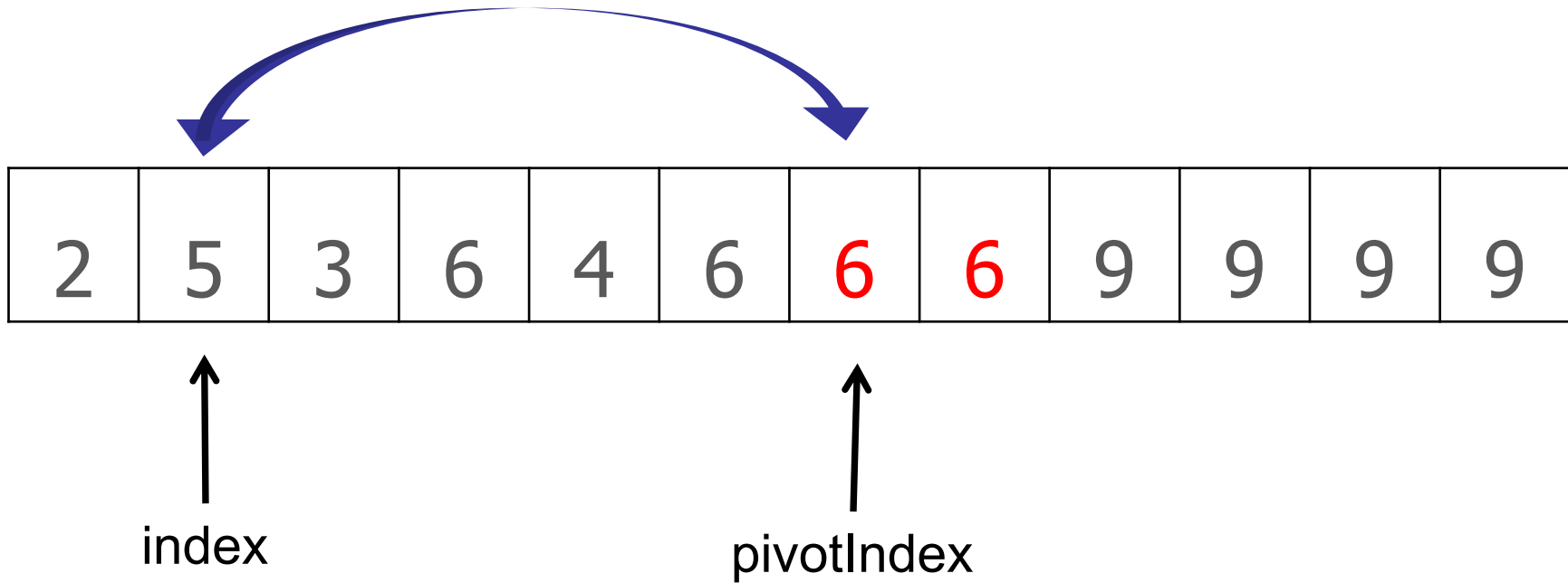
Example:



# Pack Duplicates

---

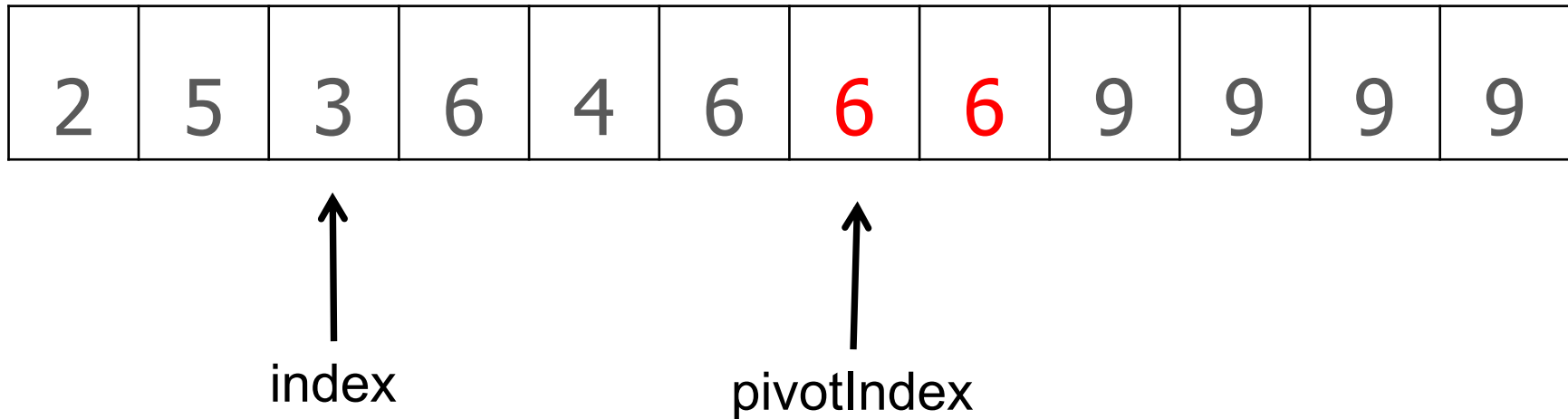
Example:



# Pack Duplicates

---

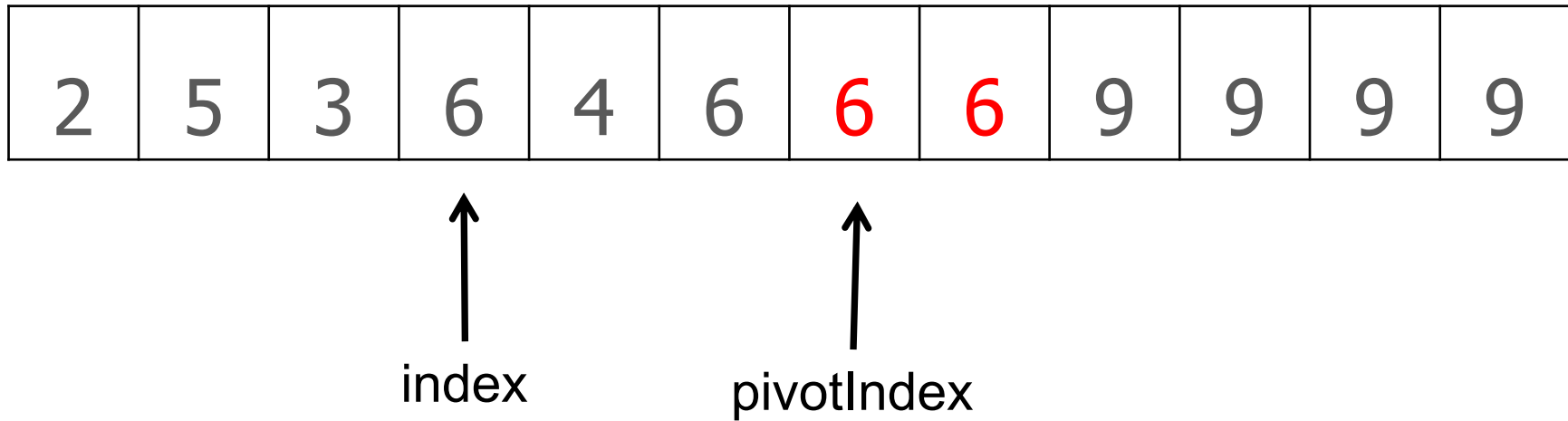
Example:



# Pack Duplicates

---

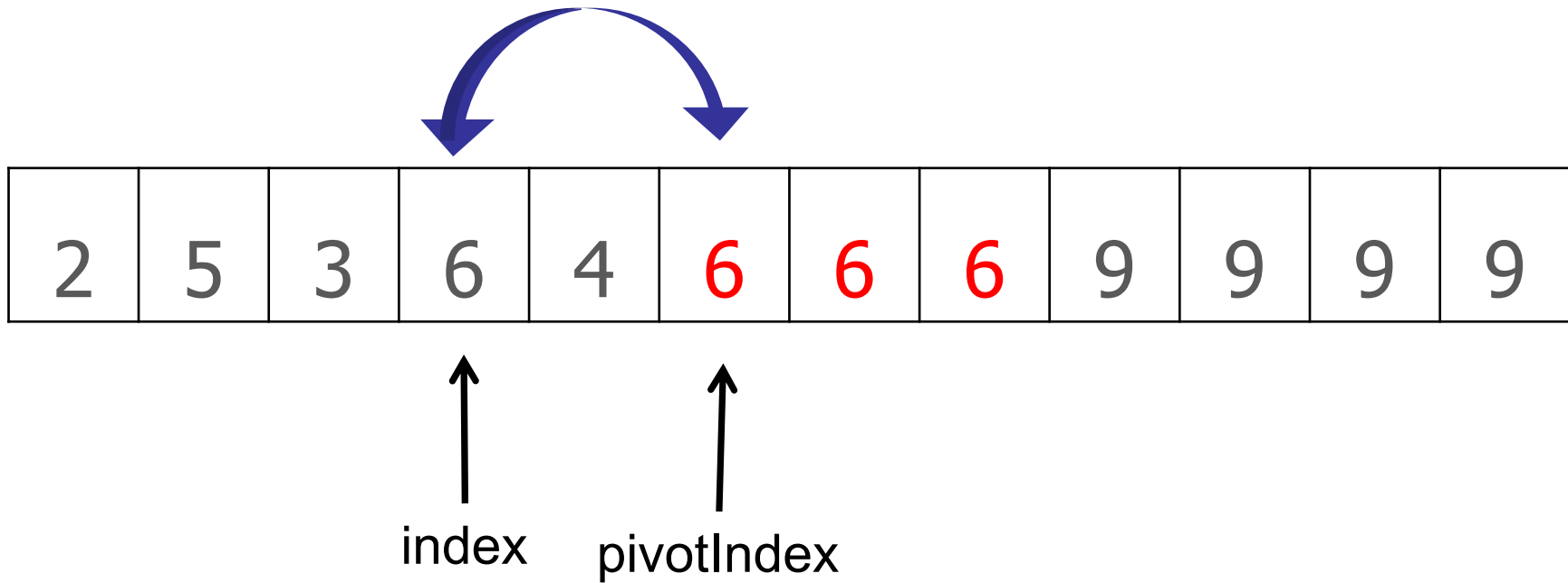
Example:



# Pack Duplicates

---

Example:



# Pack Duplicates

---

Example:

2	5	3	6	4	6	6	6	9	9	9	9
---	---	---	---	---	---	---	---	---	---	---	---

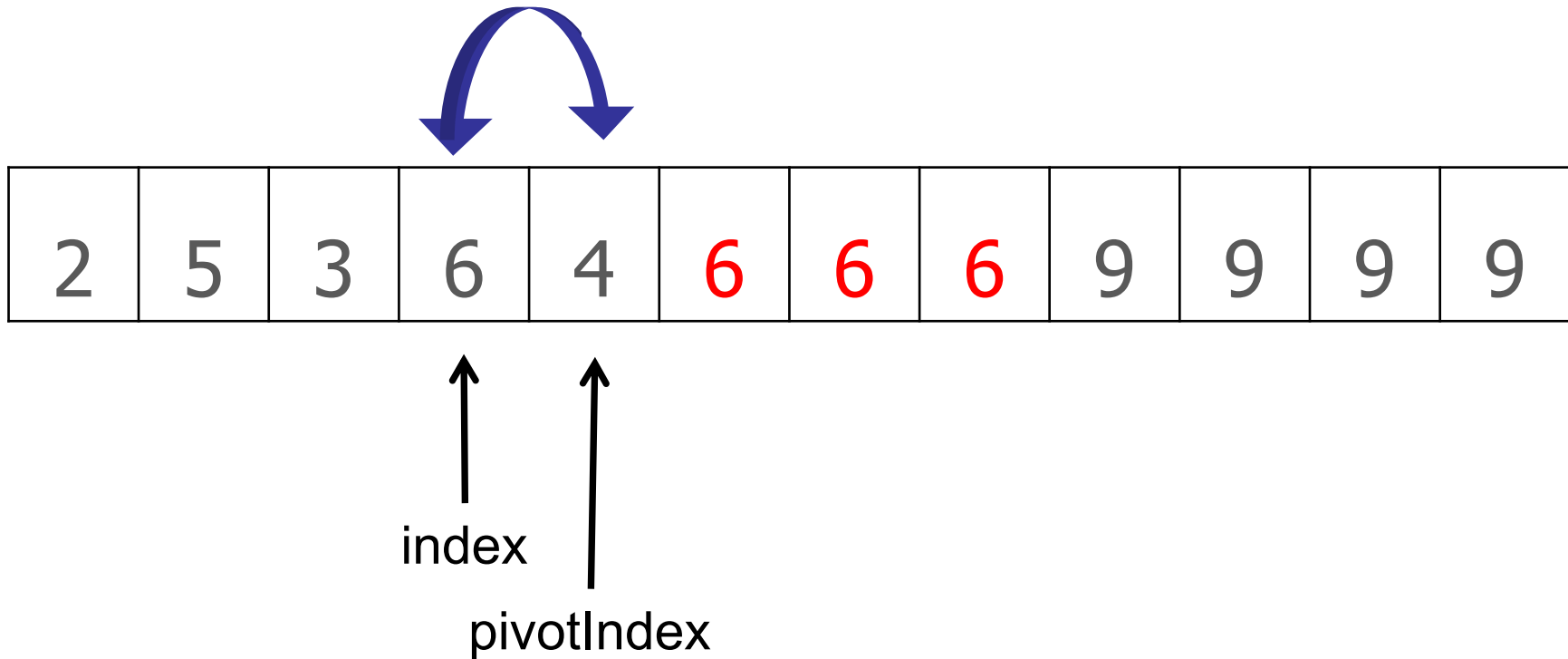
↑  
index

↑  
pivotIndex

# Pack Duplicates

---

Example:

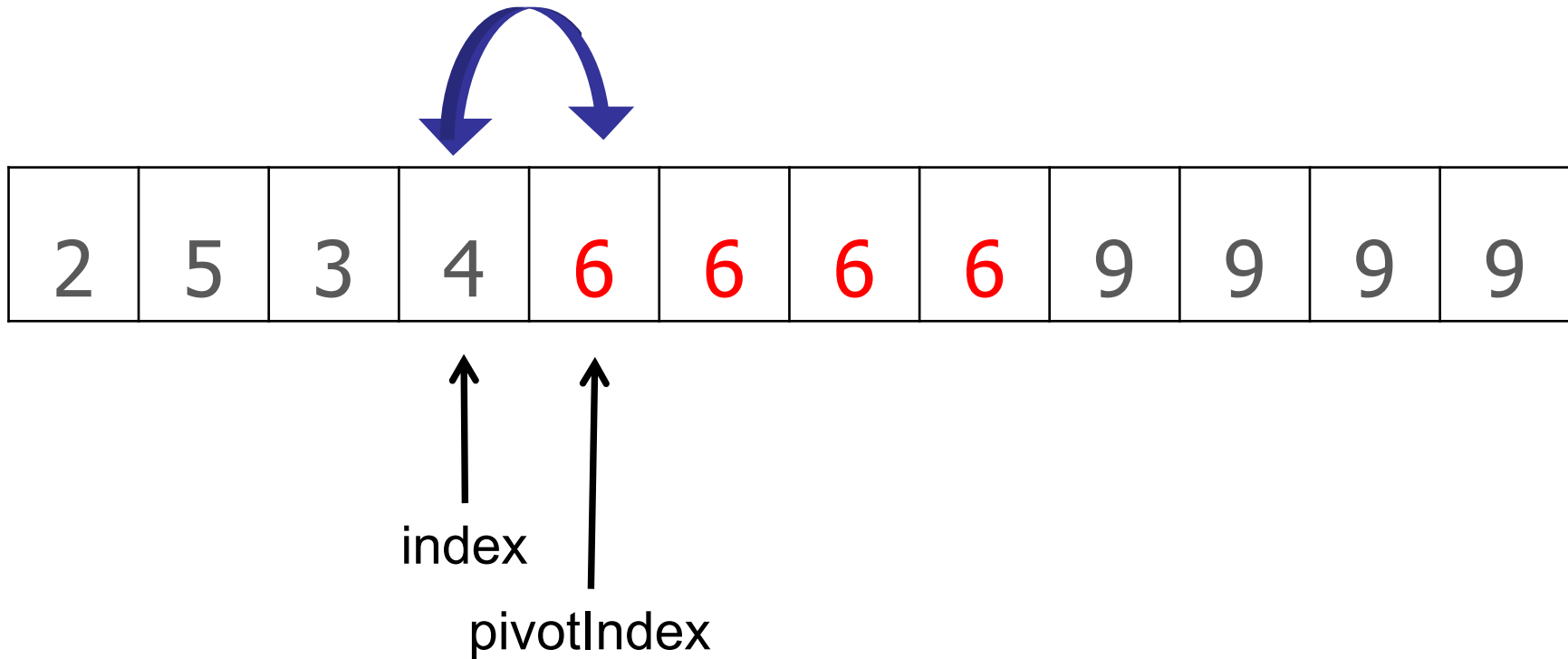




# Pack Duplicates

---

Example:



# Pack Duplicates

---

Example:

2	5	3	4	6	6	6	6	9	9	9	9
---	---	---	---	---	---	---	---	---	---	---	---

↑  
index  
pivotIndex

# Duplicates

---

**QuickSort**( $A[1..n]$ ,  $n$ )

**if** ( $n==1$ ) **then** return;

**else**

Choose pivot index  $pIndex$ .

$p = \text{3wayPartition}(A[1..n], n, pIndex)$

$x = \text{QuickSort}(A[1..p-1], p-1)$

$y = \text{QuickSort}(A[p+1..n], n-p)$

$< x$

$x \quad x \quad x$

$> x$

# Duplicates

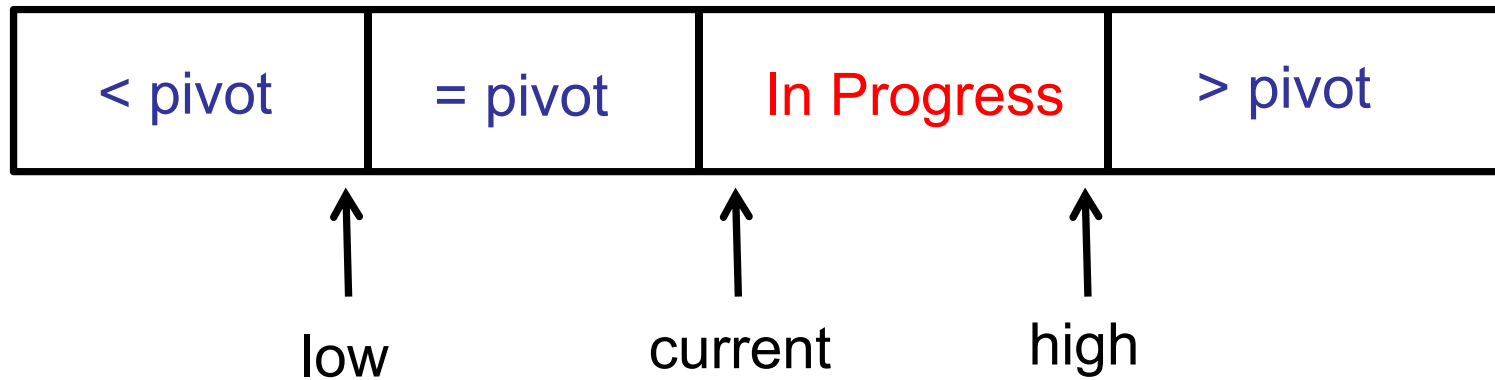
---

## 3-Way Partitioning

- Option 1: two pass partitioning
  1. Regular partition.
  2. Pack duplicates.
- Option 2: one pass partitioning
  - Standard solution.
  - Maintain four regions of the array

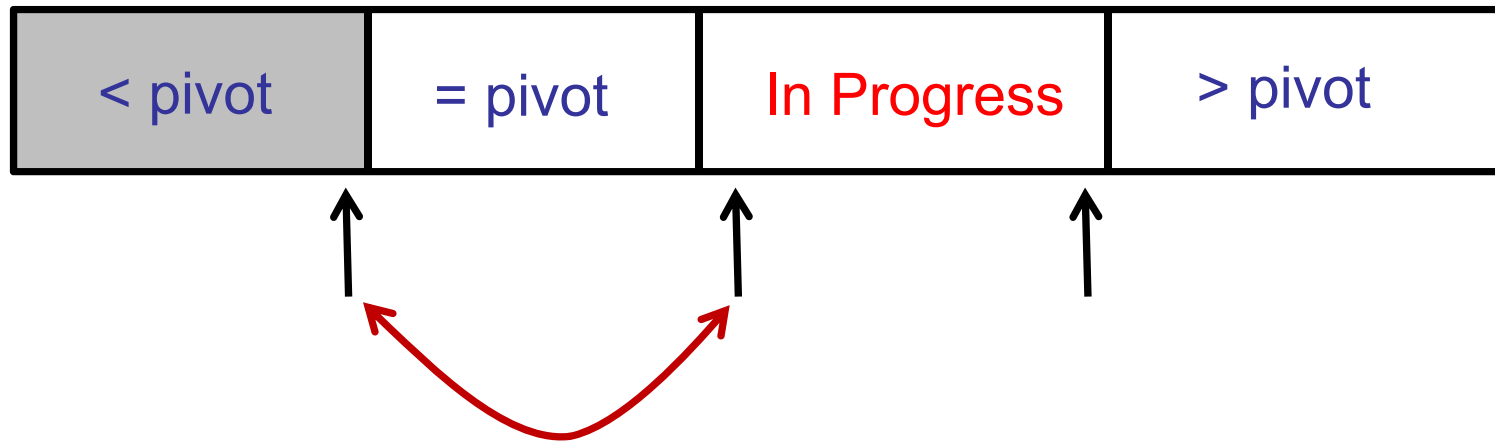
# 3-Way Partitioning

---



# 3-Way Partitioning

---

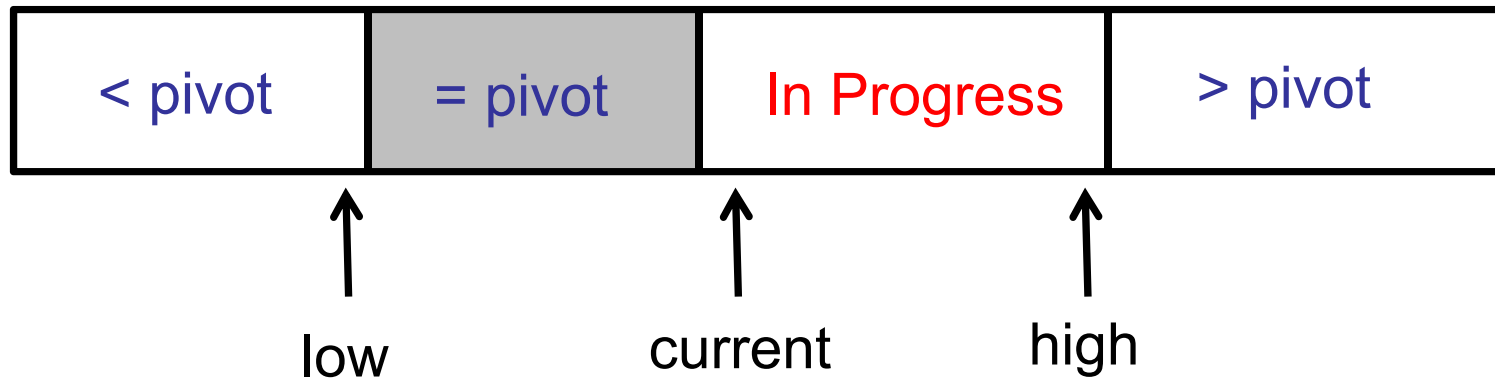


**if  $A[\text{current}] < \text{pivot}$ :**

- Increment low
- Swap  $A[\text{current}]$ ,  $A[\text{low}]$
- Increment current

# 3-Way Partitioning

---

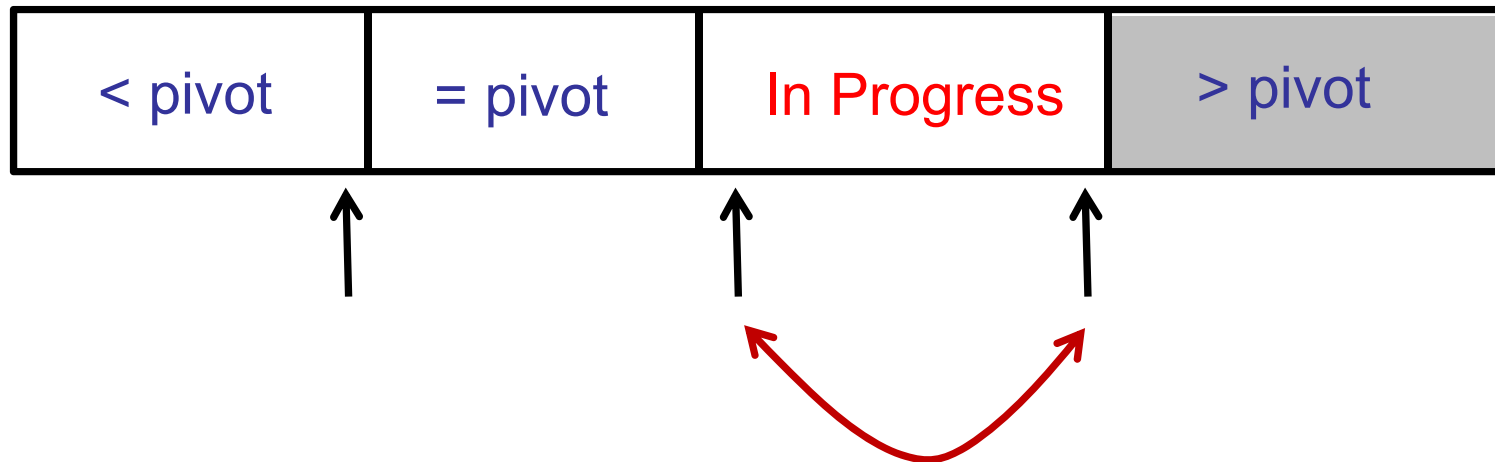


**if  $A[\text{current}] == \text{pivot}$ :**

- Increment current

# 3-Way Partitioning

---



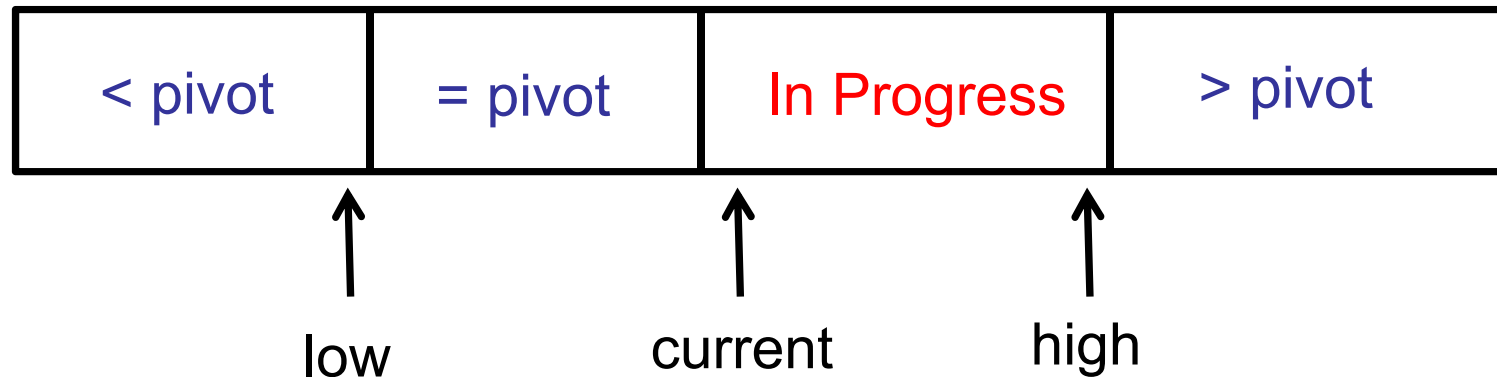
**if  $A[\text{current}] > \text{pivot}$ :**

- Swap  $A[\text{current}]$ ,  $A[\text{high}]$
- Decrement high



# 3-Way Partitioning

---

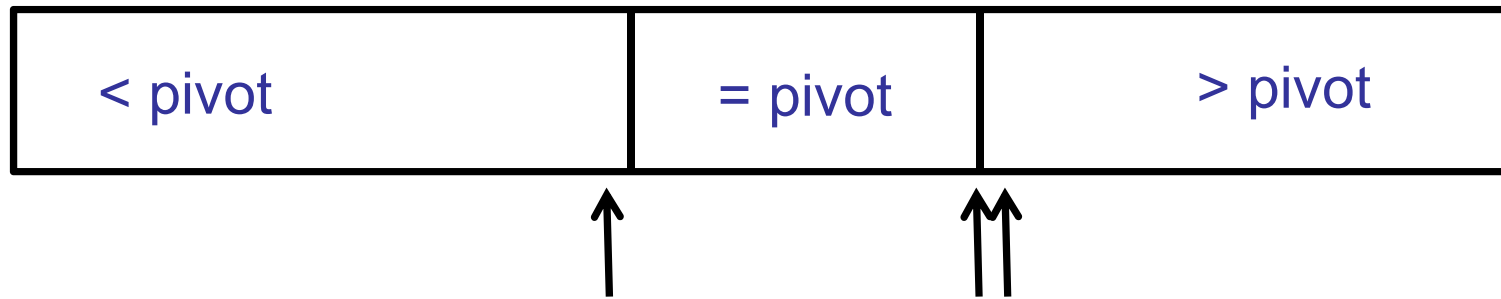


## Invariants:

- Each region has proper elements (< pivot, = pivot, > pivot).
- Each iteration, In Progress region decreases by one.

# 3-Way Partitioning

---



# Duplicates

---

**QuickSort**( $A[1..n]$ ,  $n$ )

**if** ( $n==1$ ) **then** return;

**else**

Choose pivot index  $pIndex$ .

$p = \text{3wayPartition}(A[1..n], n, pIndex)$

$x = \text{QuickSort}(A[1..p-1], p-1)$

$y = \text{QuickSort}(A[p+1..n], n-p)$

$< x$

$x \quad x \quad x$

$> x$

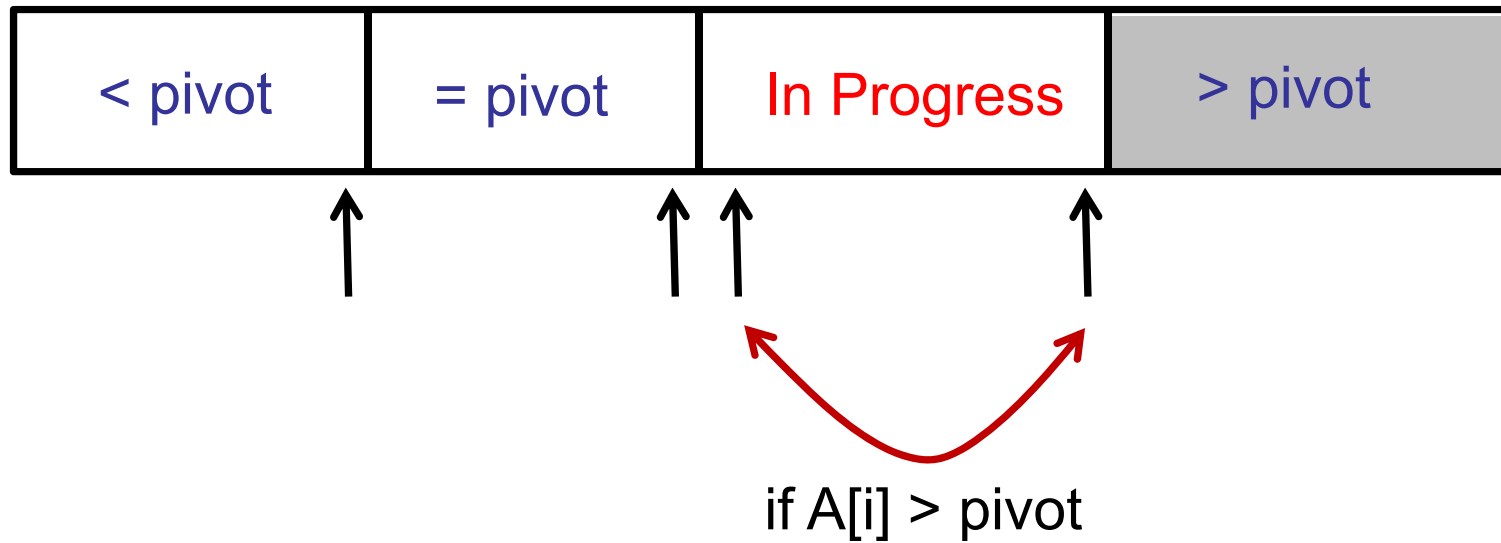
# Is QuickSort stable?

**ARCHIPELAGO**

is open

# QuickSort is not stable

---



# Sorting, Part II

---

## QuickSort

- Divide-and-Conquer
- Partitioning
- Duplicates
- Choosing a pivot
- Randomization
- Analysis

# Choice of Pivot

---

## Options:

- first element:  $A[1]$
- last element:  $A[n]$
- middle element:  $A[n/2]$
- median of  $(A[1], A[n/2], A[n])$

**ARCHIPELAGO**

is open

# Choice of Pivot

---

## Options:

- first element:  $A[1]$
- last element:  $A[n]$
- middle element:  $A[n/2]$
- median of  $(A[1], A[n/2], A[n])$

In the worst case, it does not matter!

All options are equally bad.

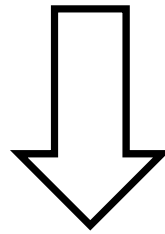


# Choice of Pivot

---

Choose  $A[1]$  for pivot:

100 99 98 97 96 95 94 93 92



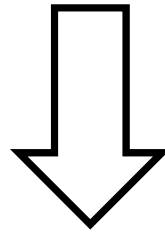
99 98 97 96 95 94 93 92 100

# Choice of Pivot

---

Choose  $A[1]$  for pivot:

99 98 97 96 95 94 93 92 100



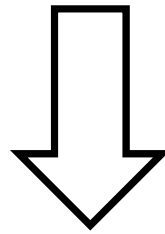
98 97 96 95 94 93 92 99 100

# Choice of Pivot

---

Choose  $A[1]$  for pivot:

98 97 96 95 94 93 92 99 100



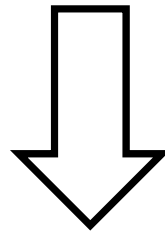
97 96 95 94 93 92 98 99 100

# Choice of Pivot

---

Choose  $A[1]$  for pivot:

98 97 96 95 94 93 92 99 100



97 96 95 94 93 92 98 99 100

# Choice of Pivot

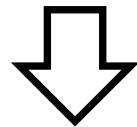
---

Sorting the array takes  $n$  executions of **partition**.

- Each call to **partition** sorts one element.
- Each call to **partition** of size  $k$  takes:  $\geq k$

Total:  $n + (n-1) + (n-2) + (n-3) + \dots = O(n^2)$

98 97 96 95 94 93 92 99 100



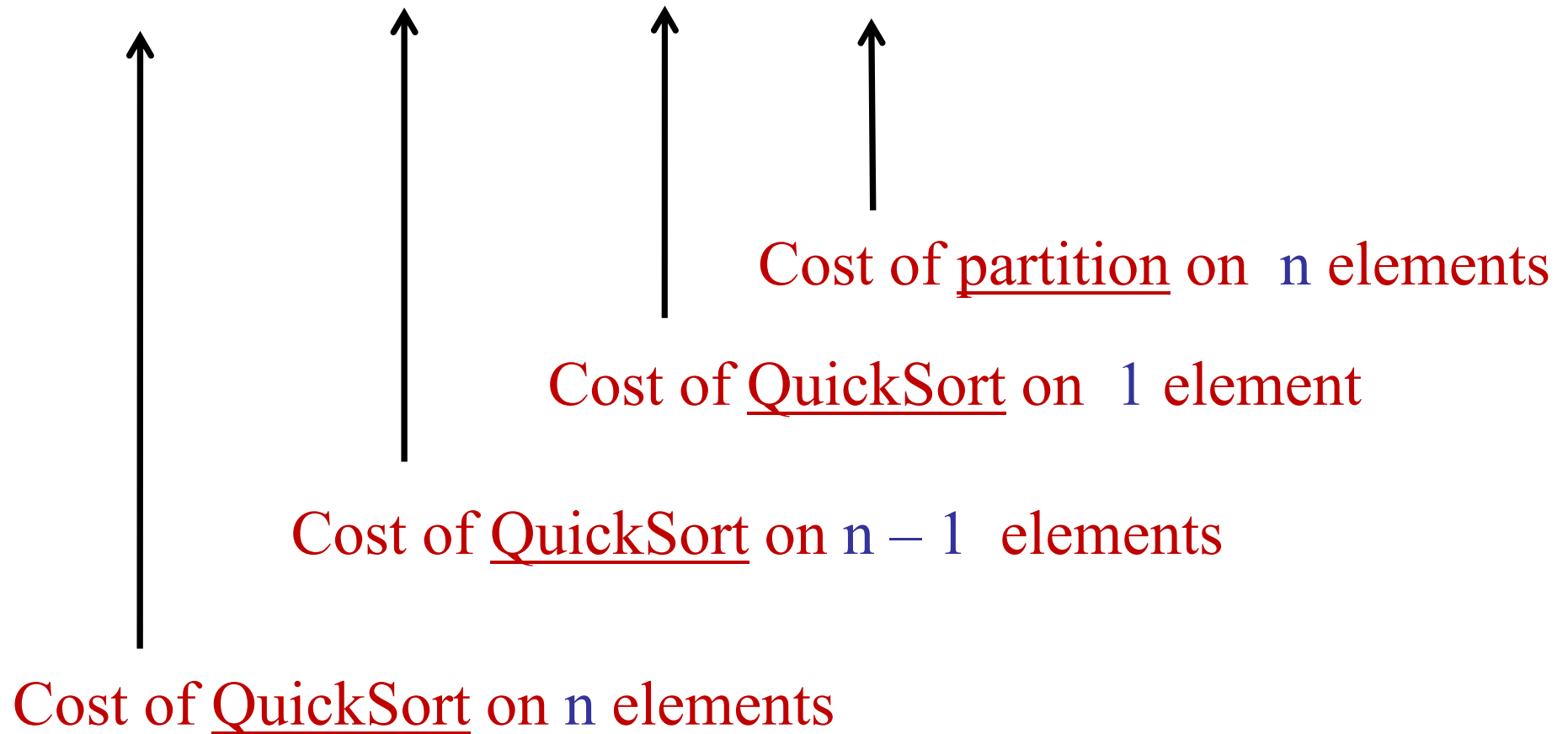
97 96 95 94 93 92 98 99 100

# Deterministic QuickSort

---

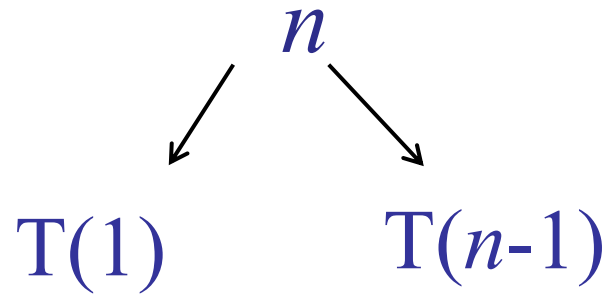
QuickSort Recurrence:

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



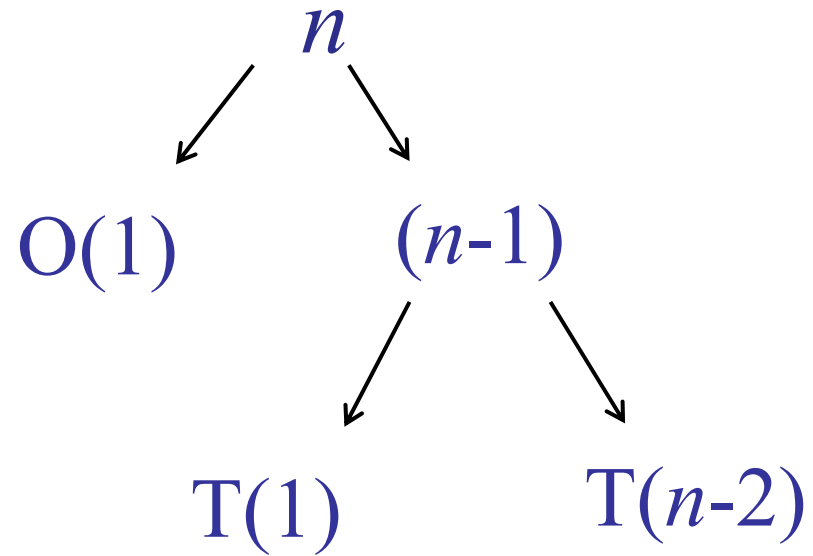
# Deterministic QuickSort

---



# Deterministic QuickSort

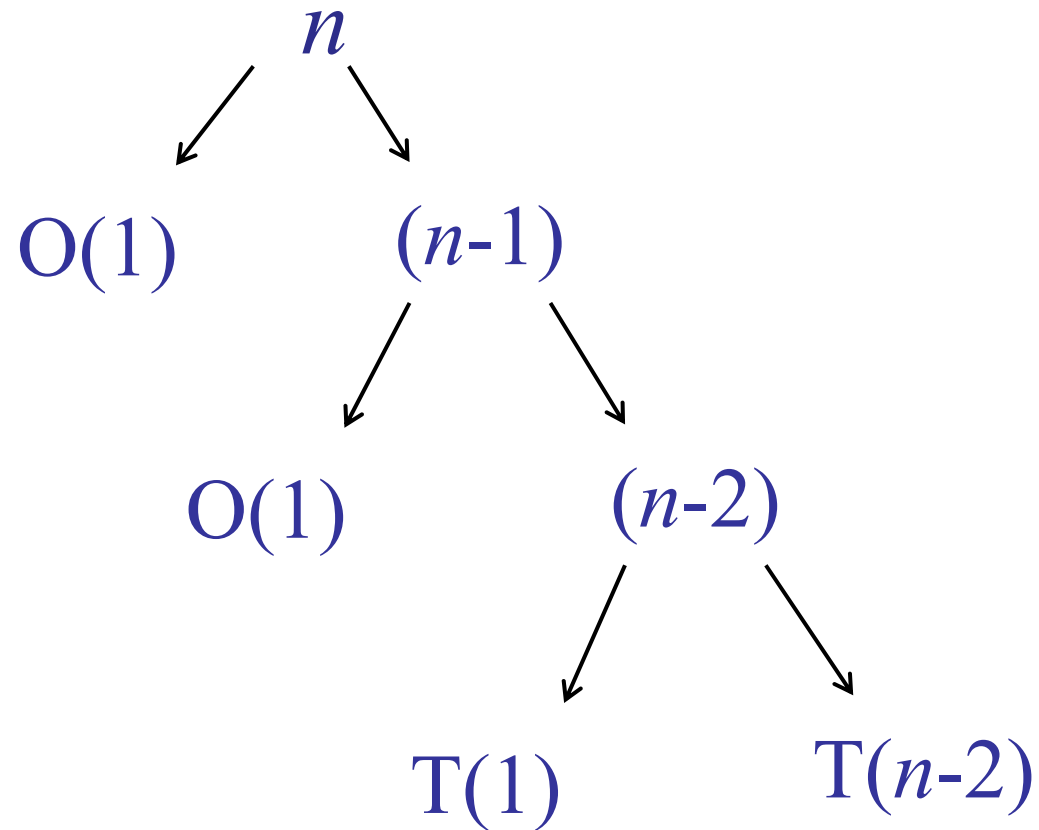
---



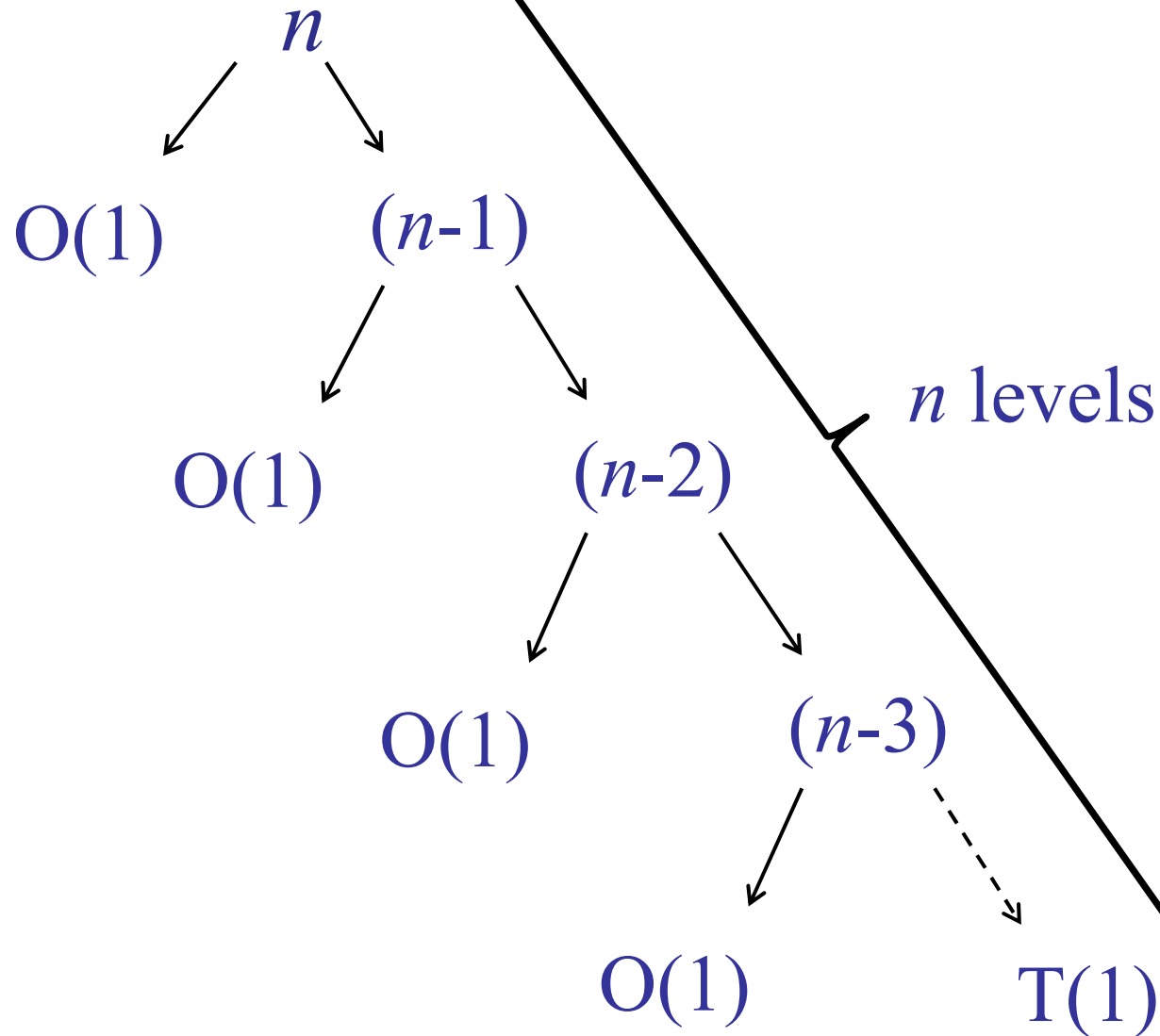


# Deterministic QuickSort

---

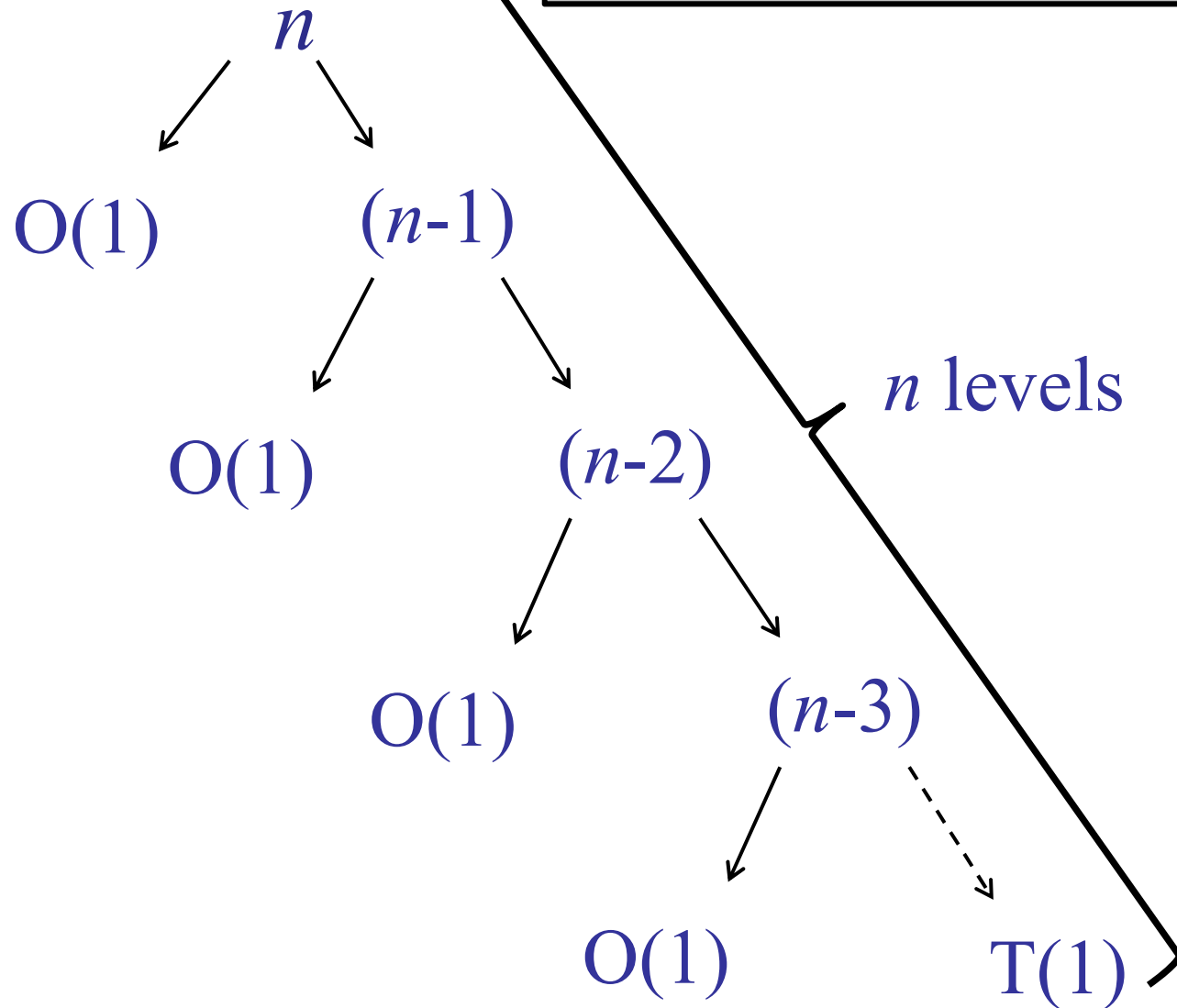


# Deterministic QuickSort



# Deterministic QuickSort

$$n + (n-1) + (n-2) + (n-3) + \dots = O(n^2)$$



# QuickSort

---

**QuickSort**( $A[1..n]$ ,  $n$ )

**if** ( $n==1$ ) **then** return;

**else**

Choose pivot index  $pIndex$ .

$p = \text{partition}(A[1..n], n, pIndex)$

$x = \text{QuickSort}(A[1..p-1], p-1)$

$y = \text{QuickSort}(A[p+1..n], n-p)$

$< x$

$x$

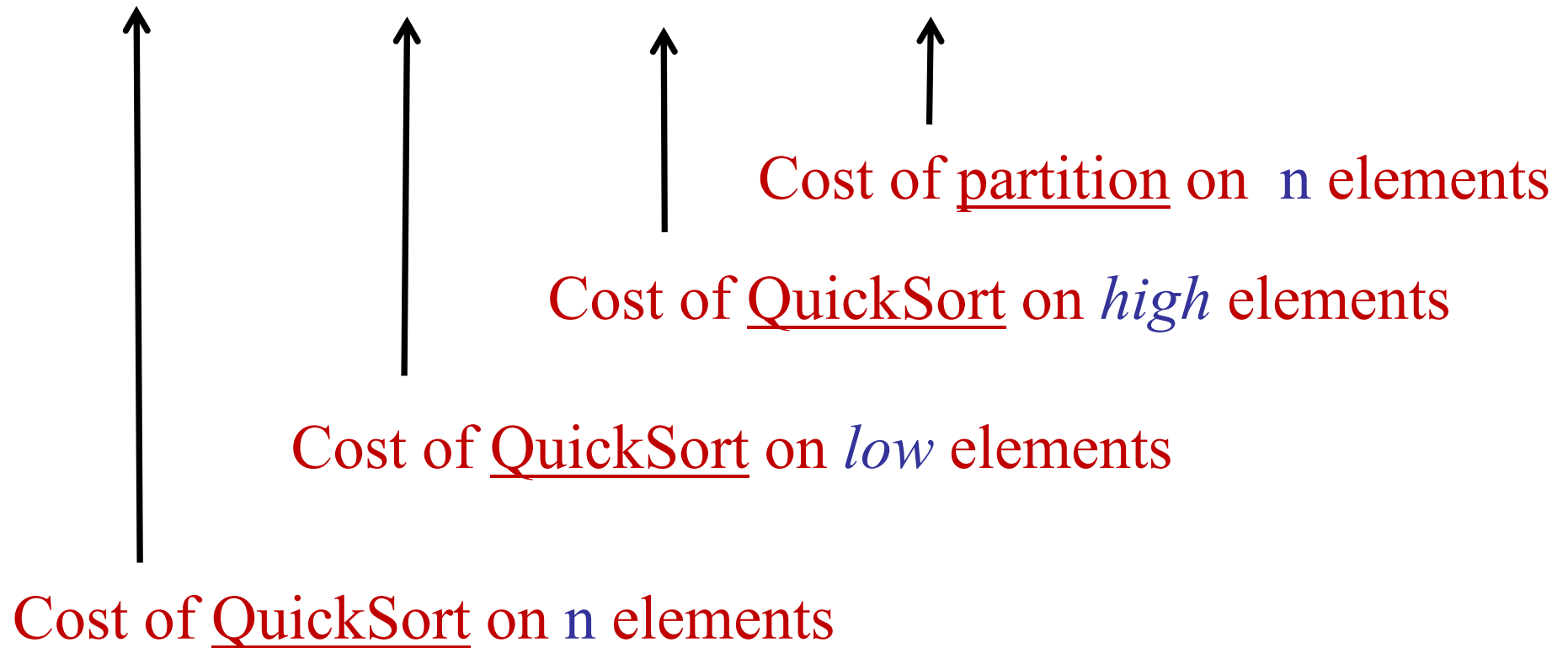
$> x$

# Better QuickSort

---

What if we chose the *median* element for the pivot?

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



# Better QuickSort

---

If we split the array evenly:

$$\begin{aligned}T(n) &= T(n/2) + T(n/2) + cn \\&= 2T(n/2) + cn \\&= O(n \log n)\end{aligned}$$

# QuickSort Summary

---

- If we choose the pivot as  $A[1]$ :
  - Bad performance:  $\Omega(n^2)$
- If we could choose the median element:
  - Good performance:  $O(n \log n)$
- If we could split the array  $(1/10) : (9/10)$ 
  - ??

# QuickSort Pivot Choice

---

Define sets  $L$  (low) and  $H$  (high):

- $L = \{A[i] : A[i] < pivot\}$
- $H = \{A[i] : A[i] > pivot\}$



What if the *pivot* is chosen so that:

1.  $L > n/10$
2.  $H > n/10$



# QuickSort

---

$$k = \min(|L|, |H|)$$

QuickSort with interesting *pivot* choice:

$$T(n) = T(n-k) + T(k) + n$$

Cost of partition on  $n$  elements

Assume:  $9n/10 > k > n/10$

Assume:  $9n/10 > (n - k) > n/10$

Cost of QuickSort on  $n$  elements

# QuickSort

---

Tempting solution:

$$\begin{aligned}T(n) &= T(n-k) + T(k) + n \\&< T(9n/10) + T(9n/10) + n \\&< 2T(9n/10) + n \\&< O(n \log n)\end{aligned}$$

What is wrong?

ARCHIPELAGO

is open

# QuickSort

---

Tempting solution:

$$\begin{aligned}T(n) &= T(n-k) + T(k) + n \\&< T(9n/10) + T(9n/10) + n \\&< 2T(9n/10) + n \\&\leq \cancel{O(n \log n)} \\&= O(n^{6.58})\end{aligned}$$

Too loose an estimate.

# QuickSort Pivot Choice

---

Define sets  $L$  (low) and  $H$  (high):

- $L = \{A[i] : A[i] < pivot\}$
- $H = \{A[i] : A[i] > pivot\}$



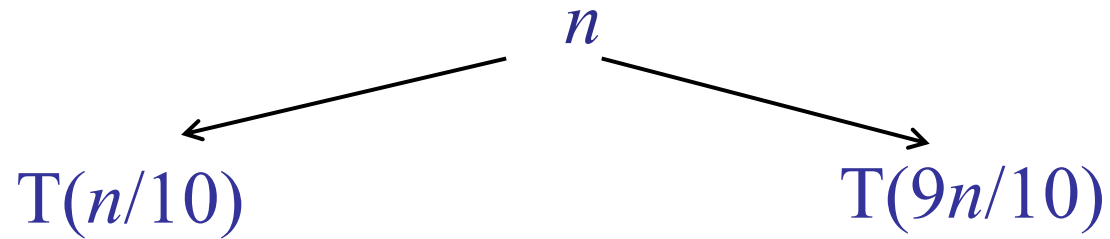
What if the *pivot* is chosen so that:

1.  $L = n(1/10)$
2.  $H = n(9/10)$  (or *vice versa*)

# QuickSort Analysis

---

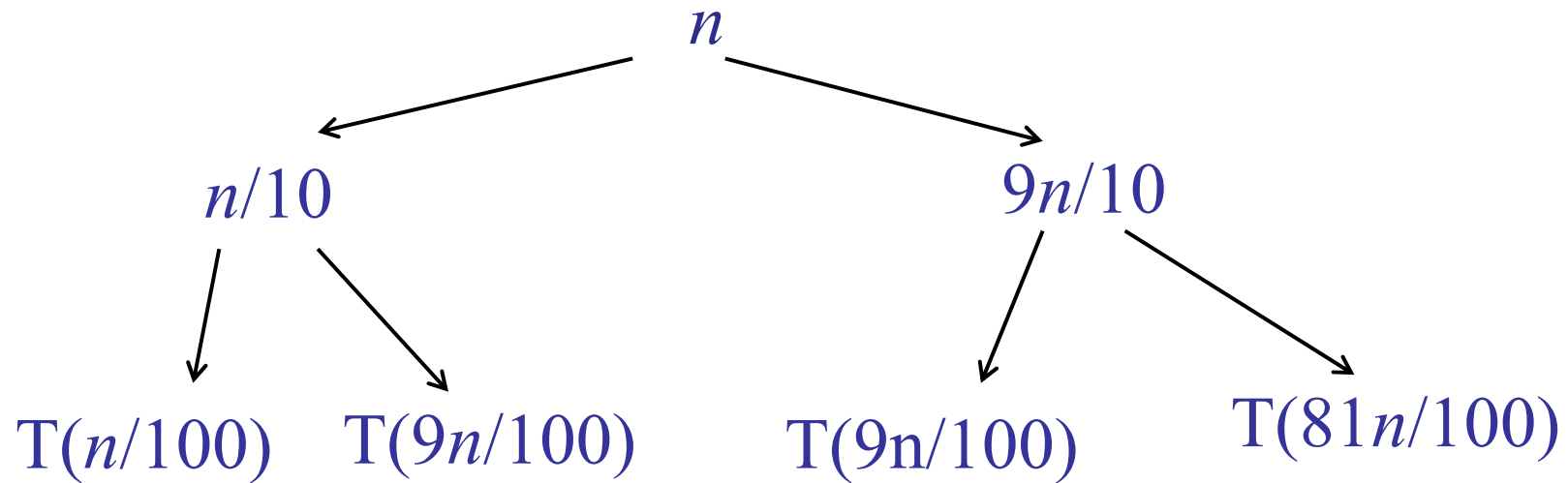
$$k = n/10$$



# QuickSort Analysis

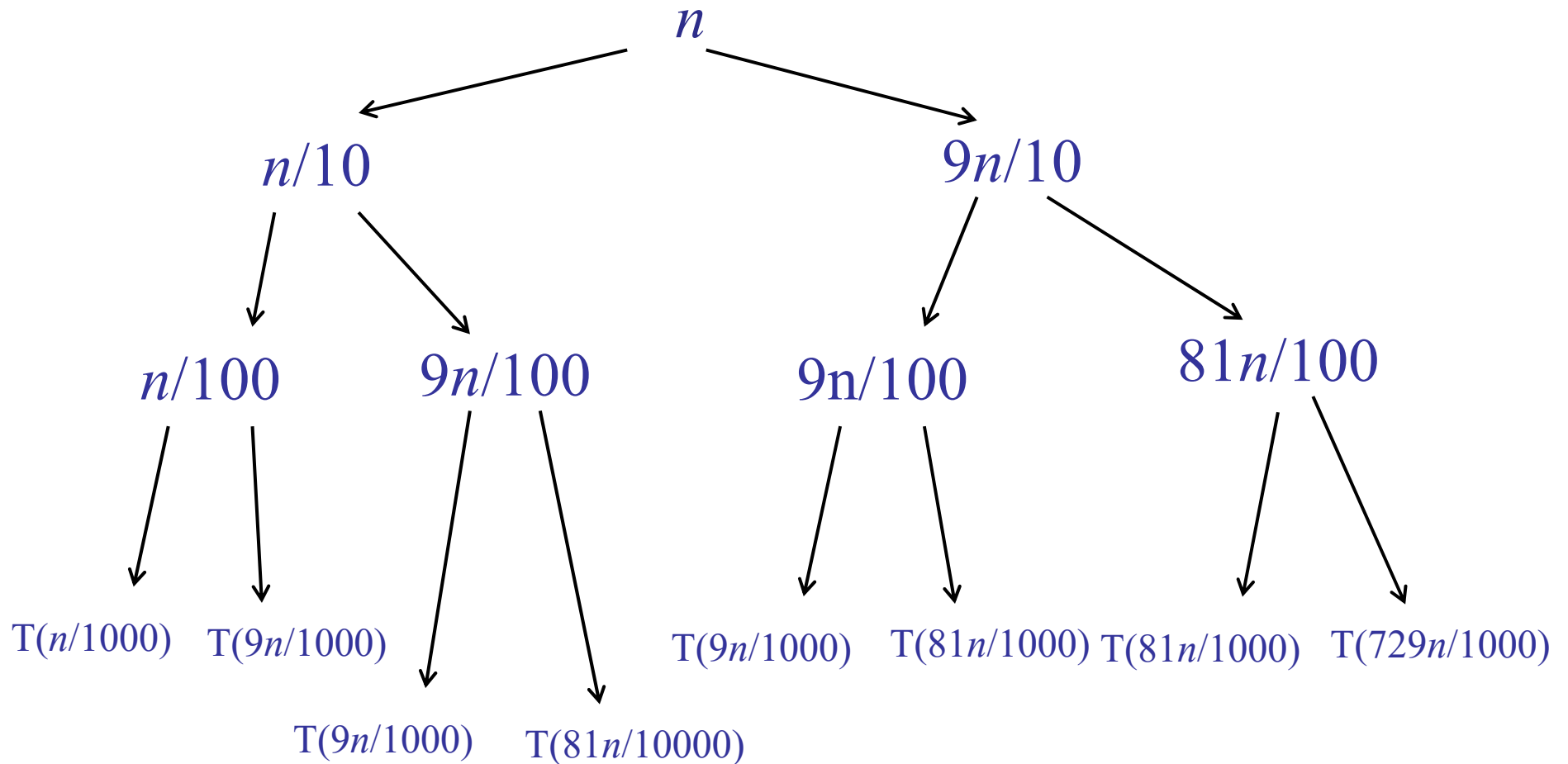
---

$$k = n/10$$



# QuickSort Analysis

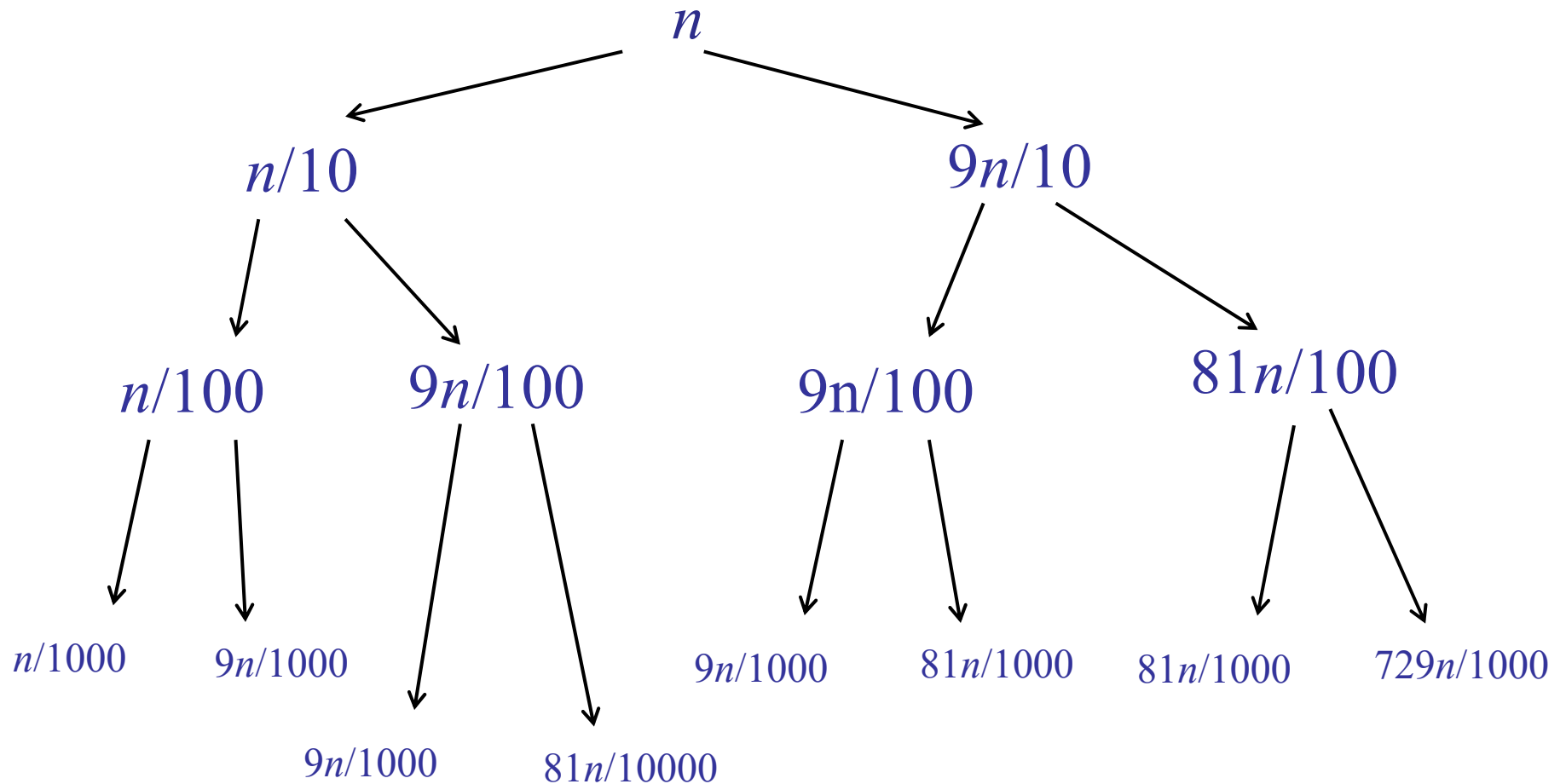
$$k = n/10$$



# QuickSort Analysis

---

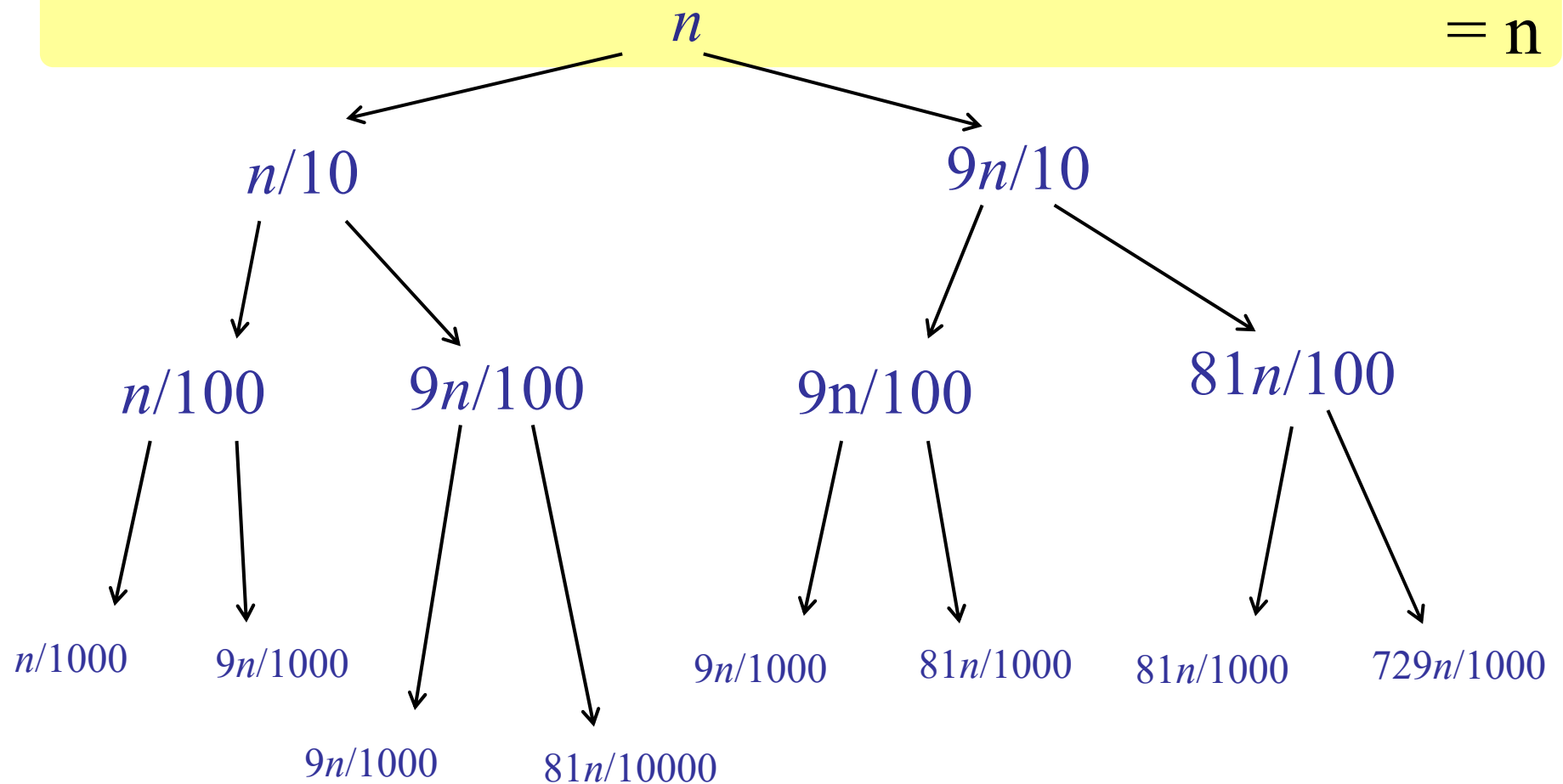
$$k = n/10$$





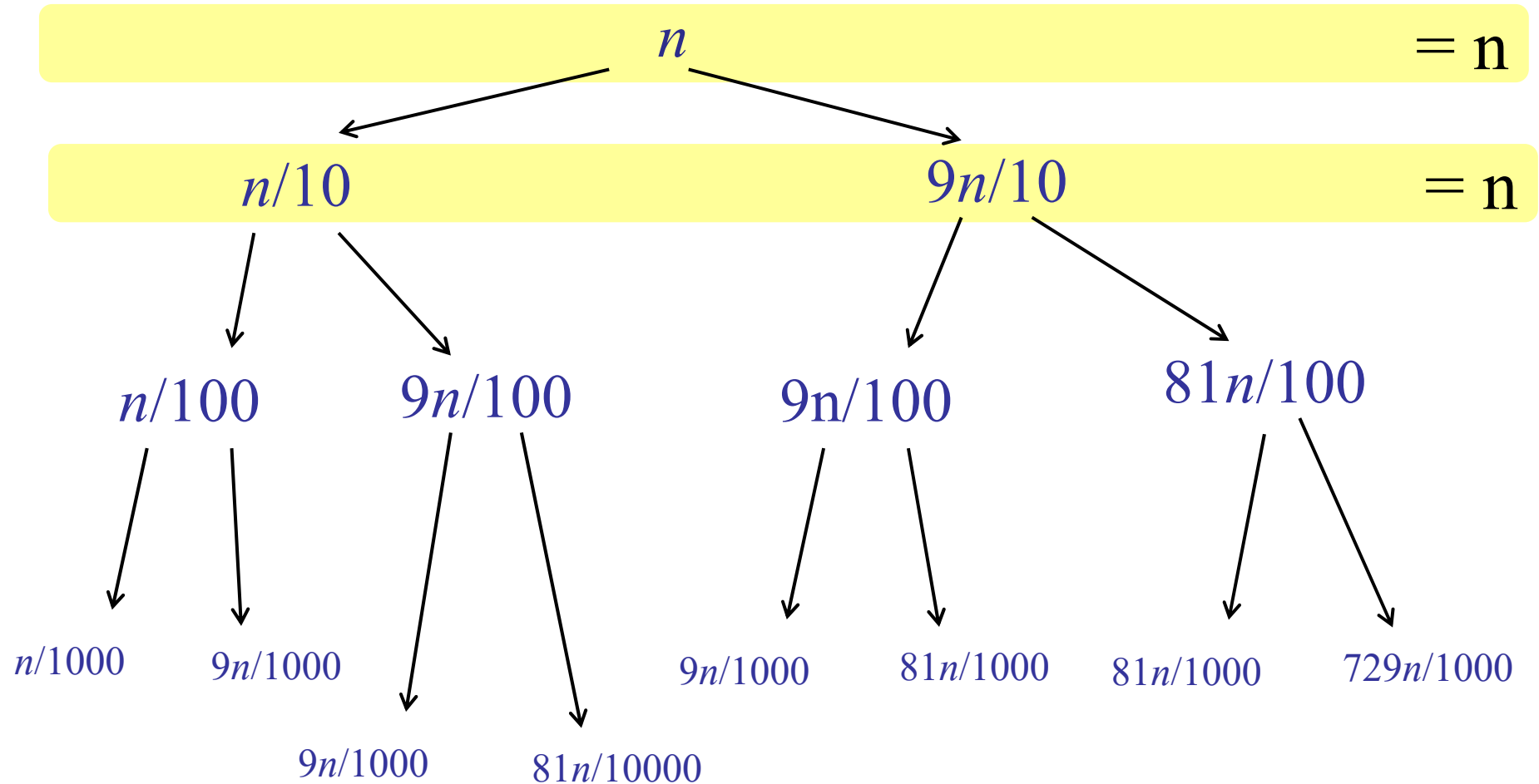
# QuickSort Analysis

---



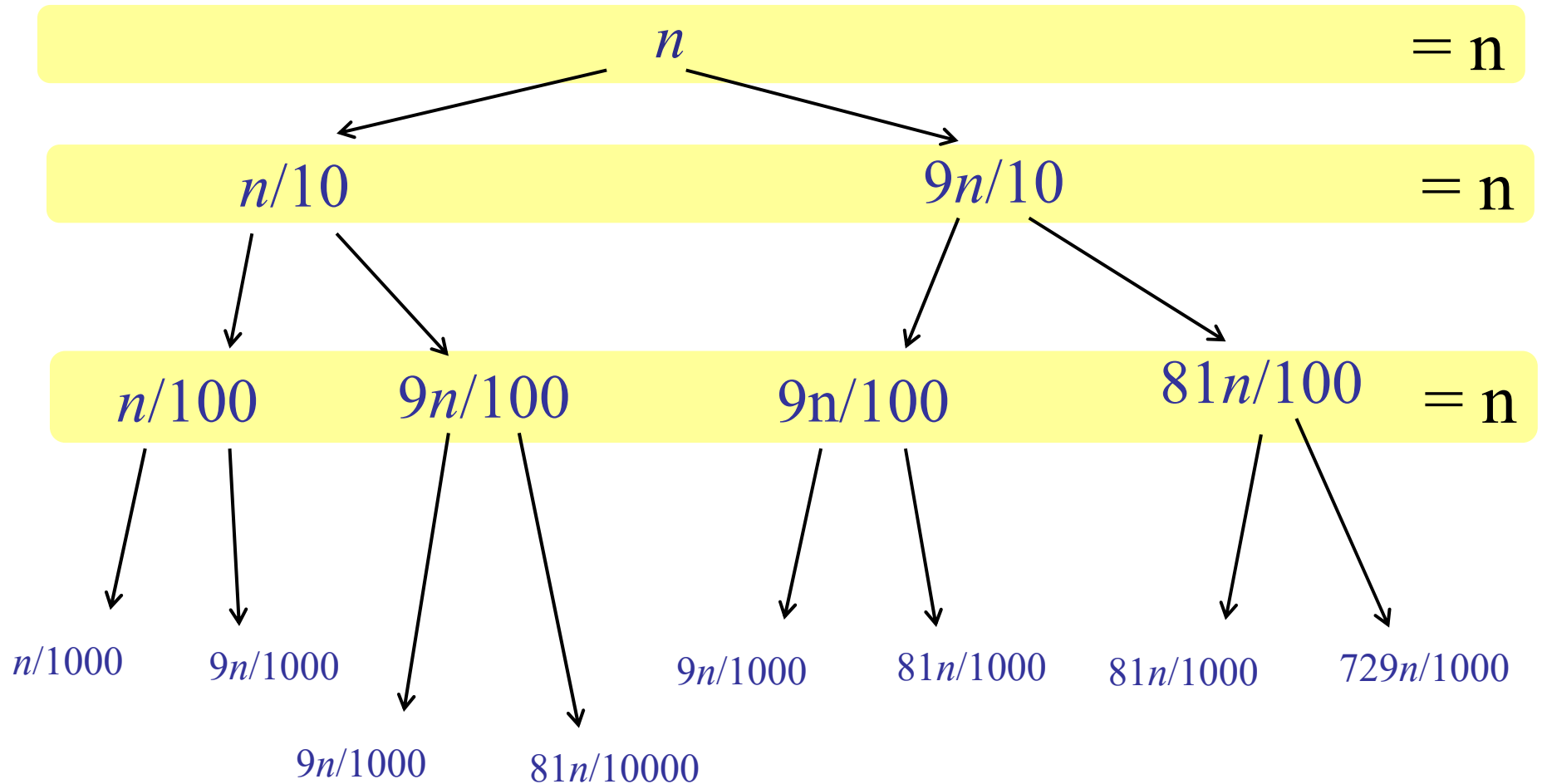
# QuickSort Analysis

---

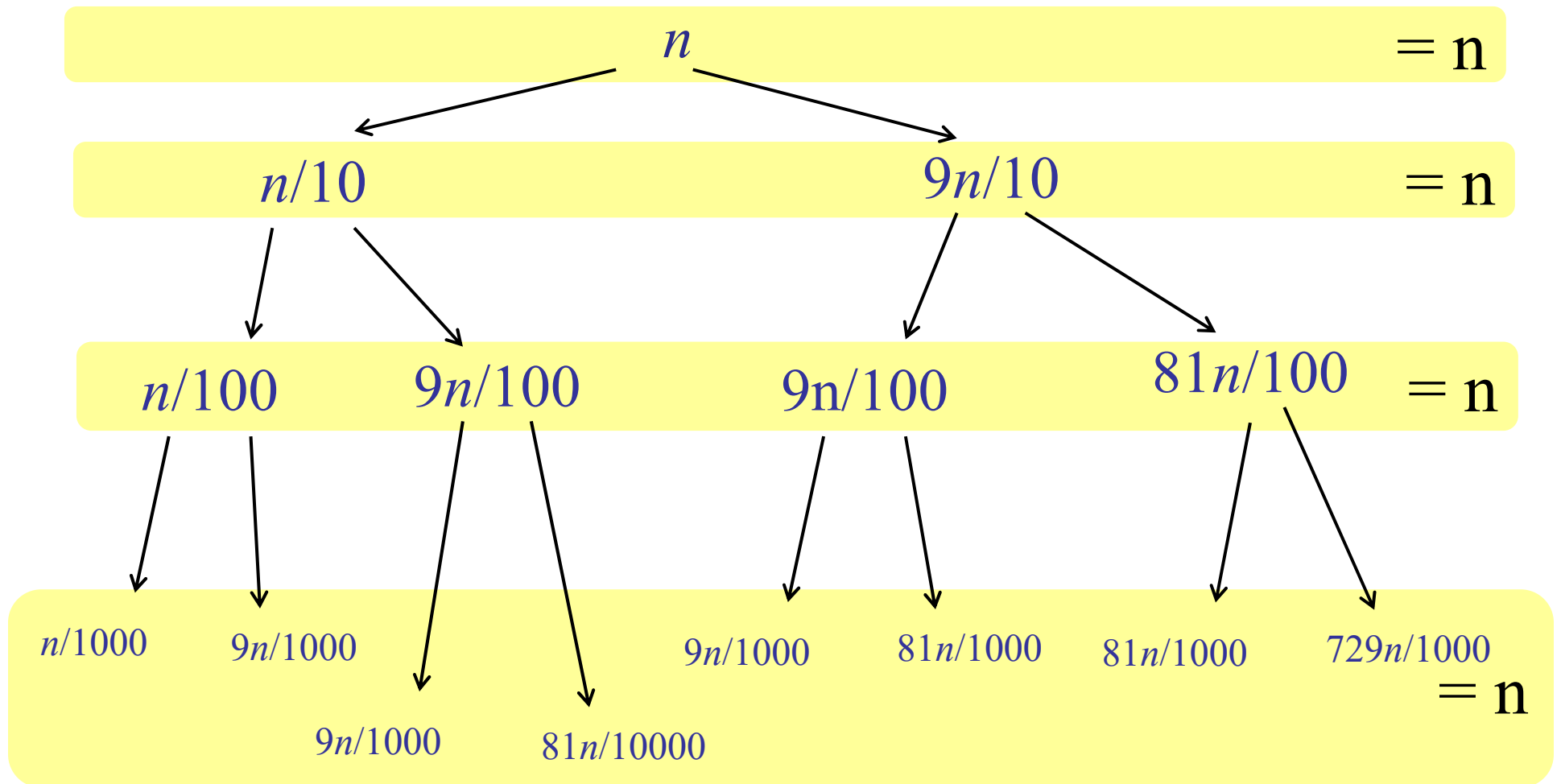


# QuickSort Analysis

---



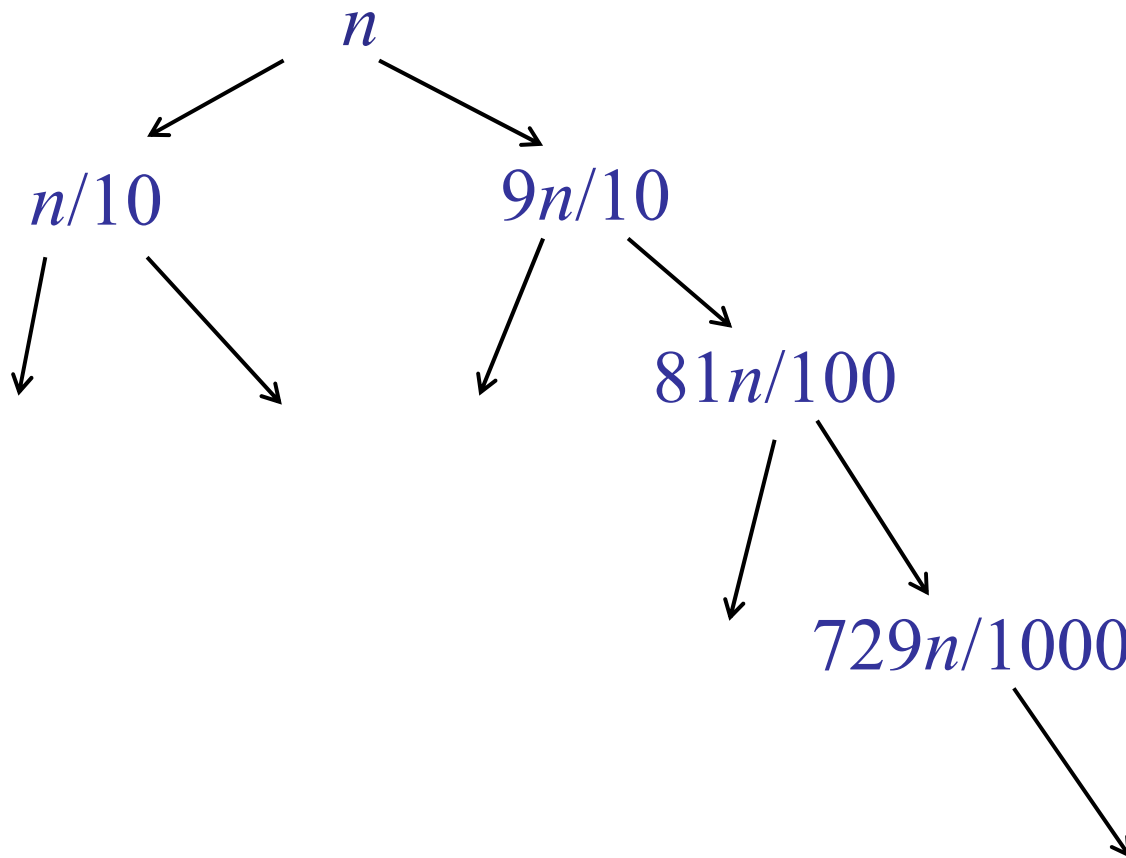
# QuickSort Analysis



How many levels??

# QuickSort Analysis

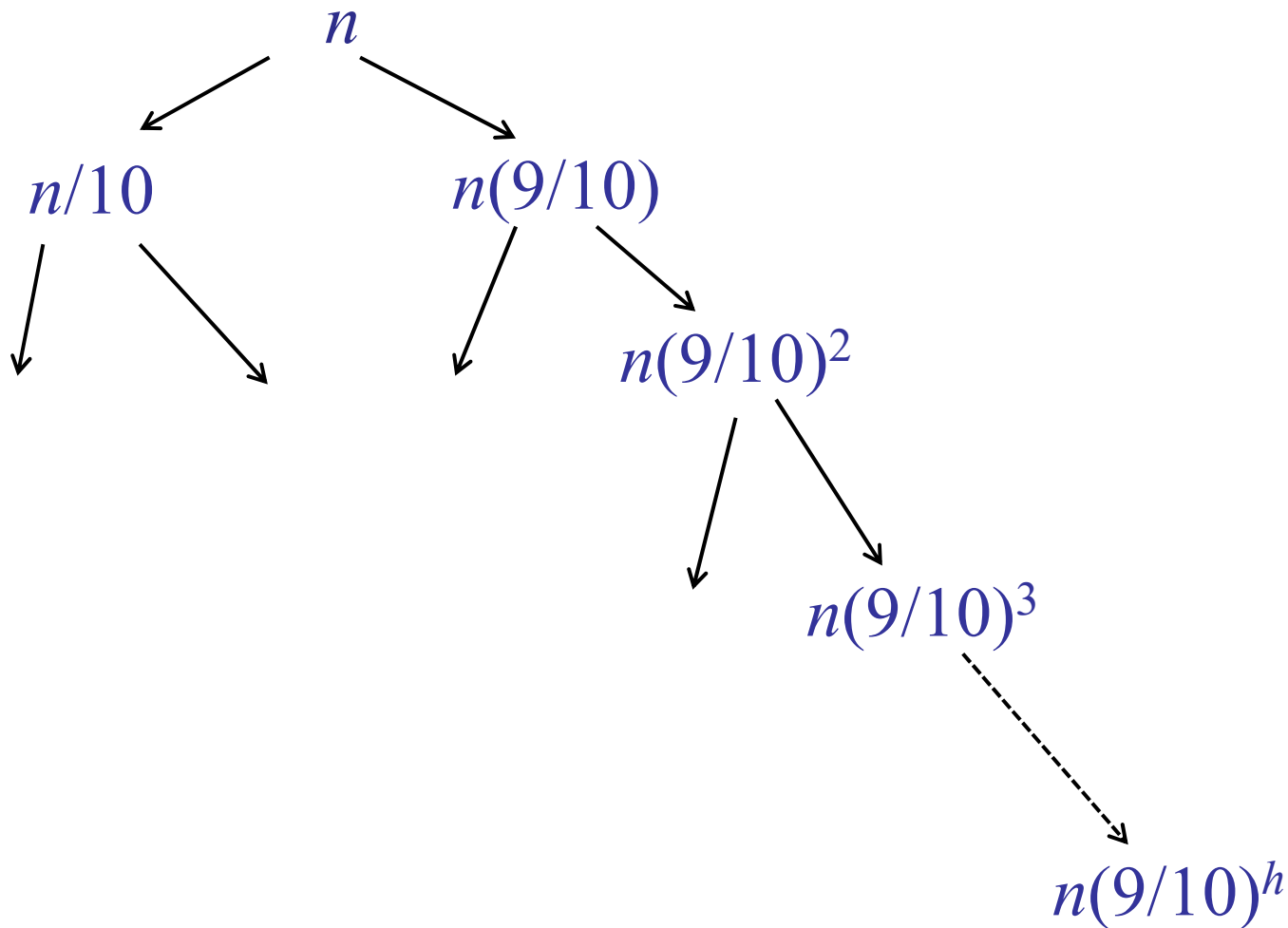
---



How many levels??

# QuickSort Analysis

---



How many levels??

# QuickSort Analysis

---

Maximum number of levels:

$$1 = n(9/10)^h$$

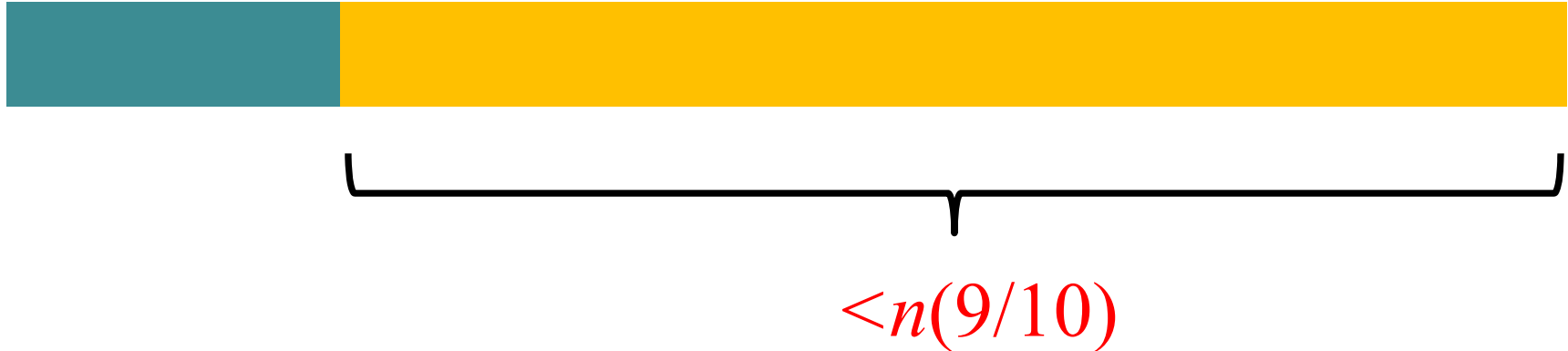
$$(10/9)^h = n$$

$$h = \log_{10/9}(n) = O(\log n)$$

# QuickSort Analysis

---

Assume larger part shrinks by at least 9/10 every iteration:





# QuickSort Analysis

---

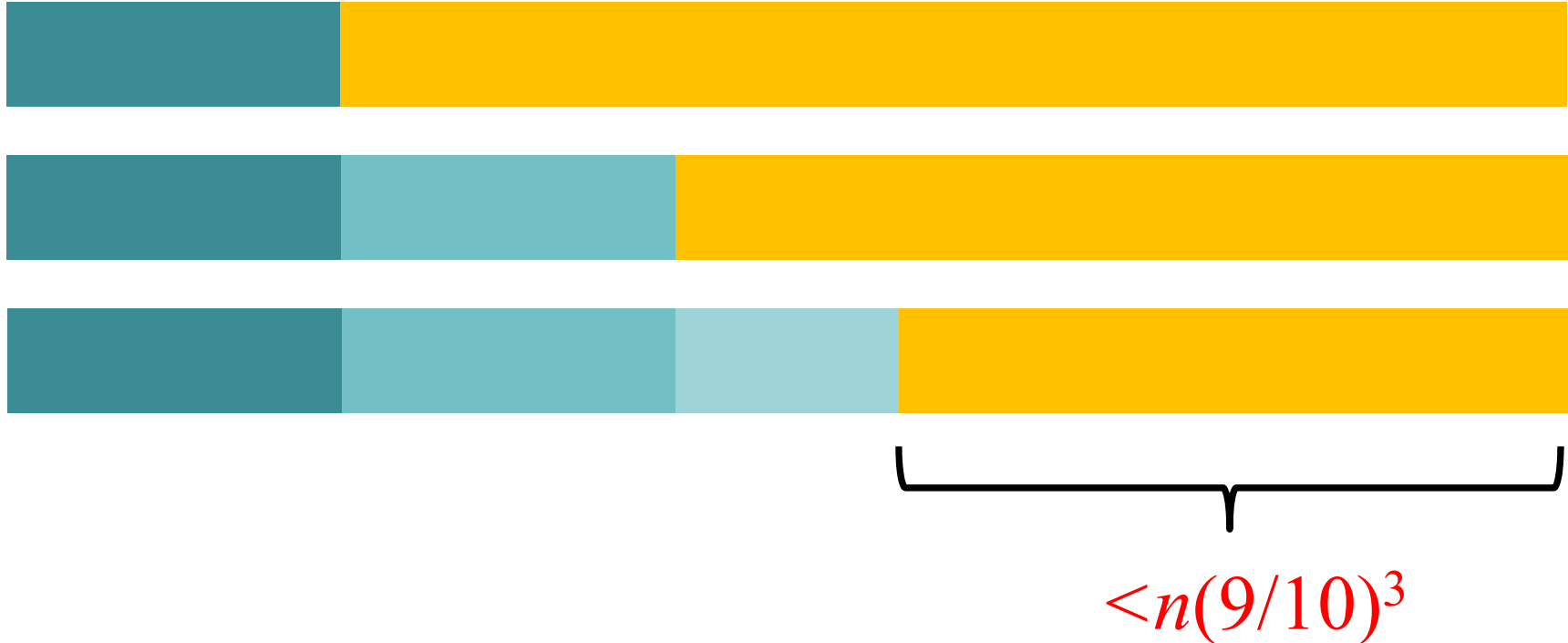
Assume larger part shrinks by at least 9/10 every iteration:



# QuickSort Analysis

---

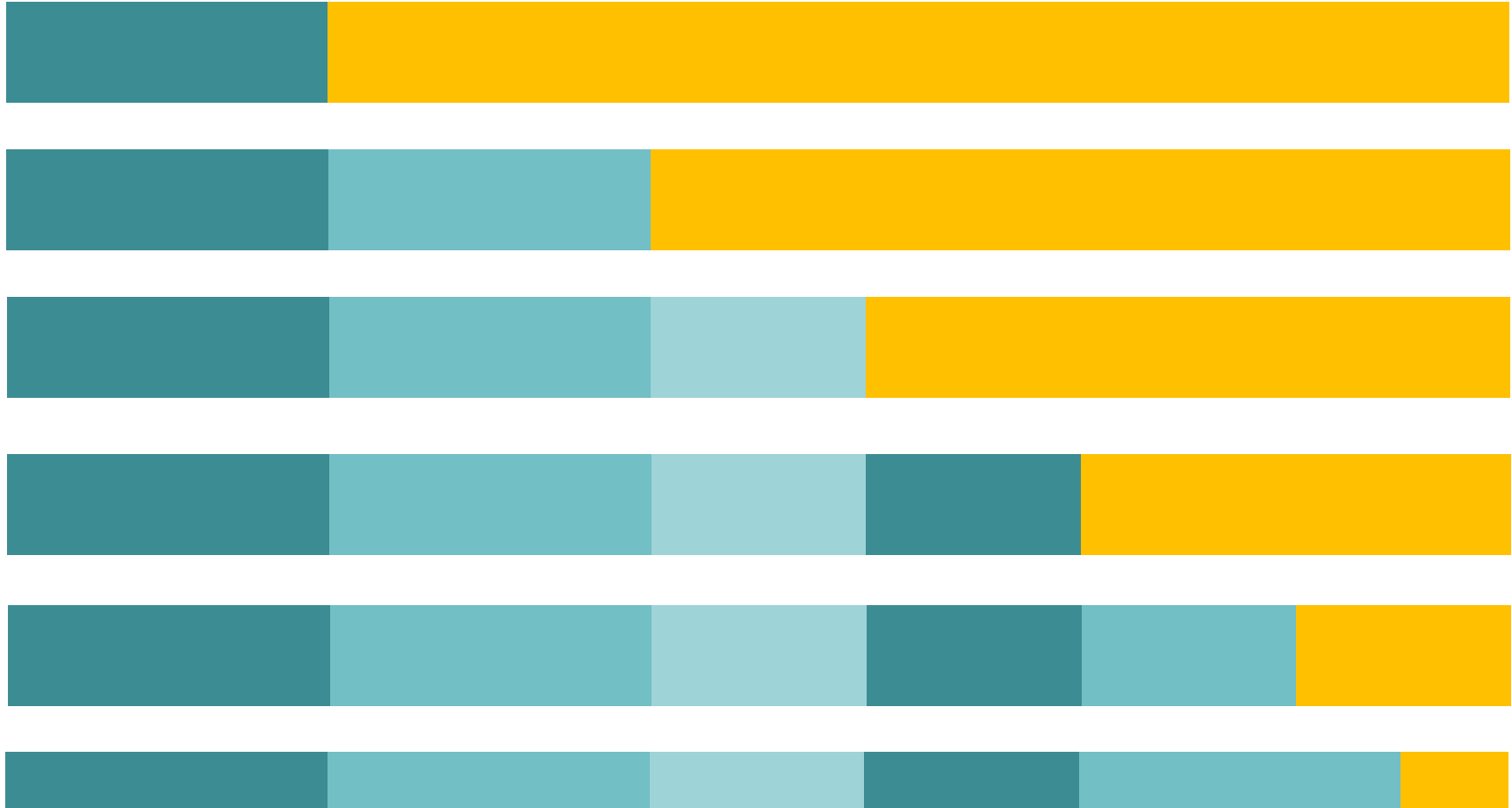
Assume larger part shrinks by at least 9/10 every iteration:



# QuickSort Analysis

---

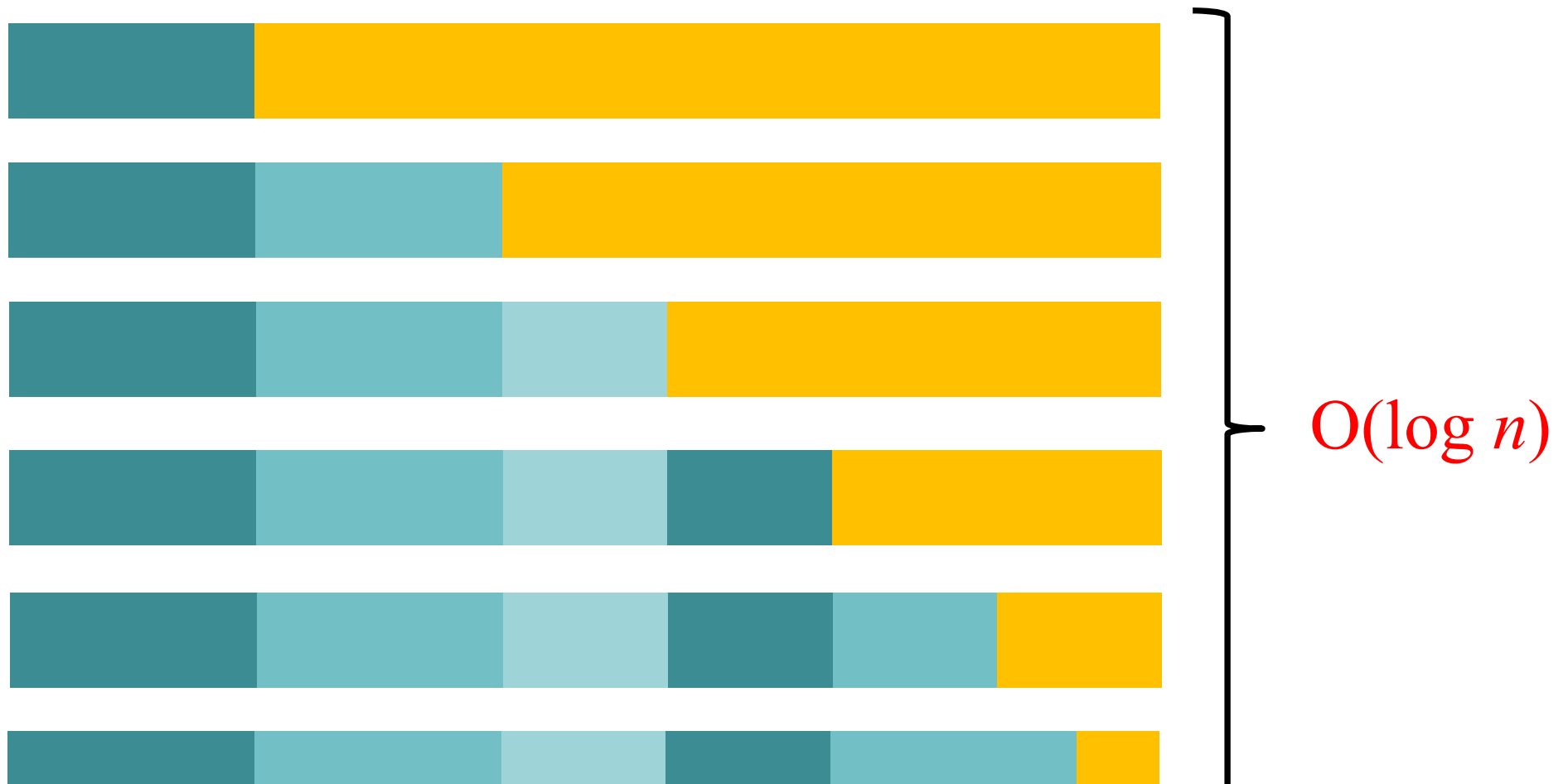
Assume larger part shrinks by at least 9/10 every iteration:



# QuickSort Analysis

---

Assume larger part shrinks by at least 9/10 every iteration:



# QuickSort Summary

---

- If we choose the pivot as  $A[1]$ :
  - Bad performance:  $\Omega(n^2)$
- If we could choose the median element:
  - Good performance:  $O(n \log n)$
- If we could split the array  $(1/10) : (9/10)$ 
  - Good performance:  $O(n \log n)$

# QuickSort

---

**QuickSort**( $A[1..n]$ ,  $n$ )

**if** ( $n==1$ ) **then** return;

**else**

Choose pivot index  $pIndex$ .

$p = \text{partition}(A[1..n], n, pIndex)$

$x = \text{QuickSort}(A[1..p-1], p-1)$

$y = \text{QuickSort}(A[p+1..n], n-p)$

$< x$

$x$

$> x$

# QuickSort

---

## Key Idea:

- Choose the pivot at random.

## Randomized Algorithms:

- Algorithm makes decision based on random coin flips.
- Can “fool” the adversary (who provides bad input)
- Running time is a *random variable*.

# Randomization

---

What is the difference between:

Randomized algorithms

Average-case analysis



# Randomization

---

## Randomized algorithm:

- Algorithm makes random choices
- *For every input*, there is a good probability of success.

## Average-case analysis:

- Algorithm (may be) deterministic
- “Environment” chooses random input
- Some inputs are good, some inputs are bad
- For most inputs, the algorithm succeeds

**QuickSort**(A[1..n], n)

**if** (n == 1) **then** return;

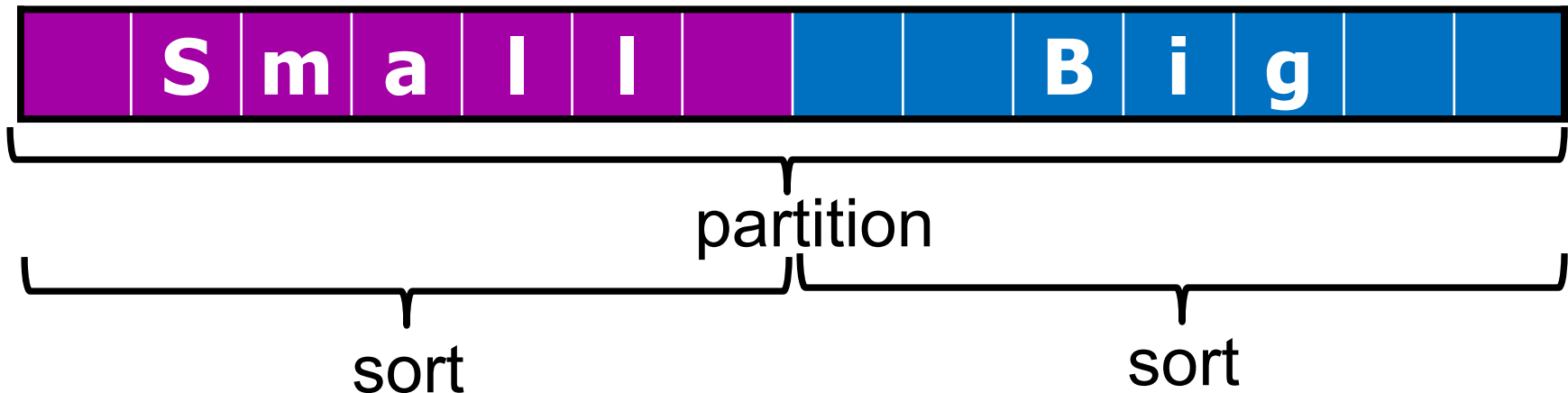
**else**

pIndex = **random**(1, n)

p = **3WayPartition**(A[1..n], n, pindex)

x = **QuickSort**(A[1..p-1], p-1)

y = **QuickSort**(A[p+1..n], n-p)



# Paranoid QuickSort

---

ParanoidQuickSort( $A[1..n]$ ,  $n$ )

**if** ( $n == 1$ ) **then** return;

**else**

**repeat**

$pIndex = \text{random}(1, n)$

$p = \text{partition}(A[1..n], n, pIndex)$

**until**  $p > (1/10)n$  **and**  $p < (9/10)n$

$x = \text{QuickSort}(A[1..p-1], p-1)$

$y = \text{QuickSort}(A[p+1..n], n-p)$

# Paranoid QuickSort

---

Easier to analyze:

- Every time we recurse, we reduce the problem size by at least  $(1/10)$ .
- We have already analyzed that recurrence!

Note: non-paranoid QuickSort works too

- Analysis is a little trickier (but not much).

# Paranoid QuickSort

---

**ParanoidQuickSort**(A[1..n], n)

**if** (n == 1) **then** return;

**else**

**repeat**

pIndex = **random**(1, n)

p = **partition**(A[1..n], n, pIndex)

**until** p > (1/10)n **and** p < (9/10)

x = **QuickSort**(A[1..p-1], p-1)

y = **QuickSort**(A[p+1..n], n-p)

# Paranoid QuickSort

$T(n)$  → **ParanoidQuickSort**(A[1..n], n)

1 → **if** (n == 1) **then** return;  
**else**

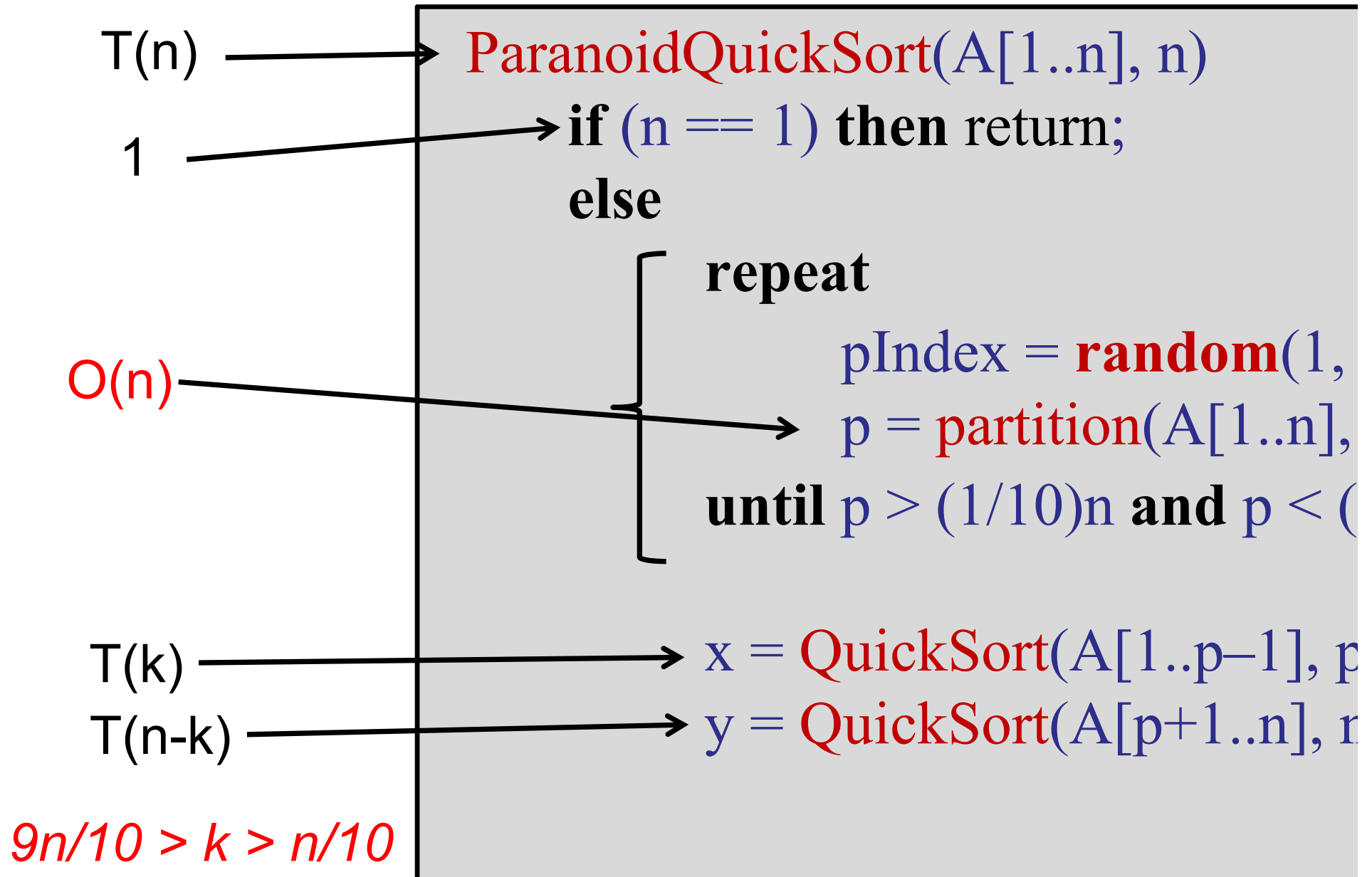
?? → **repeat**  
 $pIndex = \mathbf{random}(1, n)$   
 $p = \mathbf{partition}(A[1..n], pIndex)$   
**until**  $p > (1/10)n$  **and**  $p < (9/10)n$

$T(k)$  →  $x = \mathbf{QuickSort}(A[1..p-1], n)$

$T(n-k)$  →  $y = \mathbf{QuickSort}(A[p+1..n], n)$

$$9n/10 > k > n/10$$

# Paranoid QuickSort



# Paranoid QuickSort

---

Key claim:

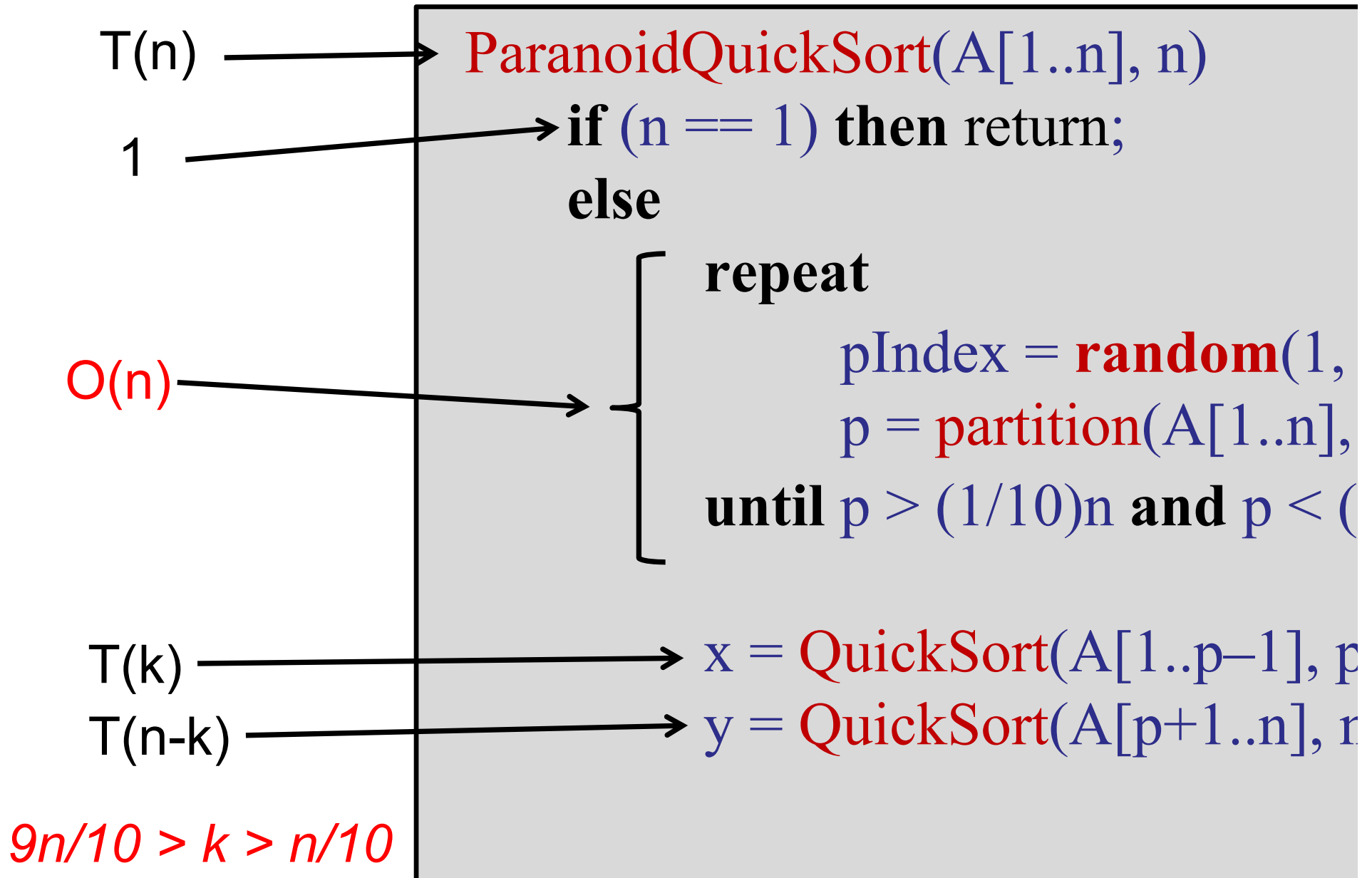
- We only execute the **repeat** loop  $O(1)$  times (in expectation).

Then we know:

$$\begin{aligned} T(n) &\leq T(n/10) + T(9n/10) + n(\text{\# iterations of repeat}) \\ &= O(n \log n) \end{aligned}$$



# Paranoid QuickSort



# Probability Theory

---

# Probability Theory

---

Flipping a coin:

- $\Pr(\text{heads}) = 1/2$
- $\Pr(\text{tails}) = 1/2$

Coin flips are independent:

- $\Pr(\text{heads} \rightarrow \text{heads}) = 1/2 * 1/2 = 1/4$
- $\Pr(\text{heads} \rightarrow \text{tails} \rightarrow \text{heads}) = 1/2 * 1/2 * 1/2 = 1/8$

You flip a coin 8 times. Which is more likely?

- a. 4 heads, followed by 4 tails
- b. 8 heads in a row
- c. Alternating heads, tails, heads, tails, ...
- ✓ d. Same
- e. Incomparable

ARCHIPELAGO

is open

# Probability Theory

---

Flipping a coin:

- $\Pr(\text{heads}) = 1/2$
- $\Pr(\text{tails}) = 1/2$

Set of uniform events ( $e_1, e_2, e_3, \dots, e_k$ ):

- $\Pr(e_1) = 1/k$
- $\Pr(e_2) = 1/k$
- ...
- $\Pr(e_k) = 1/k$

# Independent Events

---

Assume events **A**, **B**:

- Given:  $\Pr(\mathbf{A})$ ,  $\Pr(\mathbf{B})$
- Given: **A** and **B** are independent  
(e.g., unrelated random coin flips)

Then:

- $\Pr(\mathbf{A} \text{ and } \mathbf{B}) = \Pr(\mathbf{A})\Pr(\mathbf{B})$

How many times do you have to flip a coin before it comes up heads?

How many times do you have to flip a coin before it comes up heads?

Poorly defined question...



# Probability Theory

Expected value:

- Weighted average

Example: event **A** has two outcomes:

- $\Pr(\mathbf{A} = 12) = \frac{1}{4}$
- $\Pr(\mathbf{A} = 60) = \frac{3}{4}$

Expected value of A:

$$E[A] = (\frac{1}{4})12 + (\frac{3}{4})60 = 48$$

What is the expected number of times you have to flip a coin before it comes up heads?

# Probability Theory

Flipping a coin:

- $\Pr(\text{heads}) = 1/2$
- $\Pr(\text{tails}) = 1/2$

In two coin flips: I expect one heads.

# Probability Theory

Define event **A**:

**A** = number of heads in two coin flips

In two coin flips: I expect one heads.

– Pr(heads, heads) = $\frac{1}{4}$	2 * $\frac{1}{4}$	=	$\frac{1}{2}$
– Pr(heads, tails) = $\frac{1}{4}$	1 * $\frac{1}{4}$	=	$\frac{1}{4}$
– Pr(tails, heads) = $\frac{1}{4}$	1 * $\frac{1}{4}$	=	$\frac{1}{4}$
– Pr(tails, tails) = $\frac{1}{4}$	0 * $\frac{1}{4}$	=	0
	<hr/>		
			1

Coin flipping game:

Every day you flip two coins. If at least one is heads, you win for the day.

After four days, what is the expected number of winning days?

**ARCHIPELAGO**

is open

# Probability Theory

Flipping a coin:

- $\Pr(\text{heads}) = 1/2$
- $\Pr(\text{tails}) = 1/2$

In two coin flips: I expect one heads.

- If you repeated the experiment many times, on average after two coin flips, you will have one heads.

Goal: calculate expected time of QuickSort

# Probability Theory

Set of outcomes for  $X = (e_1, e_2, e_3, \dots, e_k)$ :

- $\Pr(e_1) = p_1$
- $\Pr(e_2) = p_2$
- ...
- $\Pr(e_k) = p_k$

Expected outcome:

$$E[X] = e_1p_1 + e_2p_2 + \dots + e_kp_k$$

# Probability Theory

## Linearity of Expectation:

- $E[A + B] = E[A] + E[B]$

## Example:

- $A = \# \text{ heads in 2 coin flips: } E[A] = 1$
- $B = \# \text{ heads in 2 coin flips: } E[B] = 1$
- $A + B = \# \text{ heads in 4 coin flips}$

$$E[A+B] = E[A] + E[B] = 1 + 1 = 2$$



# Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

$\mathbf{E}[X]$  = expected number of flips to get one head

Example:  $X = 7$

T T T T T T H

# Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

$$\begin{aligned} \mathbf{E}[X] = & \Pr(\text{heads after 1 flip}) * 1 + \\ & \Pr(\text{heads after 2 flips}) * 2 + \\ & \Pr(\text{heads after 3 flips}) * 3 + \\ & \Pr(\text{heads after 4 flips}) * 4 + \\ & \dots \end{aligned}$$

# Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

$$\begin{aligned} \mathbf{E}[X] = & \Pr(\text{H}) * 1 + \\ & \Pr(\text{T H}) * 2 + \\ & \Pr(\text{T T H}) * 3 + \\ & \Pr(\text{T T T H}) * 4 + \\ & \dots \end{aligned}$$

# Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

$$\begin{aligned} \mathbf{E}[X] = & p(1) + \\ & (1 - p)(p)(2) + \\ & (1 - p)(1 - p)(p)(3) + \\ & (1 - p)(1 - p)(1 - p)(p)(4) + \\ & \dots \end{aligned}$$

# Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

$$\mathbf{E}[X] = (p)(1) + (1 - p)(1 + \mathbf{E}[X])$$

How many more flips to get a head?

**Idea:** If I flip “tails,” the expected number of additional flips to get a “heads” is still  $\mathbf{E}[X]$ !!

# Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

$$\begin{aligned}\mathbf{E}[X] &= (p)(1) + (1 - p)(1 + \mathbf{E}[X]) \\ &= p + 1 - p + 1\mathbf{E}[X] - p\mathbf{E}[X]\end{aligned}$$

# Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

$$\mathbf{E}[X] = (p)(1) + (1 - p)(1 + \mathbf{E}[X])$$

$$= p + 1 - p + 1\mathbf{E}[X] - p\mathbf{E}[X]$$

$$\mathbf{E}[X] - \mathbf{E}[X] + p\mathbf{E}[X] = 1$$

# Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

$$\mathbf{E}[X] = (p)(1) + (1 - p)(1 + \mathbf{E}[X])$$

$$= p + 1 - p + 1\mathbf{E}[X] - p\mathbf{E}[X]$$

$$p\mathbf{E}[X] = 1$$

$$\mathbf{E}[X] = 1/p$$



# Probability Theory

Flipping an (unfair) coin:

- $\Pr(\text{heads}) = p$
- $\Pr(\text{tails}) = (1 - p)$

How many flips to get at least one head?

If  $p = 1/2$ , the expected number of flips to get one head equals:

$$\mathbf{E}[X] = 1/p = 1/1/2 = 2$$

# Paranoid QuickSort

---

**ParanoidQuickSort**(A[1..n], n)

**if** (n == 1) **then** return;

**else**

**repeat**  
    pIndex = **random**(1, n)  
    p = **partition**(A[1..n], n, pIndex)  
**until** p > (1/10)n **and** p < (9/10)

x = **QuickSort**(A[1..p-1], p-1)

y = **QuickSort**(A[p+1..n], n-p)

# QuickSort Partition

---

Remember:

A *pivot* is good if it divides the array into two pieces, each of which is size at least  $n/10$ .



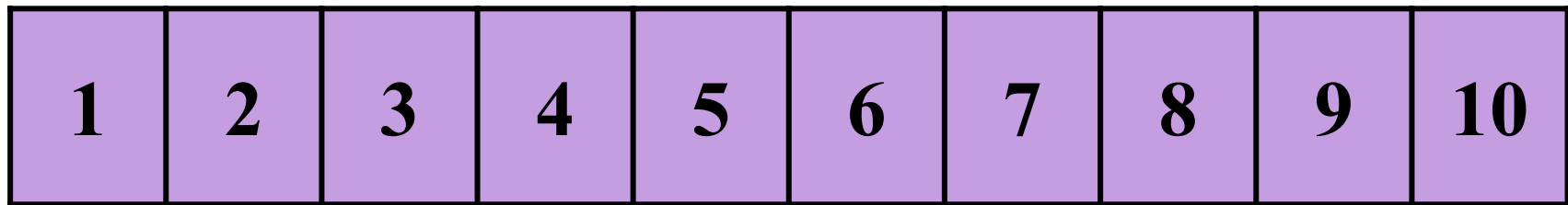
If we choose a pivot at random, what is the probability that it is good?

1.  $1/10$
2.  $2/10$
3.  $8/10$
4.  $1/\log(n)$
5.  $1/n$
6. I have no idea.

# Choosing a Good Pivot

---

Imagine the array divided into 10 pieces:

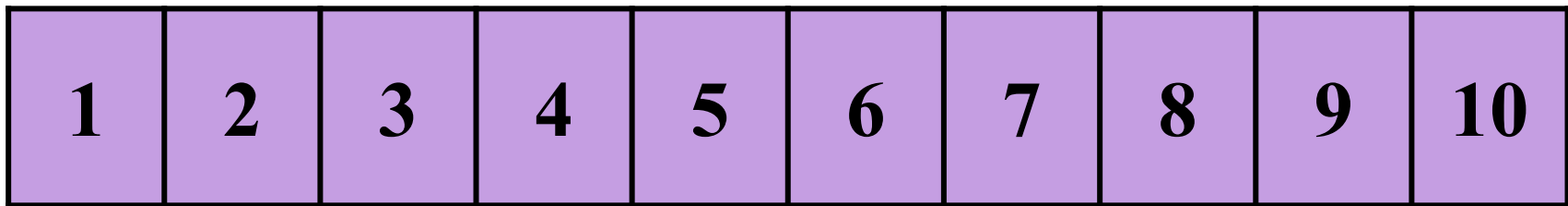


Choose a random point at which to partition.

# Choosing a Good Pivot

---

Imagine the array divided into 10 pieces:



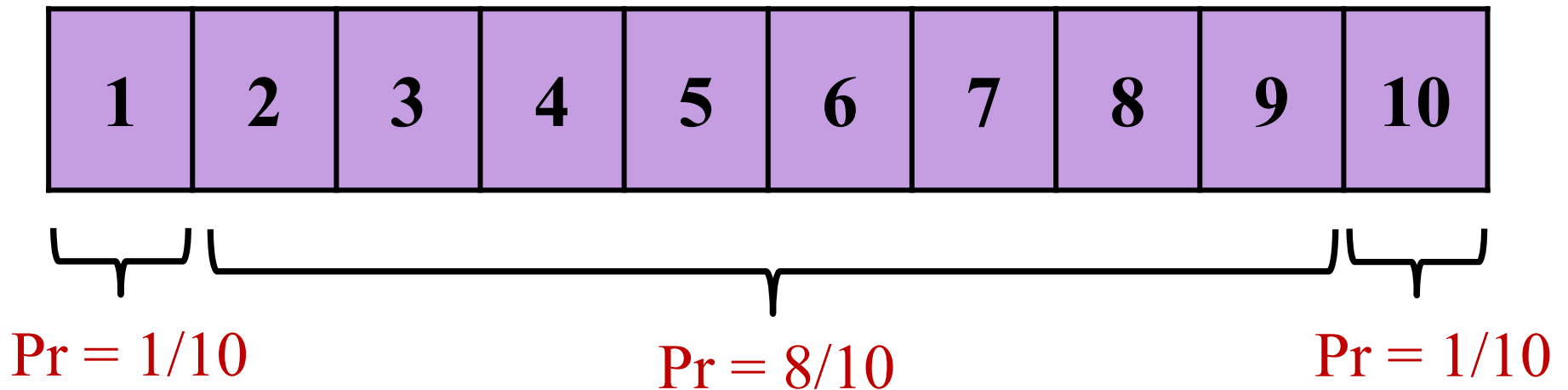
Choose a random point at which to partition.

- 10 possible events
- each occurs with probability  $1/10$

# Choosing a Good Pivot

---

Imagine the array divided into 10 pieces:



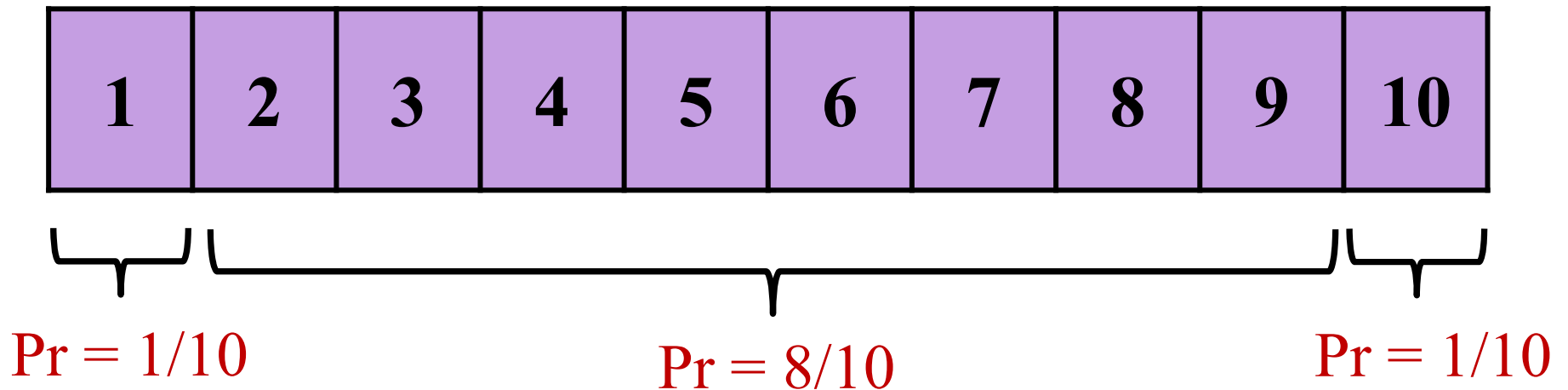
Choose a random point at which to partition.

- 10 possible events
- each occurs with probability  $1/10$

# Choosing a Good Pivot

---

Imagine the array divided into 10 pieces:



Probability of a good pivot:

$$p = 8/10$$

$$(1 - p) = 2/10$$



# Choosing a Good Pivot

---

Probability of a good pivot:

$$p = 8/10$$

$$(1 - p) = 2/10$$

Expected number of times to repeatedly choose a pivot to achieve a good pivot:

$$\mathbf{E}[\# \text{ choices}] = 1/p = 10/8 < 2$$

# Paranoid QuickSort

---

```
ParanoidQuickSort(A[1..n], n)
```

```
  if (n == 1) then return;
```

```
  else
```

```
    { repeat
```

```
      pIndex = random(1,
```

```
      p = partition(A[1..n],
```

```
    until  $p > (1/10)n$  and  $p < ($ 
```

```
    x = QuickSort(A[1..p-1], p
```

```
    y = QuickSort(A[p+1..n], r
```

Expected  
number of  
iterations:  
**10/8**



# Paranoid QuickSort

---

Key claim:

We only execute the **repeat** loop  $< 2$  times  
(in expectation).

Then we know:

$$\begin{aligned}\mathbf{E}[T(n)] &= \mathbf{E}[T(k)] + \mathbf{E}[T(n - k)] + \mathbf{E}[\# \text{ pivot choices}](n) \\ &\leq \mathbf{E}[T(k)] + \mathbf{E}[T(n - k)] + 2n \\ &= O(n \log n)\end{aligned}$$

# Summary

---

## QuickSort:

- Divide-and-Conquer
- Partitioning
- Duplicates
- Choosing a pivot
- Randomization
- Analysis

# QuickSort Choices

---

## How to choose a pivot?

1. Choose the first element of the array.



2. Choose the last element of the array.



3. Choose the middle element in the array.



4. Choose the median element in the array.



5. Choose a random element in the array.



# QuickSort Choices

---

How to choose a pivot?

1. Choose the first element of the array.



2. Choose the last element of the array.



3. Choose the middle element in the array.



Worst-case time:  $\Theta(n^2)$

# QuickSort Choices

---

How to choose a pivot?

Worst-case (expected) time:  $\Theta(n \log n)$

4. Choose the median element in the array.



5. Choose a random element in the array.



# QuickSort Choices

---

How to choose a pivot?

Worst-case (expected) time:  $\Theta(n \log n)$

Simplest option: choose randomly!

4. Choose the median element in the array.



5. Choose a random element in the array.





# QuickSort Choices:

---

How to partition?

1. Copy elements to new array.
2. In-place partitioning. ←

What about duplicate keys?

1. Ignore. They don't exist.
2. Two-pass partitioning.
3. One-pass partitioning. ←

# QuickSort Stability

---

QuickSort is stable if partitioning is stable.

1. In-place partitioning is not *stable*.
2. Extra-memory allows QuickSort to be stable.

# QuickSort Analysis:

---

How to show good performance?

1. If pivot is median: simple recurrence analysis.
2. If pivot is random: *most* of the time, we get a good split.

Use coin flipping analysis and Paranoid variant to show that performance is good.

# QuickSort Optimizations

---

In practice, more efficient to recurse into smaller half first.

- Less to store on call stack.
- Minimizes depth of call stack to deal with small cases first??

# QuickSort Optimizations:

---

Base case?

1. Recurse all the way to single-element arrays.

# QuickSort Optimizations:

---

Base case?

1. Recurse all the way to single-element arrays.
2. Switch to InsertionSort for small arrays.

# QuickSort Optimizations:

---

## Base case?

1. Recurse all the way to single-element arrays.
2. Switch to InsertionSort for small arrays.
3. Halt recursion early, leaving small arrays unsorted. Then perform InsertionSort on entire array.

Relies on fact that  
InsertionSort is very fast on  
*almost sorted arrays!*

# Summary

---

## QuickSort:

- Algorithm basics: divide-and-conquer
- How to partition an array in  $O(n)$  time.
- How to choose a good pivot.
- Paranoid QuickSort.
- Randomized analysis.



# Today: Sorting, Part III

---

## QuickSort:

- Duplicates
- Choosing a pivot
- Randomization
- Analysis

## Selection and Order Statistics

- QuickSelect

# Order Statistics

---

Find  $k^{th}$  smallest element in an *unsorted* array:

$x_{10}$	$x_2$	$x_4$	$x_1$	$x_5$	$x_3$	$x_7$	$x_8$	$x_9$	$x_6$
----------	-------	-------	-------	-------	-------	-------	-------	-------	-------

E.g.: Find the median ( $k = n/2$ )

Find the 7<sup>th</sup> element ( $k = 7$ )

# Order Statistics

---

Find  $k^{th}$  smallest element in an *unsorted* array:

$x_{10}$	$x_2$	$x_4$	$x_1$	$x_5$	$x_3$	$x_7$	$x_8$	$x_9$	$x_6$
----------	-------	-------	-------	-------	-------	-------	-------	-------	-------

Option 1:

- Sort the array.
- Count to element number  $k$ .

Running time:  $O(n \log n)$

# Order Statistics

---

Find  $k^{th}$  smallest element in an *unsorted* array:

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$	$x_{10}$
-------	-------	-------	-------	-------	-------	-------	-------	-------	----------

Option 1:

- Sort the array.
- Count to element number  $k$ .

Running time:  $O(n \log n)$

# Order Statistics

---

Find  $k^{th}$  smallest element in an *unsorted* array:

$x_{10}$	$x_2$	$x_4$	$x_1$	$x_5$	$x_3$	$x_7$	$x_8$	$x_9$	$x_6$
----------	-------	-------	-------	-------	-------	-------	-------	-------	-------

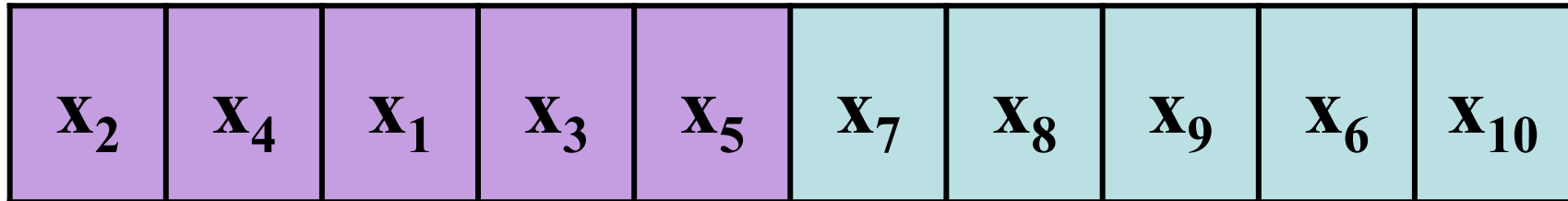
Option 2:

- Only do the minimum amount of sorting necessary

# Order Statistics

---

Key Idea: partition the array



Now continue searching in the correct half.

E.g.: Partition around  $x_5$  and recursively search for  $x_3$  in left half.

# Order Statistics

---

Example: search for 5<sup>th</sup> element

9	22	13	17	5	3	100	6	19	8
---	----	----	----	---	---	-----	---	----	---

# Order Statistics

---

Example: search for 5<sup>th</sup> element

9	22	13	17	5	3	100	6	19	8
---	----	----	----	---	---	-----	---	----	---

Partition around random pivot: 17

9	8	13	5	3	6	17	100	19	22
---	---	----	---	---	---	----	-----	----	----

1      2      3      4      5      6      7      8      9      10



# Order Statistics

---

Example: search for 5<sup>th</sup> element

9	8	13	5	3	6	17	100	19	22
1	2	3	4	5	6	7	8	9	10

Search for 5<sup>th</sup> element in left half.

9	8	13	5	3	6				
1	2	3	4	5	6				

# Order Statistics

---

Example: search for 5<sup>th</sup> element

9	8	13	5	3	6				
---	---	----	---	---	---	--	--	--	--

Partition around random pivot: 8

6	3	5	8	13	9				
---	---	---	---	----	---	--	--	--	--

1      2      3      4      5      6

# Order Statistics

---

Example: search for 5<sup>th</sup> element

9	8	13	5	3	6				
---	---	----	---	---	---	--	--	--	--

Search for:  $5 - 4 = 1$  in right half

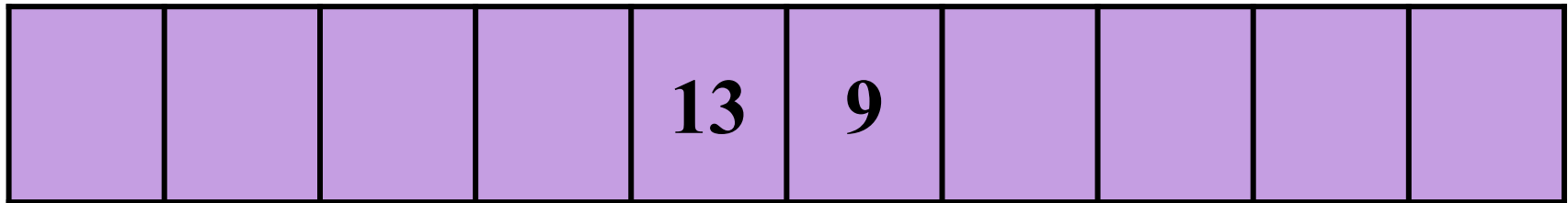
6	3	5	8	13	9				
---	---	---	---	----	---	--	--	--	--

1    2    3    4    5    6

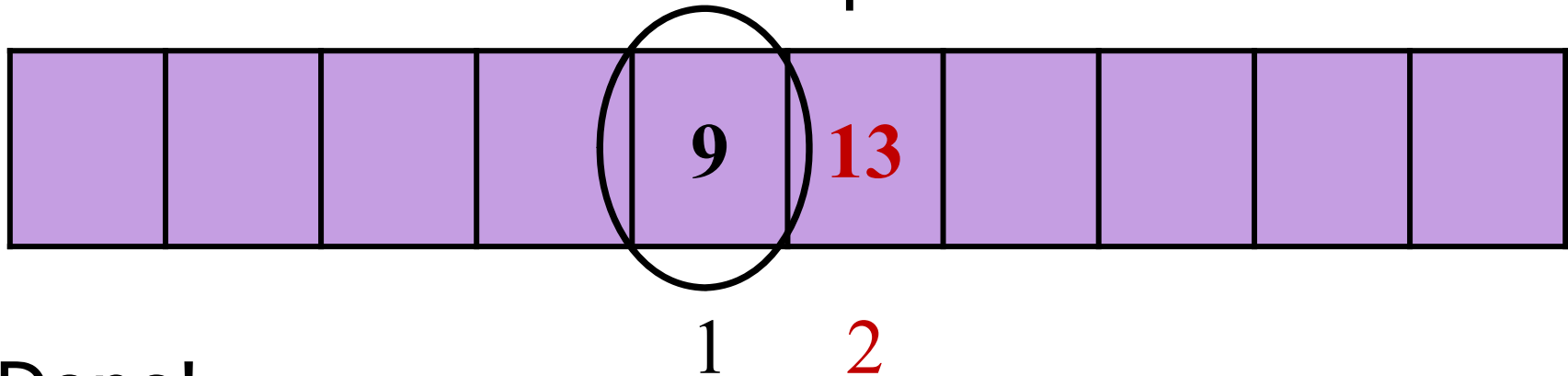
# Order Statistics

---

Search for:  $5 - 4 = 1$  in right half



Partition around random pivot: 13



Done!

# Finding the $k^{\text{th}}$ smallest element

---

**Select**(A[1..n], n, k)

**if** (n == 1) **then return** A[1];

**else** Choose random pivot index pIndex.

p = **partition**(A[1..n], n, pIndex)

**if** (k == p) **then return** A[p];

**else if** (k < p) **then**

**return** **Select**(A[1..p-1], k)

**else if** (k > p) **then**

**return** **Select**(A[p+1], k - p)

# Order Statistics

---

Recurring right  
and left are not  
exactly the same.

Example: search for 5<sup>th</sup> element

9	22	13	17	5	3	100	6	19	8
---	----	----	----	---	---	-----	---	----	---

Partition around random pivot: 17

9	8	13	5	3	6	17	100	19	22
---	---	----	---	---	---	----	-----	----	----

1    2    3    4    5    6    7    8    9    10

Search for 5<sup>th</sup> element on the left.

# Order Statistics

---

Recurring right  
and left are not  
exactly the same.

Example: search for 8<sup>th</sup> element

9	22	13	17	5	3	100	6	19	8
---	----	----	----	---	---	-----	---	----	---

Partition around random pivot: 8

5	6	3	8	17	13	100	22	19	9
---	---	---	---	----	----	-----	----	----	---

1    2    3    4    5    6    7    8    9    10

Search for 4<sup>th</sup> element on the right.

# Order Statistics

---

Recurring right  
and left are not  
exactly the same.

Example: search for 4<sup>th</sup> element

9	22	13	17	5	3	100	6	19	8
---	----	----	----	---	---	-----	---	----	---

Partition around random pivot: 8

5	6	3	8	17	13	100	22	19	9
---	---	---	---	----	----	-----	----	----	---

1    2    3    4    5    6    7    8    9    10

Return 8.



# Finding the $k^{\text{th}}$ smallest element

---

**Select**(A[1..n], n, k)

**if** (n == 1) **then return** A[1];

**else** Choose random pivot index pIndex.

p = **partition**(A[1..n], n, pIndex)

**if** (k == p) **then return** A[p];

**else if** (k < p) **then**

**return** **Select**(A[1..p-1], k)

**else if** (k > p) **then**

**return** **Select**(A[p+1], k - p)

# Finding the $k^{\text{th}}$ smallest element

---

Key point:

- Only recurse *once*!
- Why not recurse twice?
  - Does not help---the correct element is on one side.
  - You do not need to sort both sides!
  - Makes it run a lot faster.

# Analysis

---

## Paranoid-Select:

Repeatedly partition until at least  $n/10$  in each half of the partition.

**repeat**

$p = \text{partition}(A[1..n], n, p\text{Index})$

**until**  $(p > n/10)$  and  $(p < 9n/10)$

# Analysis

---

Paranoid-Select:

Repeatedly partition until at least  $n/10$  in each half of the partition.

Recurrence:

$$\mathbf{E}[T(n)] \leq \mathbf{E}[T(9n/10)] + \mathbf{E}[\# \text{ partitions}](n)$$

cost of partitioning



# Analysis

---

Paranoid-Select:

Repeatedly partition until at least  $n/10$  in each half of the partition.

Recurrence:

$$\begin{aligned}\mathbf{E}[T(n)] &\leq \mathbf{E}[T(9n/10)] + \mathbf{E}[\# \text{ partitions}](n) \\ &\leq \mathbf{E}[T(9n/10)] + 2n\end{aligned}$$

# Analysis

---

## Paranoid-Select:

Repeatedly partition until at least  $n/10$  in each half of the partition.

## Recurrence:

$$\begin{aligned}\mathbf{E}[T(n)] &\leq \mathbf{E}[\# \text{ partitions}](n) + \mathbf{E}[T(9n/10)] \\ &\leq 2n + \mathbf{E}[T(9n/10)] \\ &\leq 2n + 2n (9/10) + (9/10) \mathbf{E}[T(9n/10)] \\ &\leq 2n + 2n (9/10) + 2n (9/10)^2 + \dots\end{aligned}$$

# Analysis

---

Paranoid-Select:

Repeatedly partition until at least  $n/10$  in each half of the partition.

Recurrence:

$$\begin{aligned}\mathbf{E}[T(n)] &\leq \mathbf{E}[T(9n/10)] + \mathbf{E}[\# \text{ partitions}](n) \\ &\leq \mathbf{E}[T(9n/10)] + 2n \\ &\leq O(n)\end{aligned}$$

$$\textit{Recurrence: } T(n) = T(n/2) + O(n)$$

# Today: Sorting, Part III

---

## QuickSort:

- Duplicates
- Choosing a pivot
- Randomization
- Analysis

## Selection and Order Statistics

- QuickSelect



# Summary

---

QuickSort:  $O(n \log n)$

- Partitioning an array
- Deterministic QuickSort
- Paranoid Quicksort

Order Statistics:  $O(n)$

- Finding the  $k^{\text{th}}$  smallest element in an array.
- Key idea: partition
- Paranoid Select

# Today: Sorting, Part III

---

## QuickSort:

- Duplicates
- Choosing a pivot
- Randomization
- Analysis

## Selection and Order Statistics

- QuickSelect

Next time:

Trees!

# Summary

---

QuickSort:  $O(n \log n)$

- Partitioning an array
- Deterministic QuickSort
- Paranoid Quicksort

Order Statistics:  $O(n)$

- Finding the  $k^{\text{th}}$  smallest element in an array.
- Key idea: partition
- Paranoid Select

# Today: Sorting, Part III

---

## QuickSort:

- Duplicates
- Choosing a pivot
- Randomization
- Analysis

## Selection and Order Statistics

- QuickSelect

Next time:

Trees!