CS2040S
Data Structures and Algorithms

Welcome!

Last Time: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

Today: more sorting!

QuickSort:

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

QuickSort

History:

- Invented by C.A.R. Hoare in 1960
  - Turing Award: 1980
- Visiting student at Moscow State University
- Used for machine translation (English/Russian)

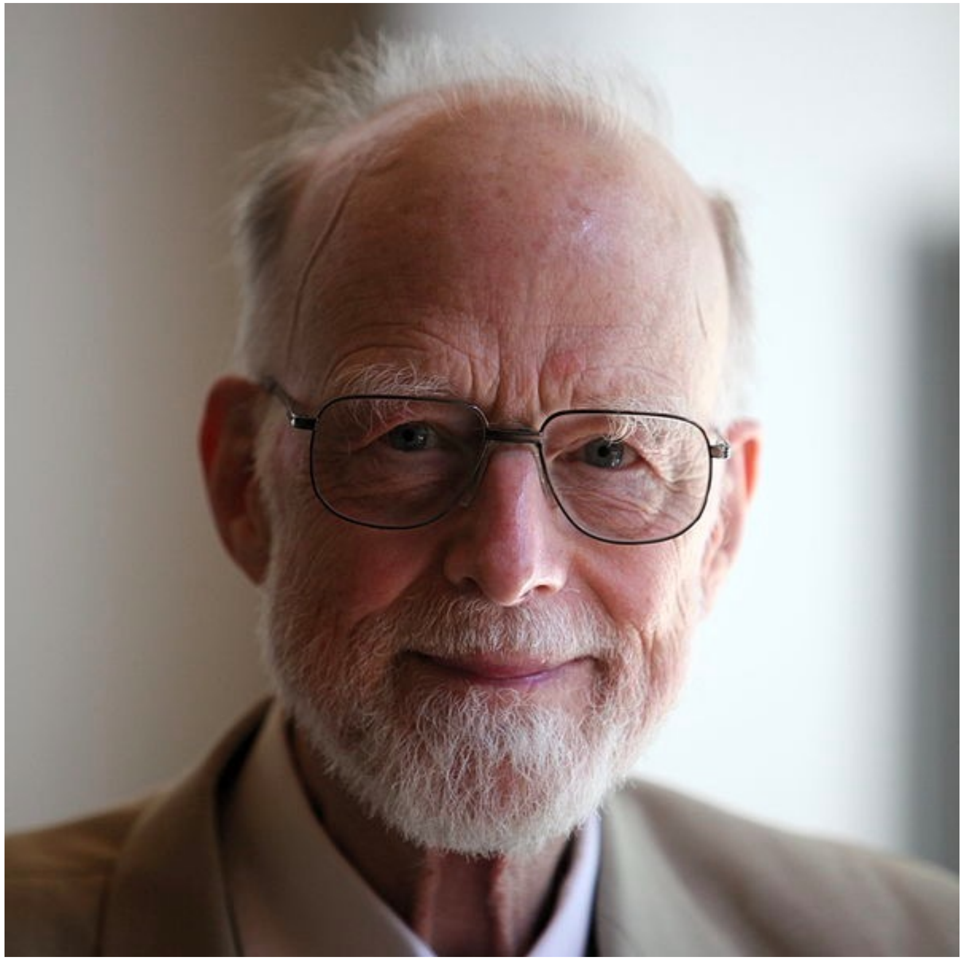


Photo: Wikimedia Commons (Rama)

QuickSort

History:

- Invented by C.A.R. Hoare in 1960
- Used for machine translation (English/Russian)

In practice:

- Very fast
- Many optimizations
- In-place (i.e., no extra space needed)
- Good caching performance
- Good parallelization

QuickSort Today

1960: Invented by Hoare

1979: Adopted everywhere (e.g., Unix qsort)

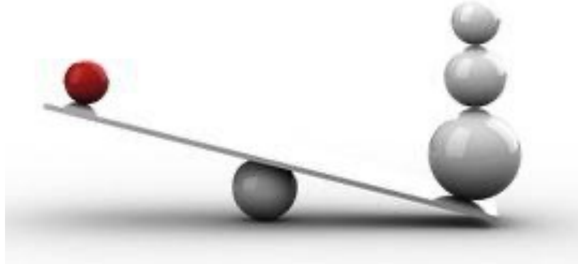
1993: Bentley & McIlroy improvements

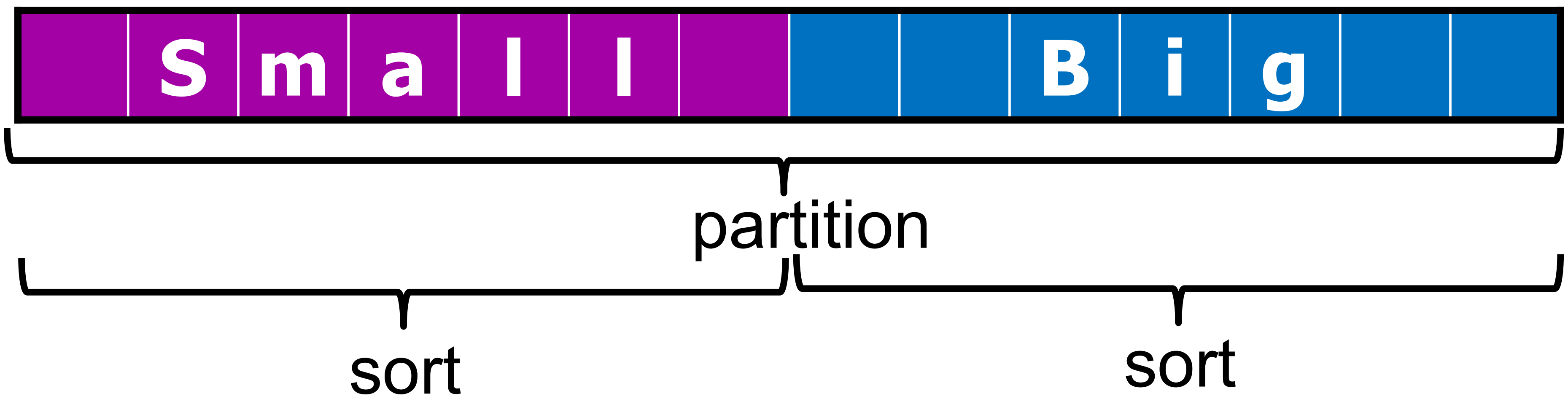
2009: Vladimir Yaroslavskiy: Dual Pivot QS

2023: ??

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

22	1	6	40	32	10	18	4	50

low
< 22

high
> 22

Move until it's bigger than the pivot

Move until it's less than the pivot

Partitioning an Array

Example: partition around 22

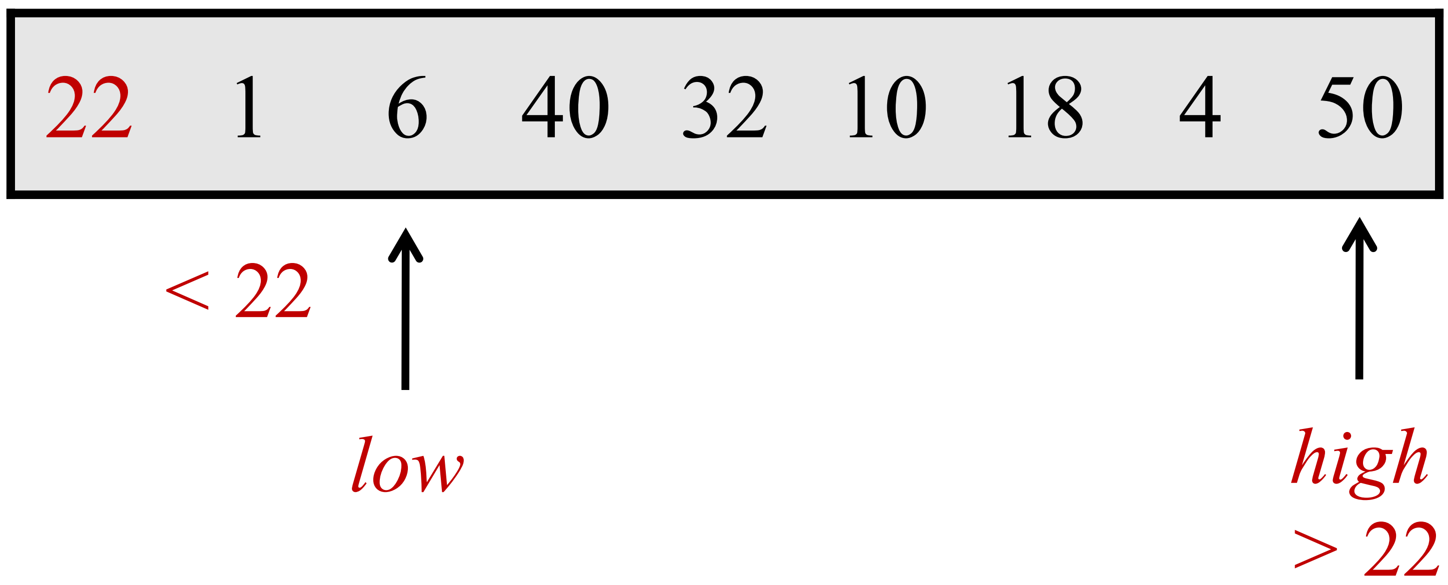
22	1	6	40	32	10	18	4	50

low
< 22

high
> 22

Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22

22	1	6	40	32	10	18	4	50

< 22

↑ *low*

↑ *high*

> 22

Partitioning an Array

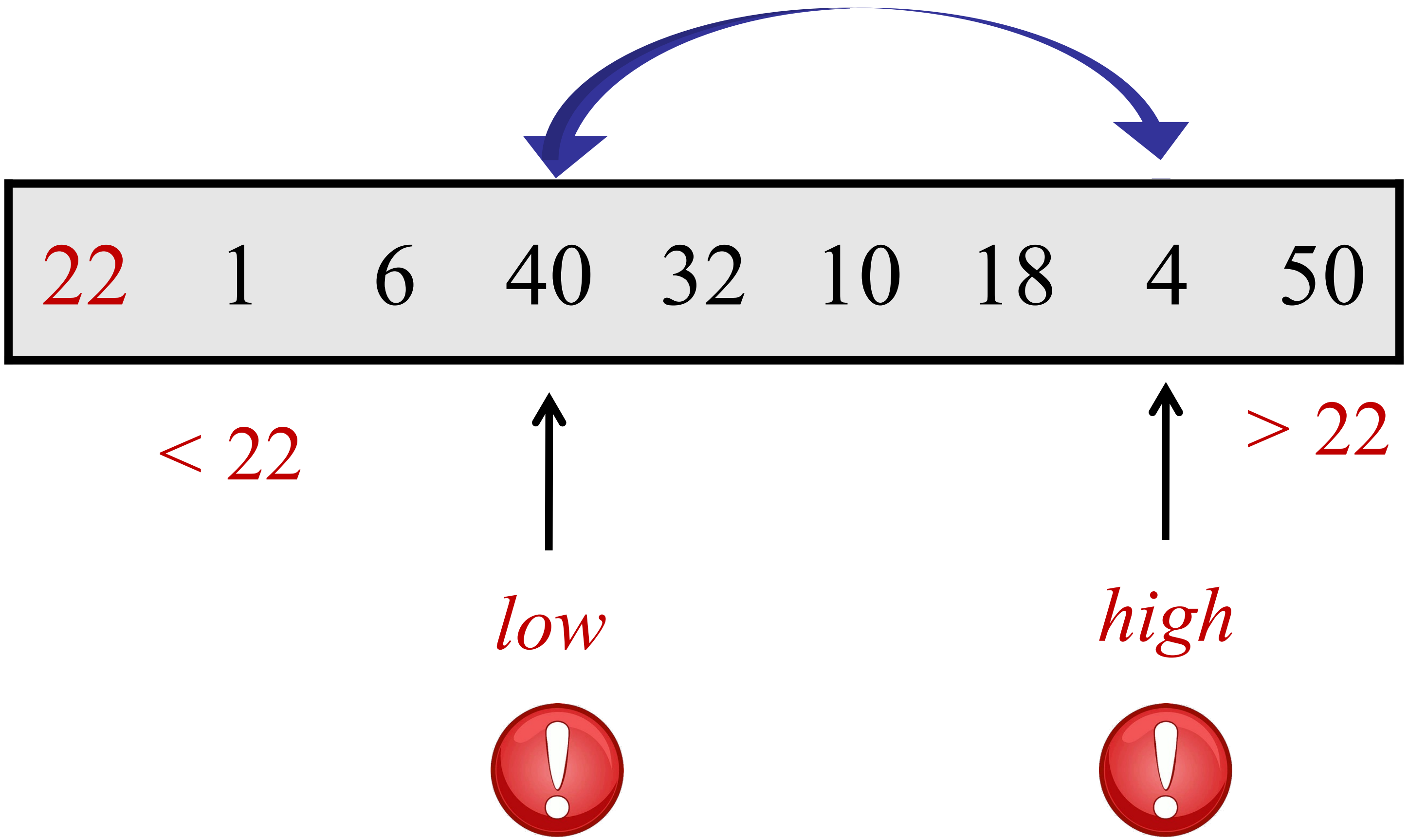
Example: partition around 22

22	1	6	40	32	10	18	4	50

< 22   ↑ *low*   ↑ *high*   > 22

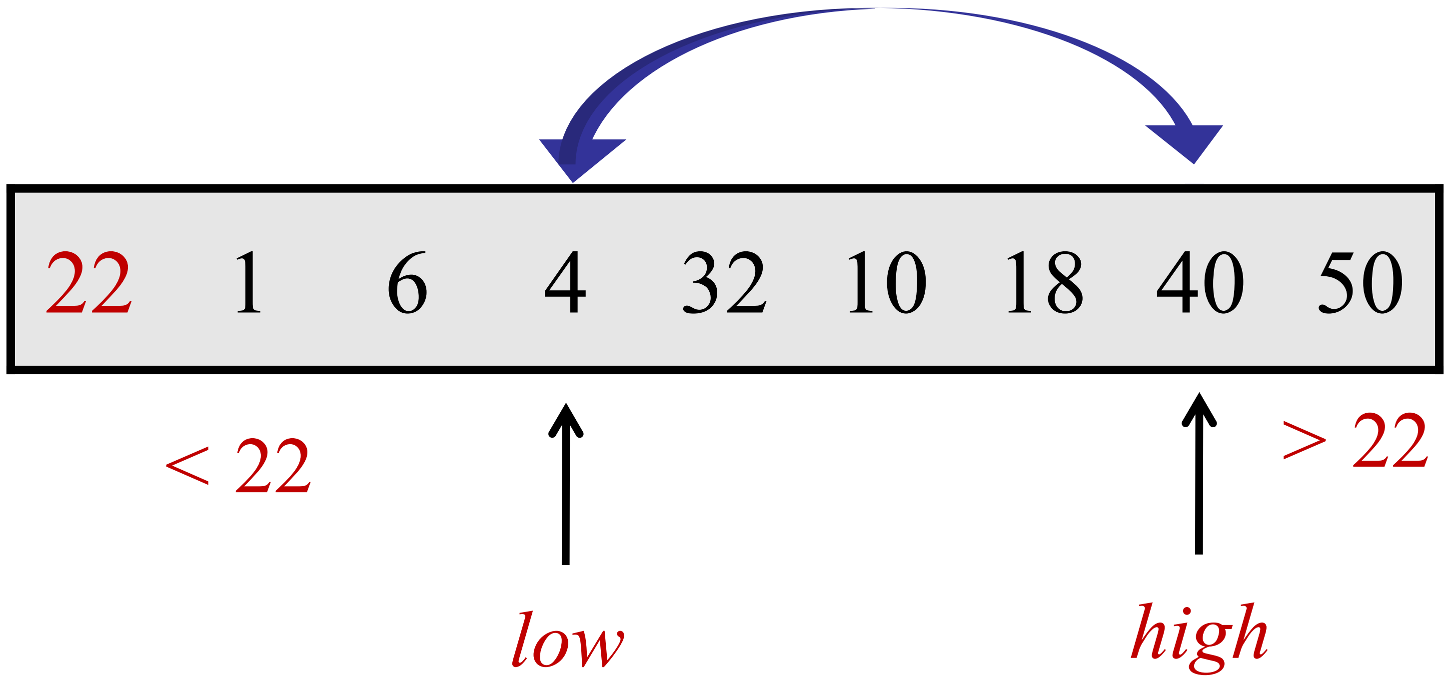
Partitioning an Array

Example: partition around 22



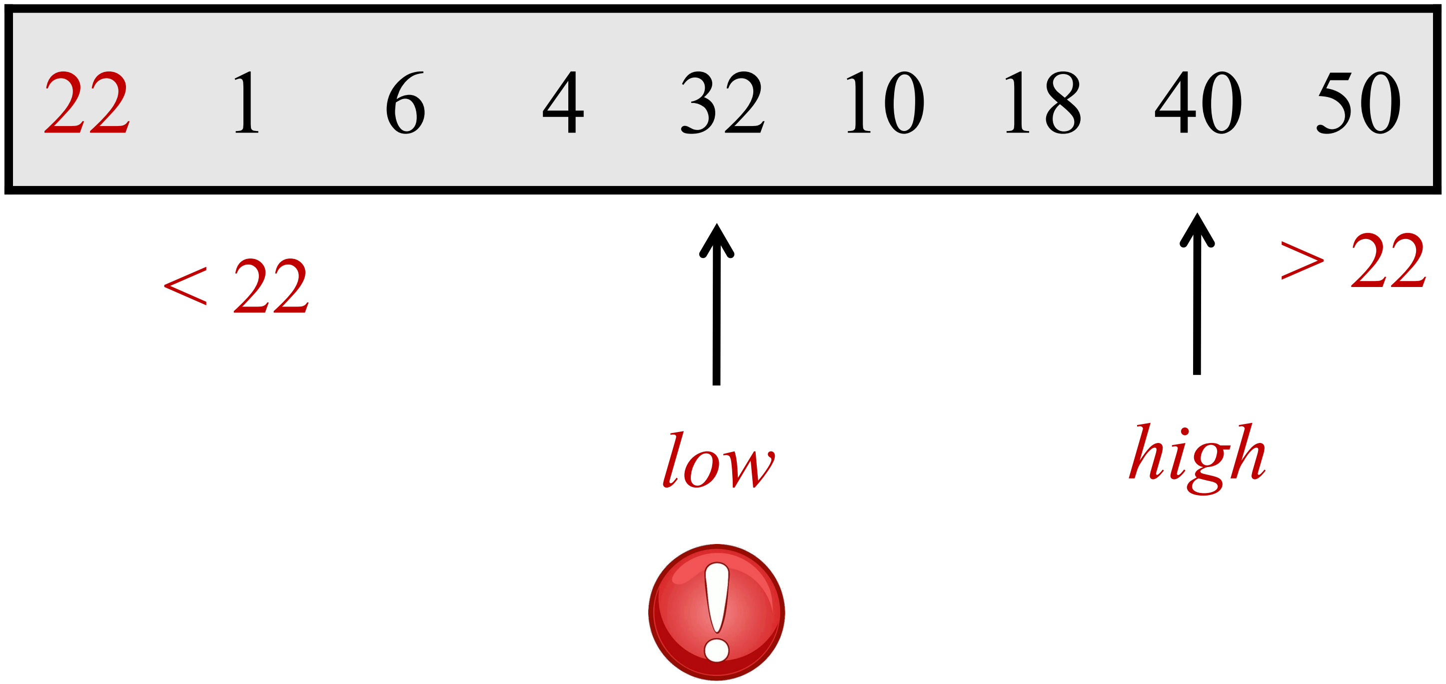
Partitioning an Array

Example: partition around 22



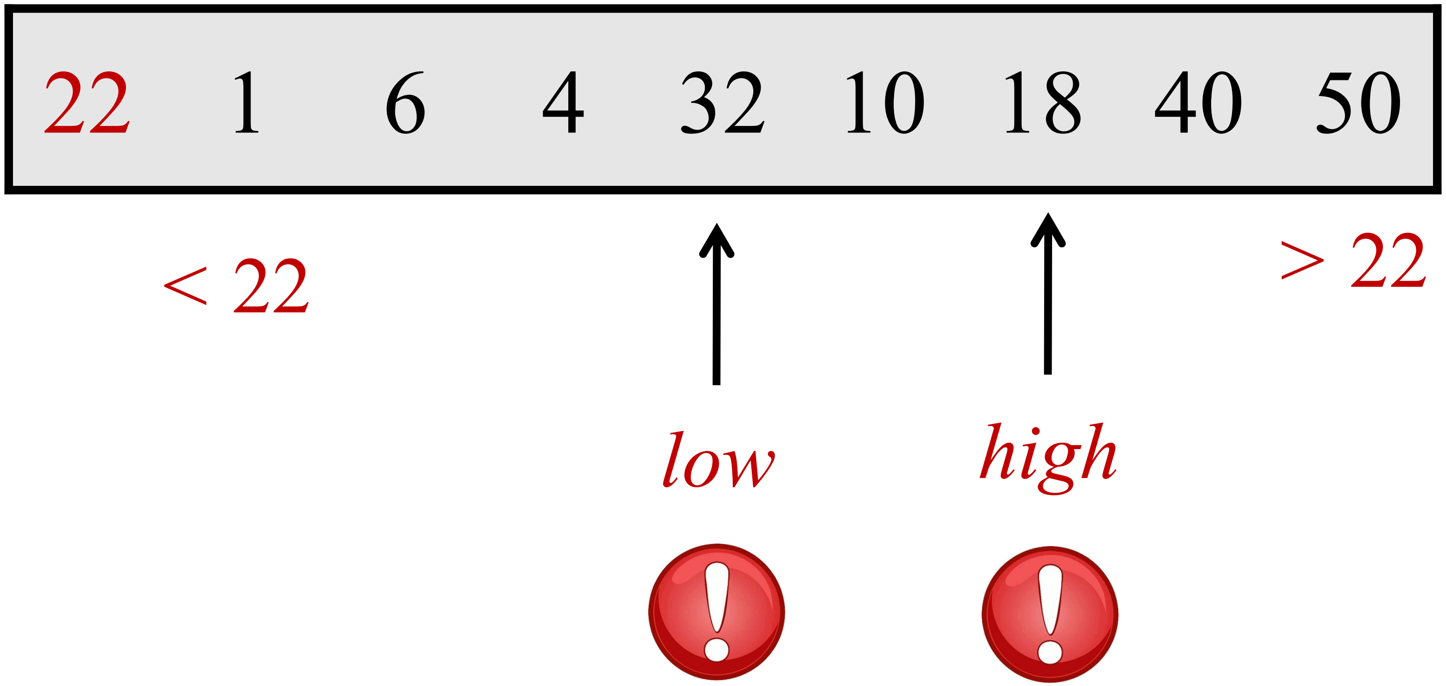
Partitioning an Array

Example: partition around 22



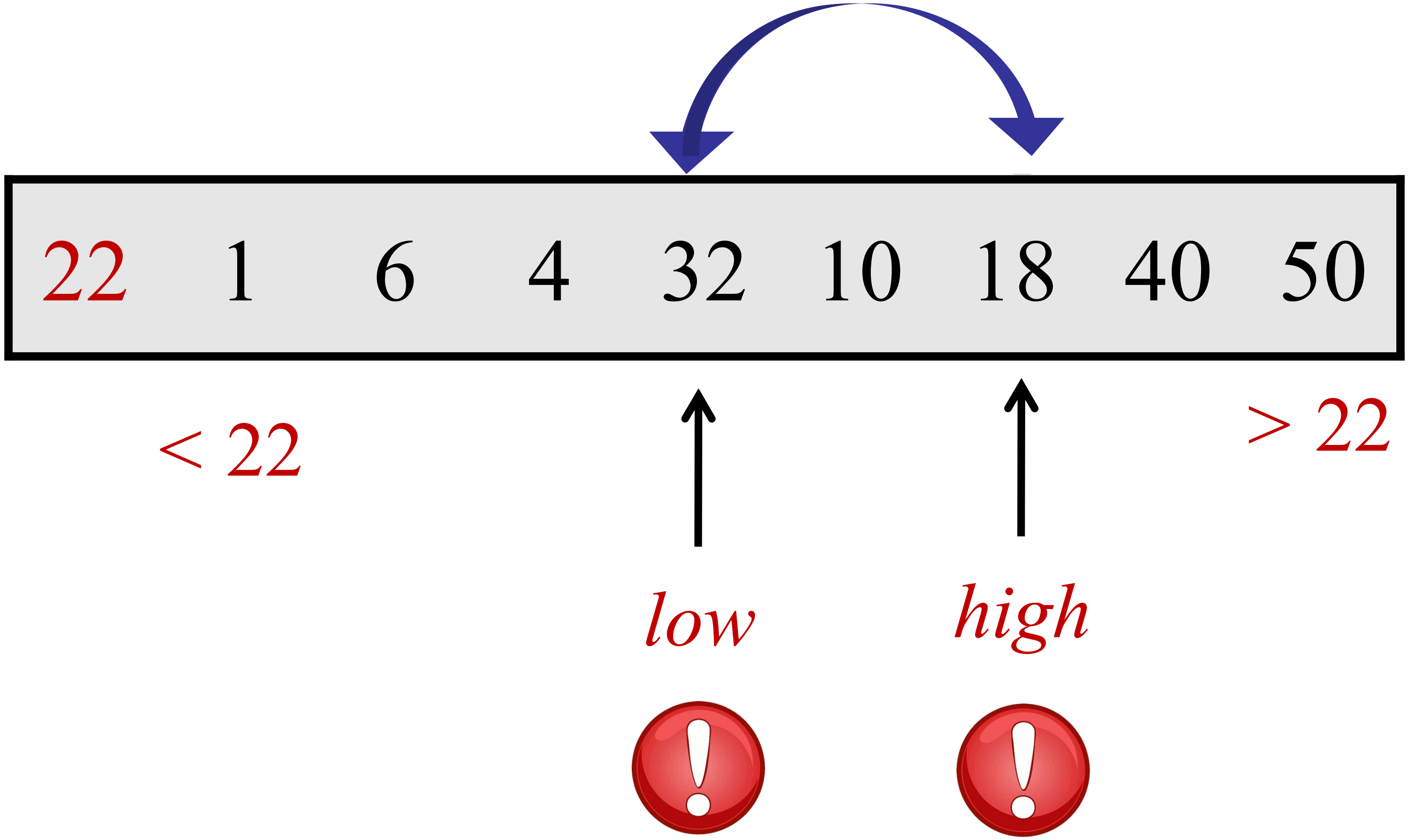
Partitioning an Array

Example: partition around 22



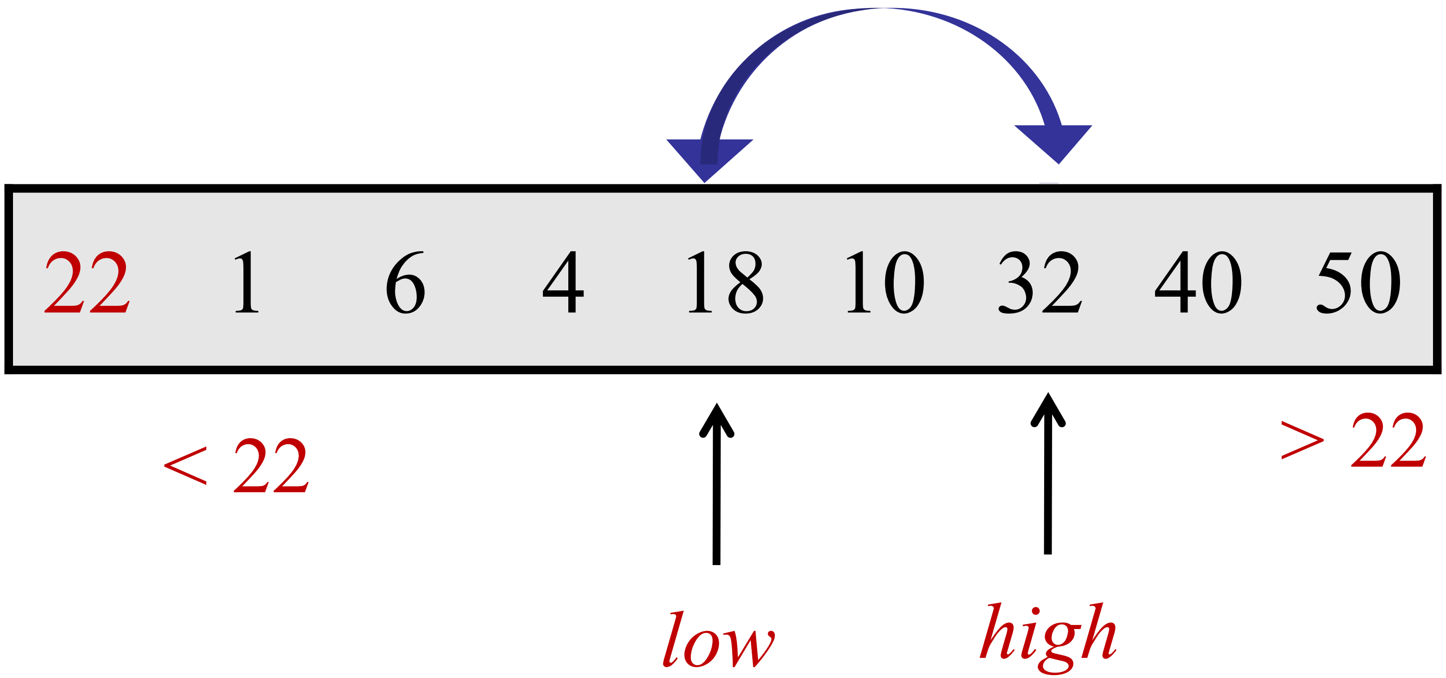
Partitioning an Array

Example: partition around 22



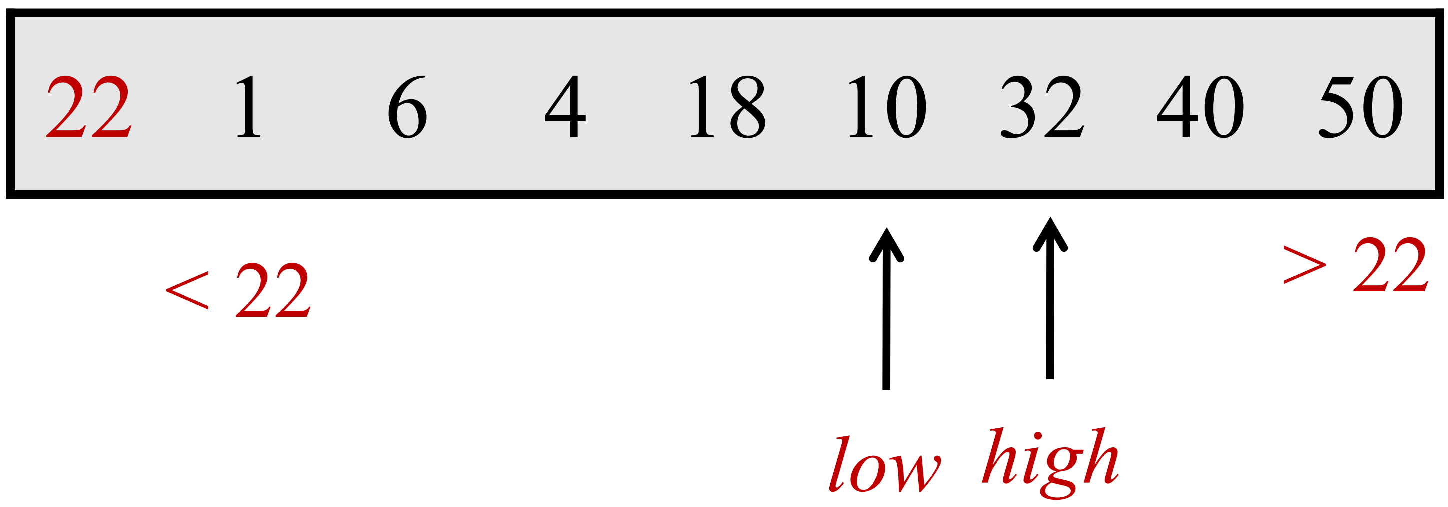
Partitioning an Array

Example: partition around 22



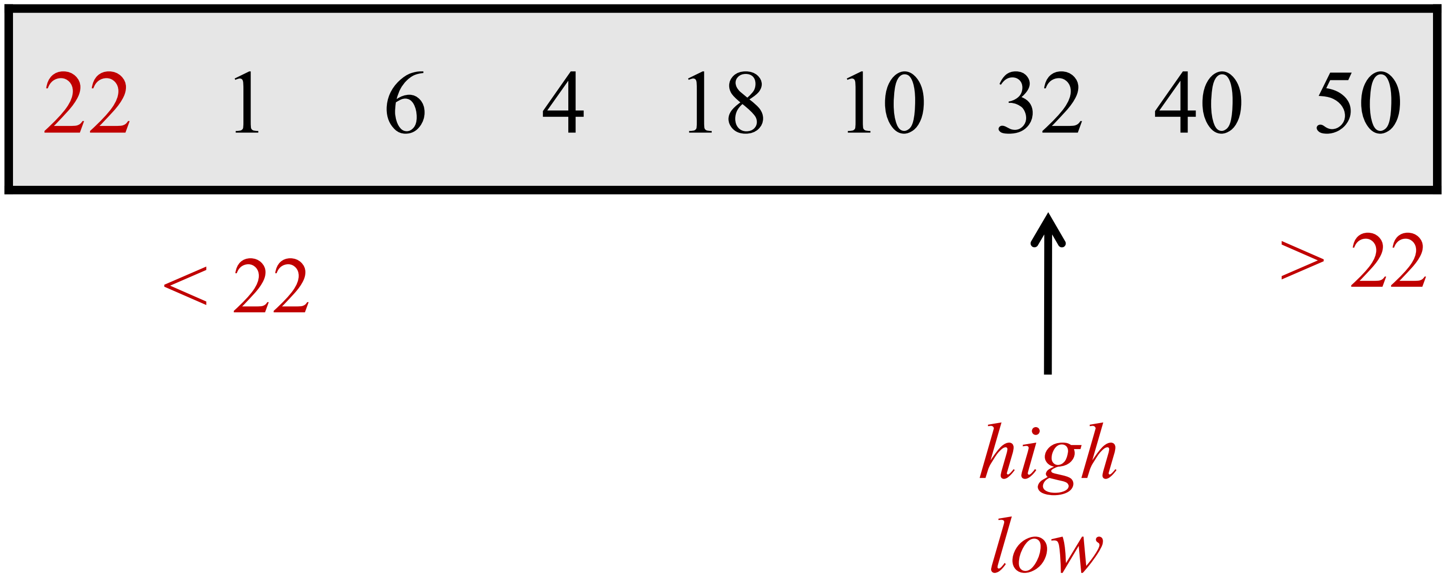
Partitioning an Array

Example: partition around 22



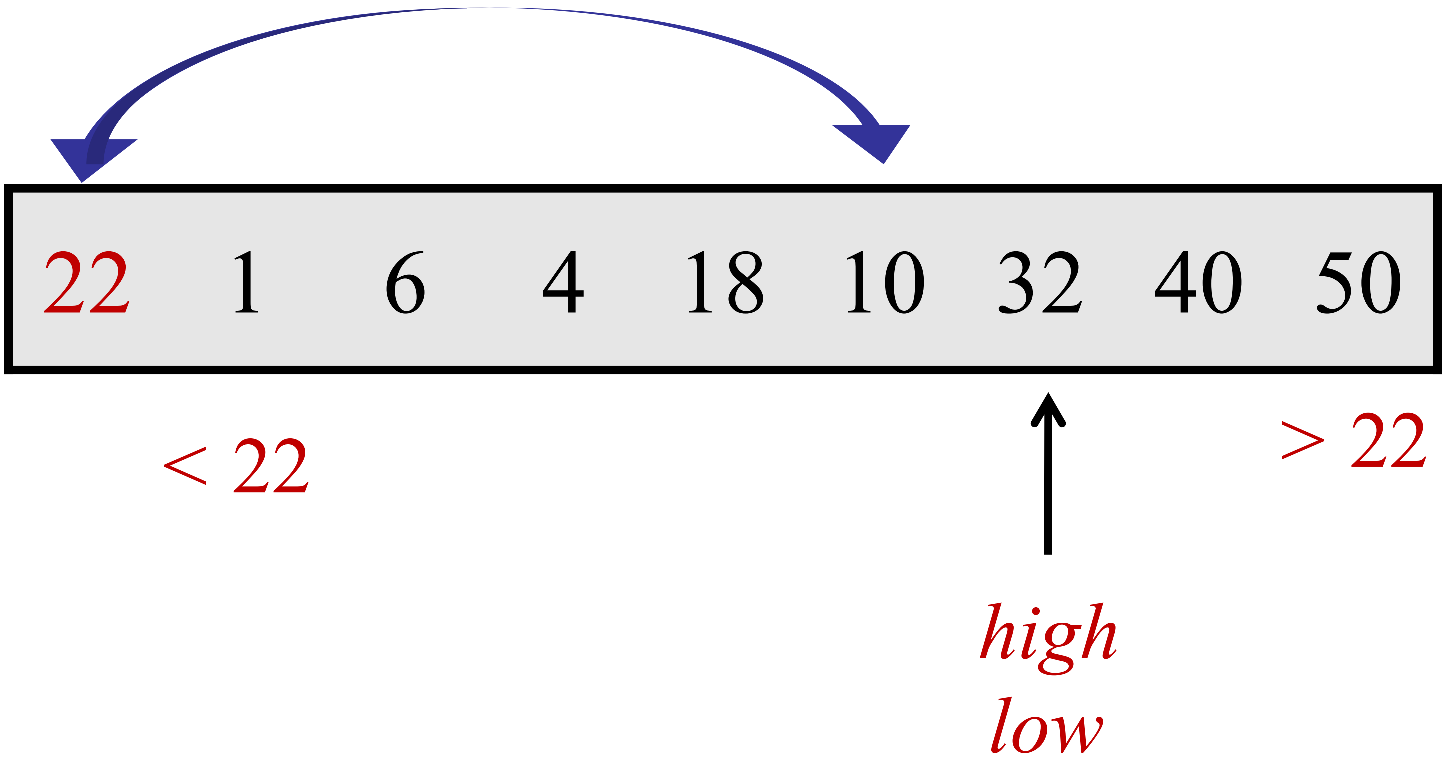
Partitioning an Array

Example: partition around 22



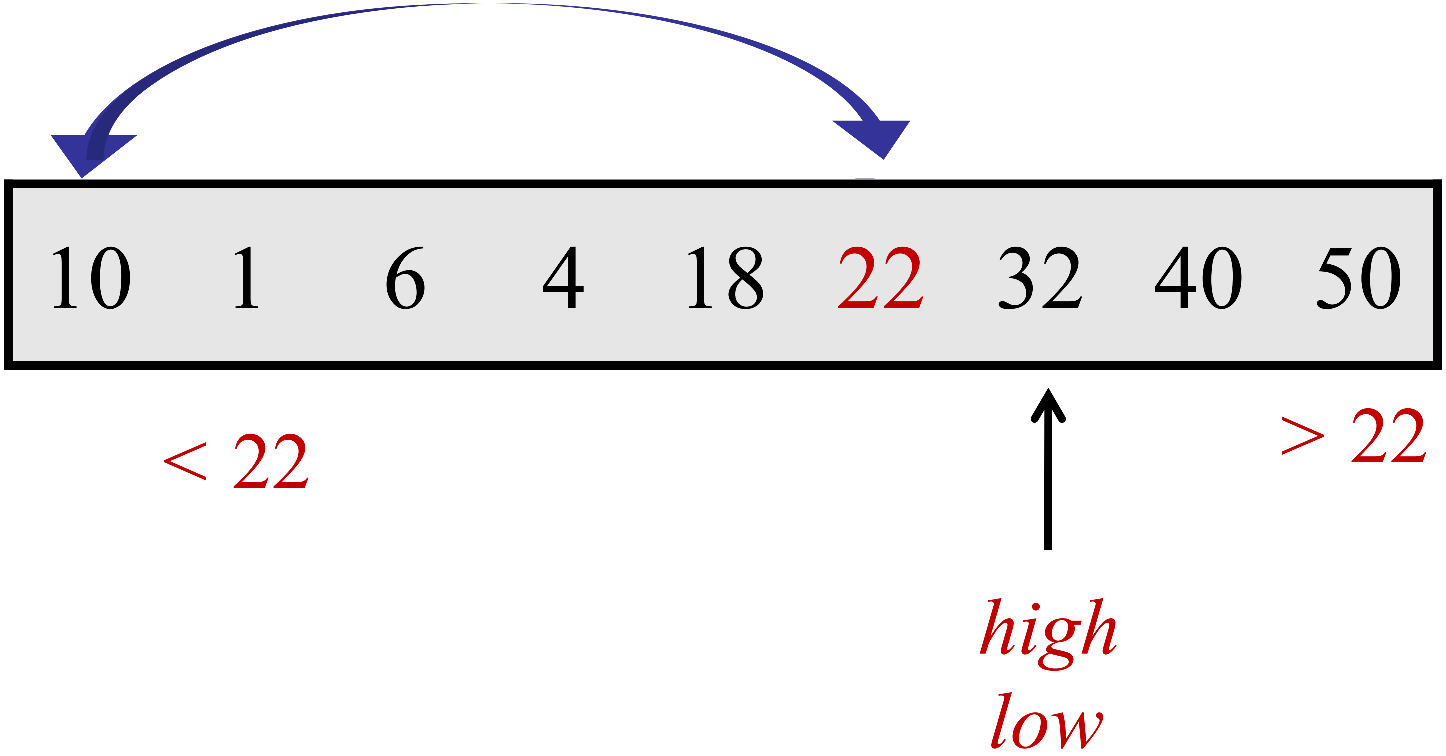
Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22



partition($A[1..n]$, n, $pIndex$)

$pivot = A[pIndex]$;

swap($A[1]$, $A[pIndex]$);

$low = 2$;

$high = n+1$;

while (low < high)

while ($A[low] < pivot$) **and** ($low < high$) **do** $low++$;

while ($A[high] > pivot$) **and** ($low < high$) **do** $high--$;

if ($low < high$) **then** swap($A[low]$, $A[high]$);

swap($A[1]$, $A[low-1]$);

return $low-1$;

Running time:

O(n)

QuickSort

What happens if there are duplicates?

Quicksort

Example:

Running time:

$O(n^2)$

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
6	6	6	6	6	6

Duplicates

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

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

else

Choose pivot index *pIndex*.

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

x = QuickSort(A[1..p–1], p–1)

y = QuickSort(A[p+1..n], n–p)

Duplicates

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

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

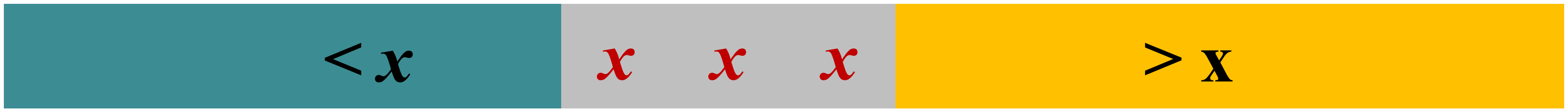
else

Choose pivot index *pIndex*.

p = 3wayPartition(A[1..n], n, *pIndex*)

x = QuickSort(A[1..p–1], p–1)

y = 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:

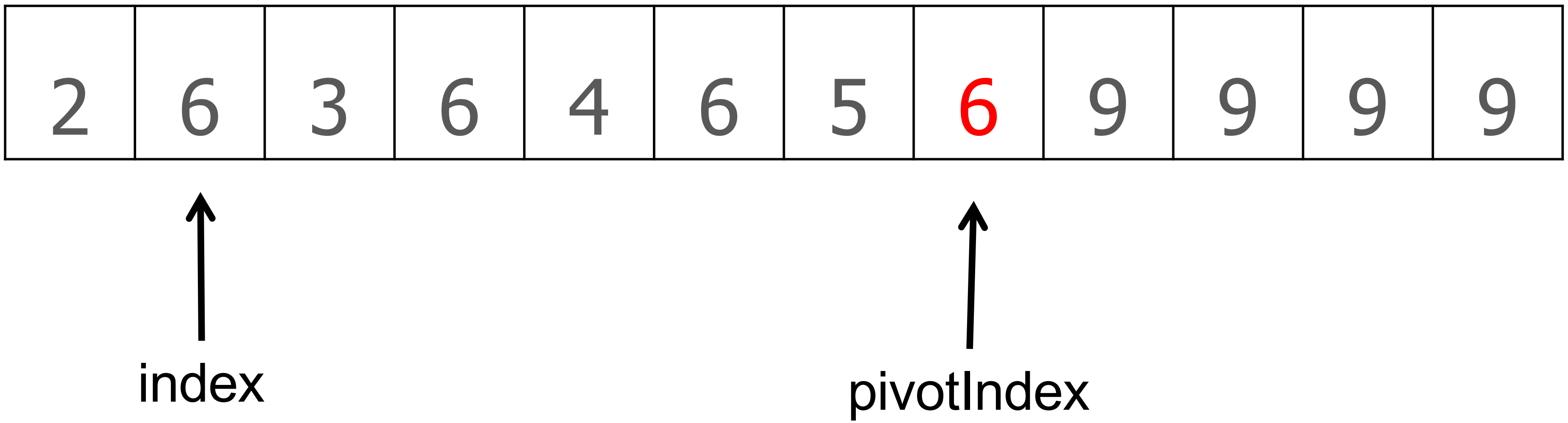
2	6	3	6	4	6	5	6	9	9	9	9

index

pivotIndex

Pack Duplicates

Example:



Pack Duplicates

Example:

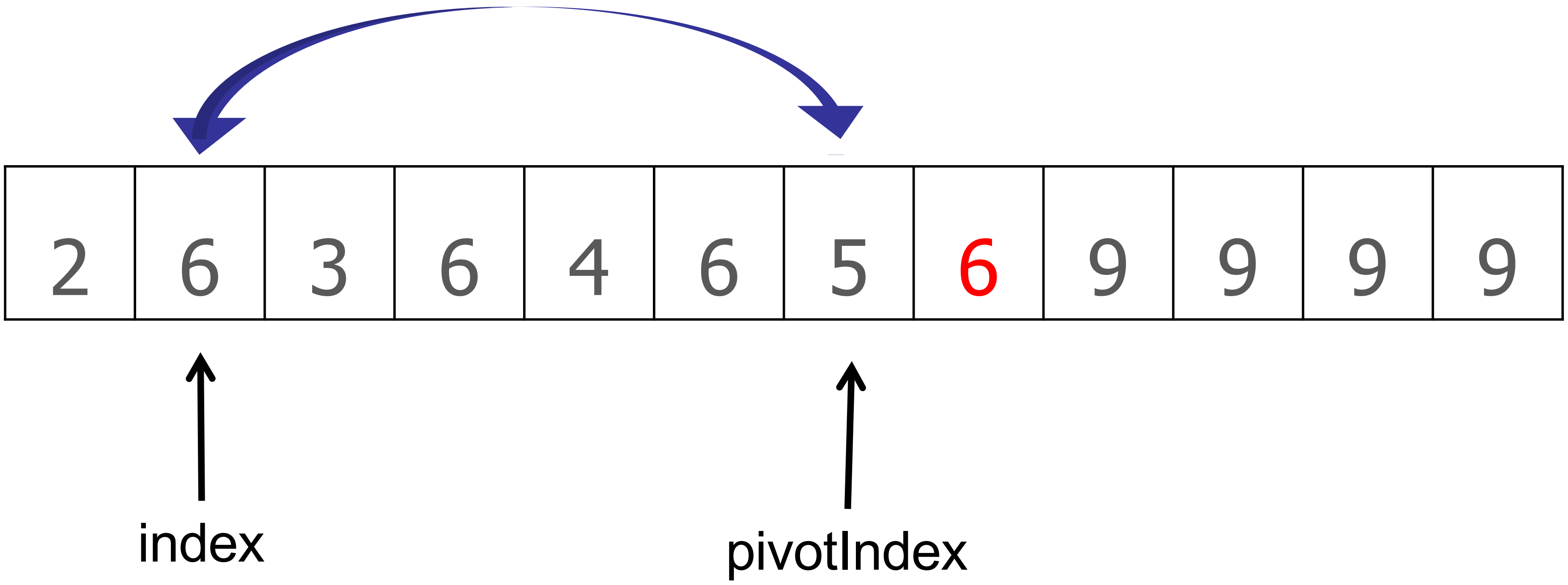
2	6	3	6	4	6	5	6	9	9	9	9

index

pivotIndex

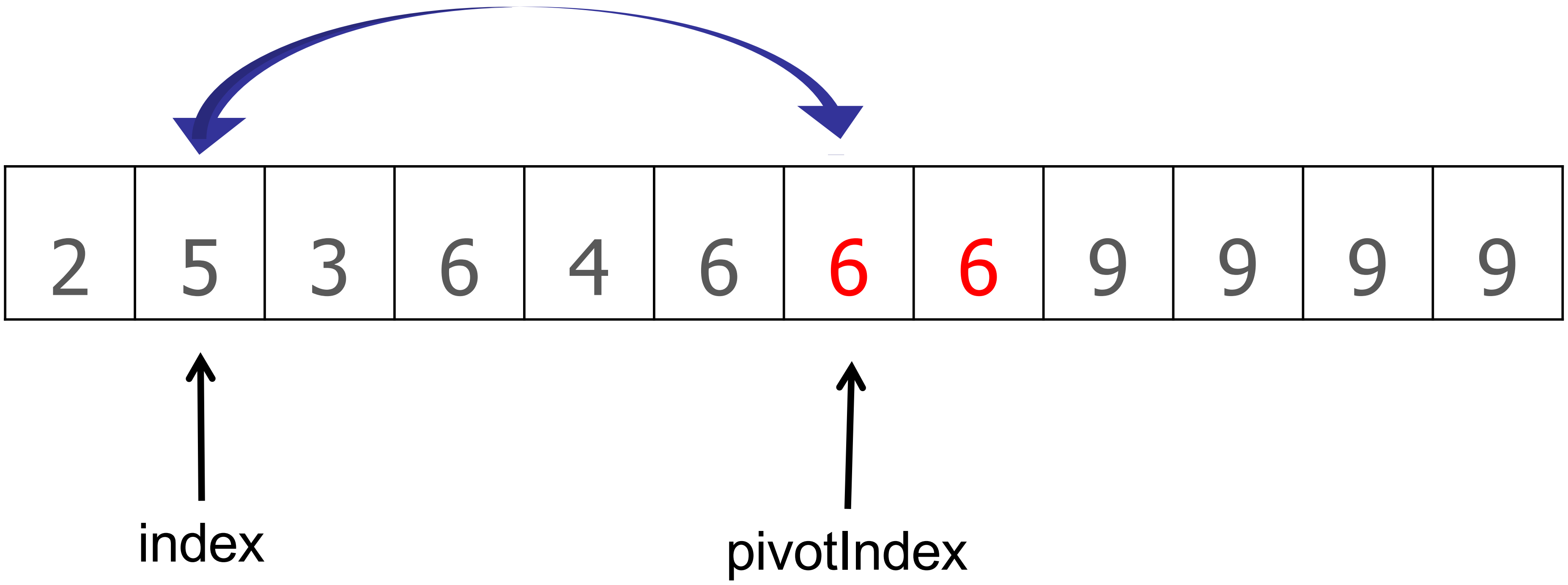
Pack Duplicates

Example:



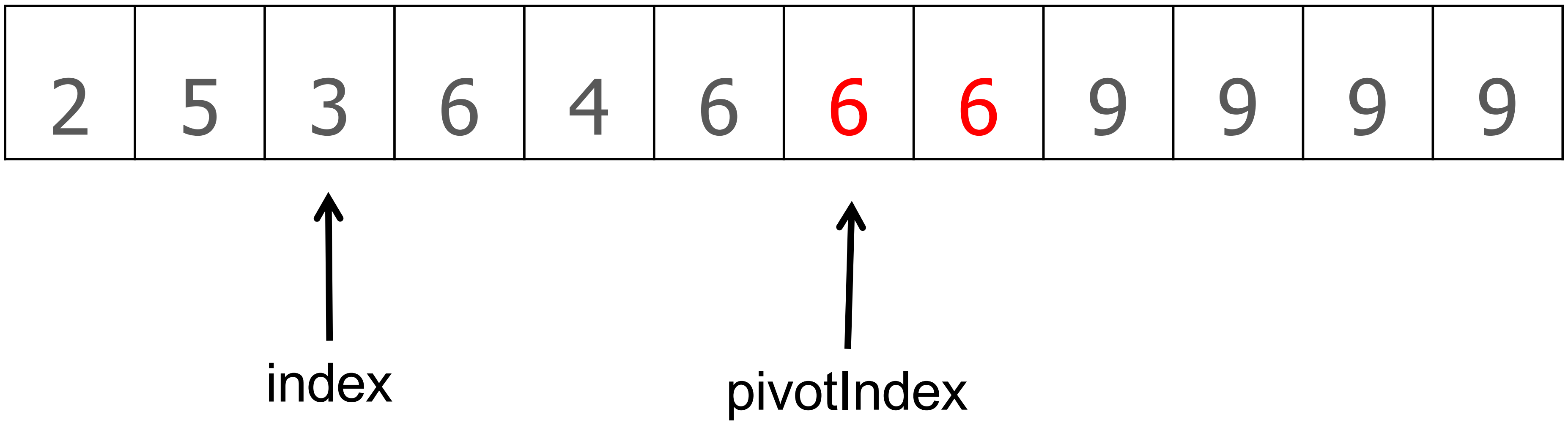
Pack Duplicates

Example:



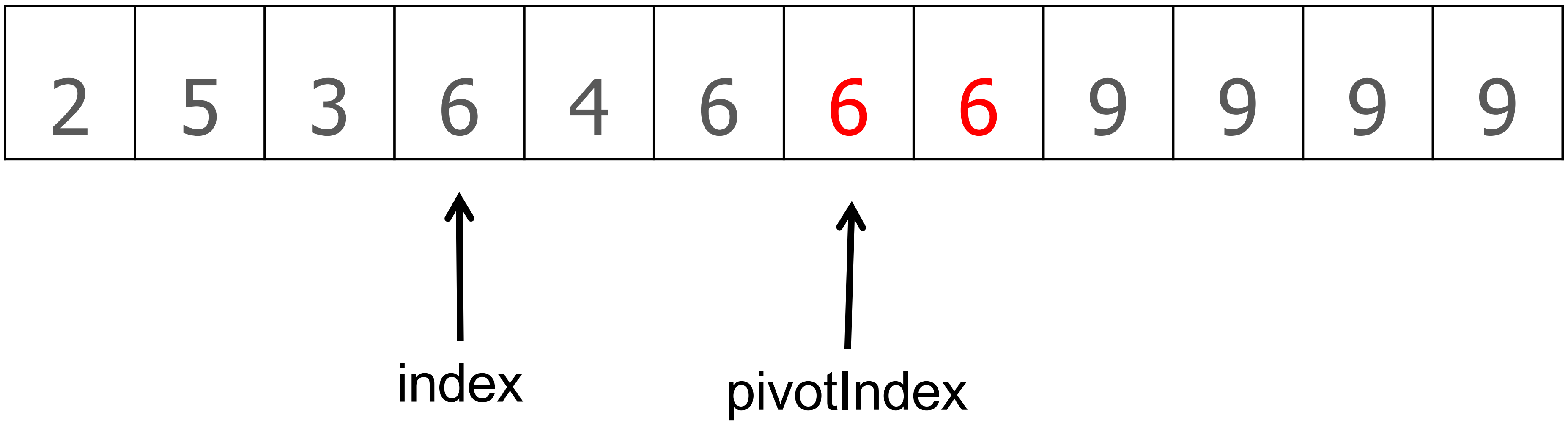
Pack Duplicates

Example:



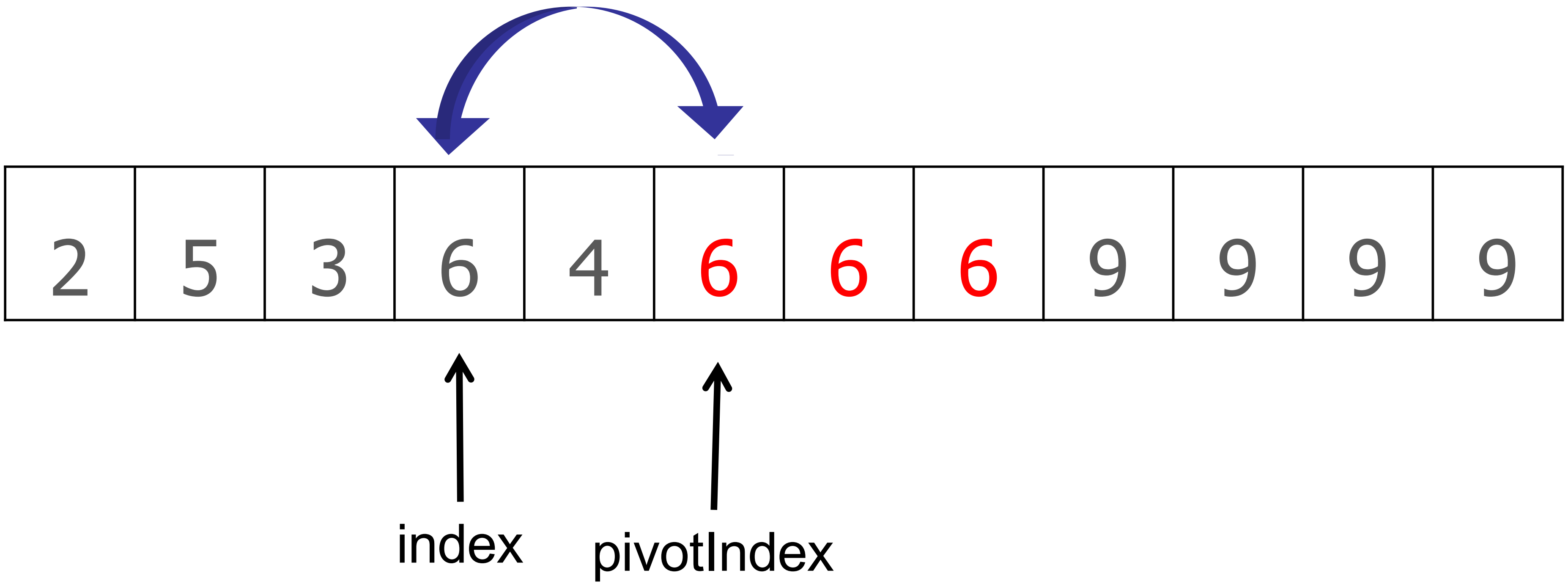
Pack Duplicates

Example:



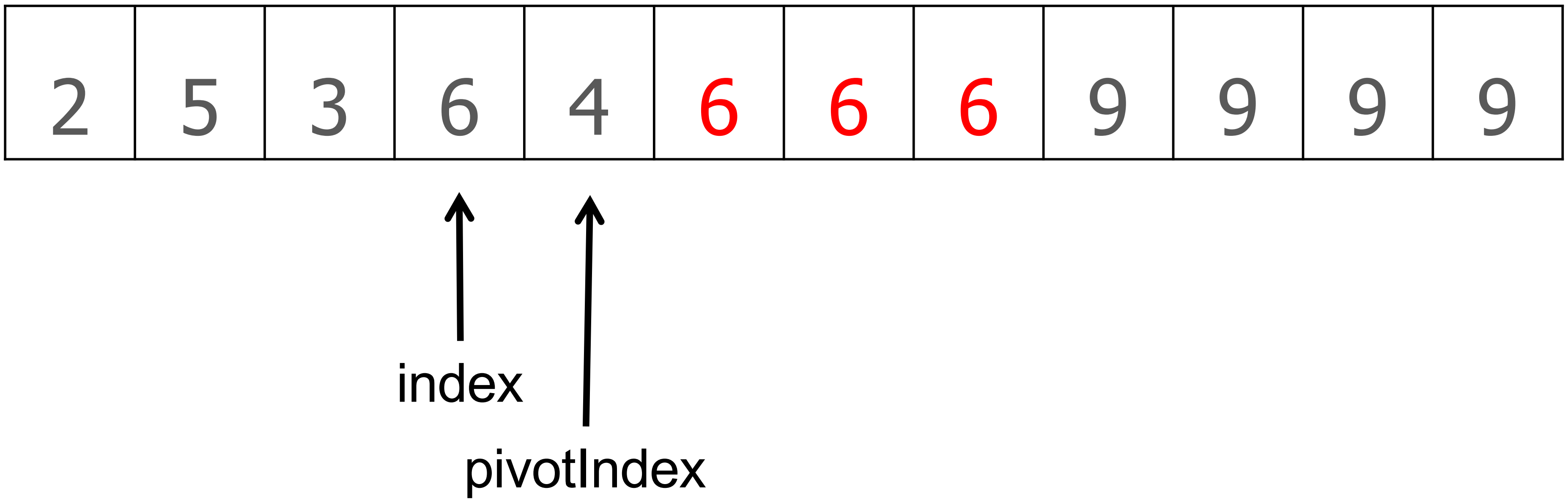
Pack Duplicates

Example:



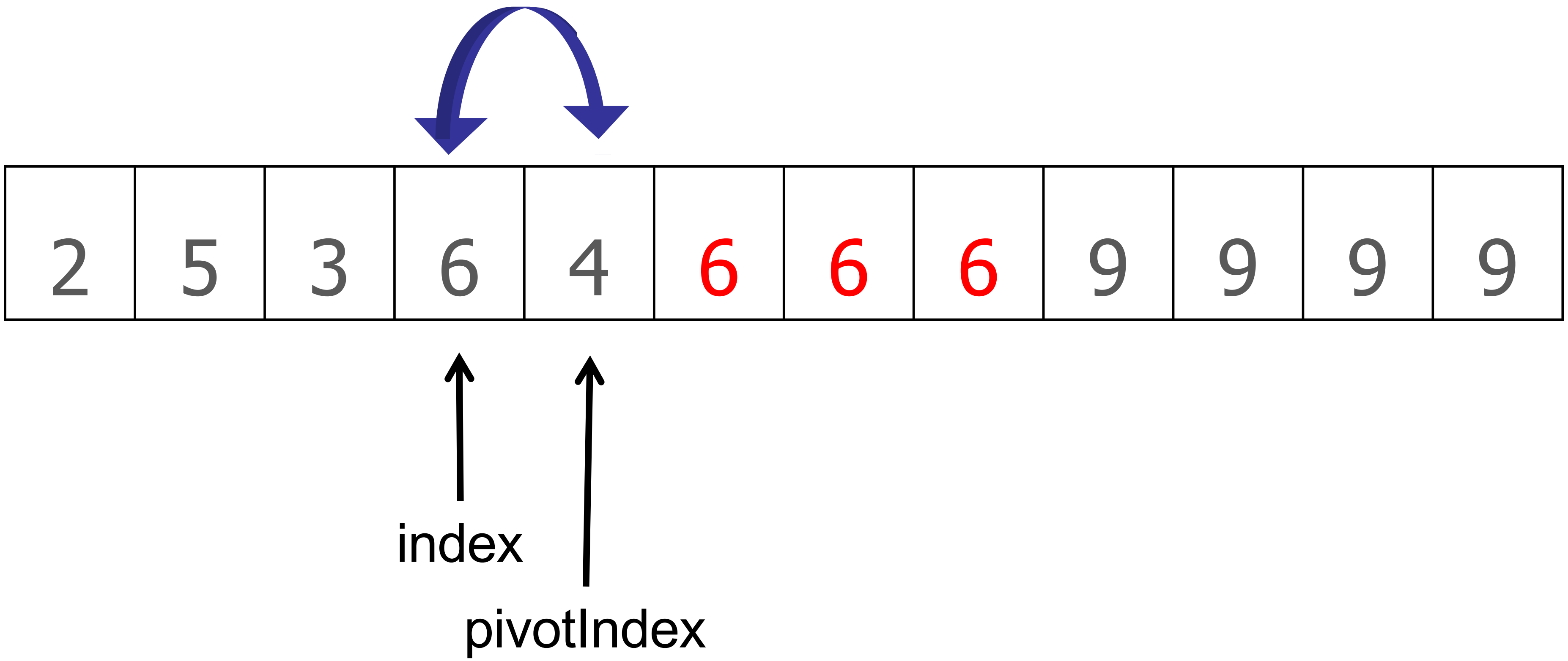
Pack Duplicates

Example:



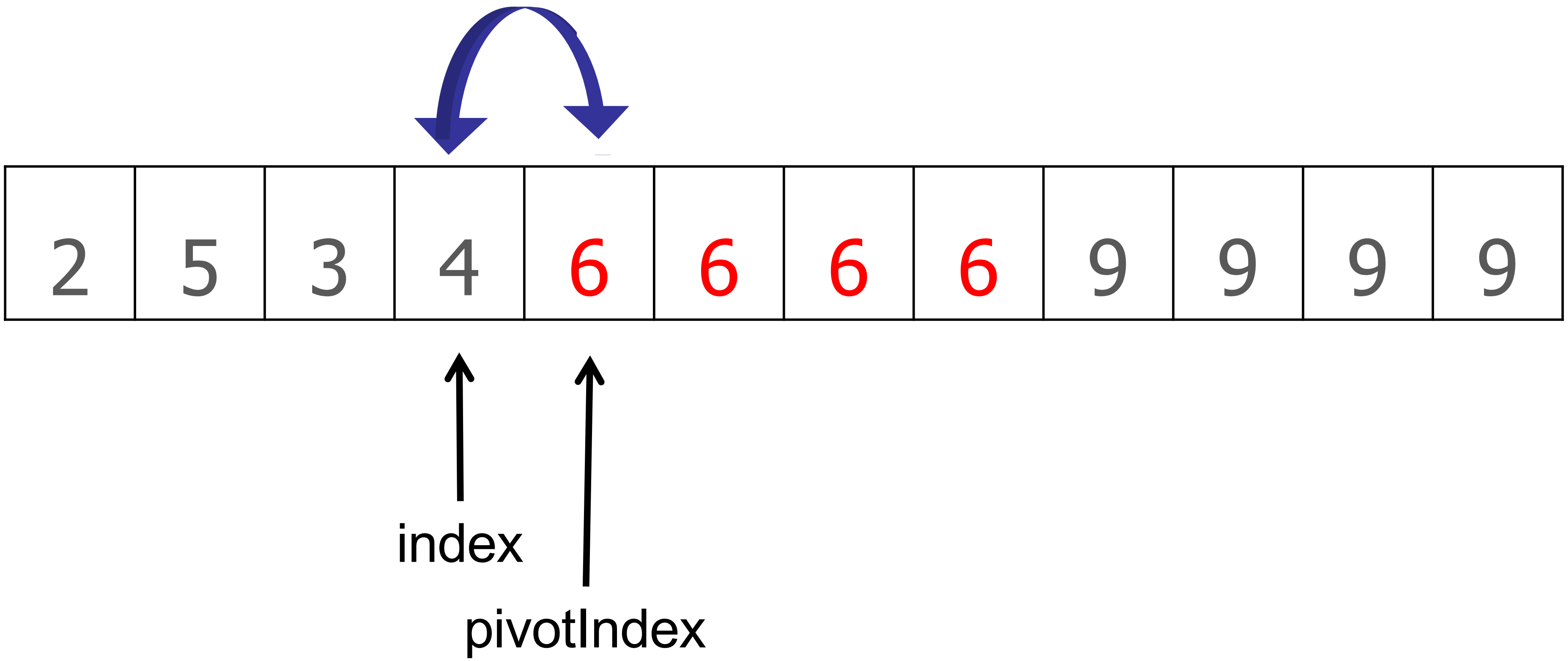
Pack Duplicates

Example:



Pack Duplicates

Example:



Pack Duplicates

Example:

2	5	3	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 = 3wayPartition(A[1..n], n, $pIndex$)

x = QuickSort(A[1..p–1], p–1)

y = QuickSort(A[p+1..n], n–p)

$< x$	x x x	$> \mathbf{x}$

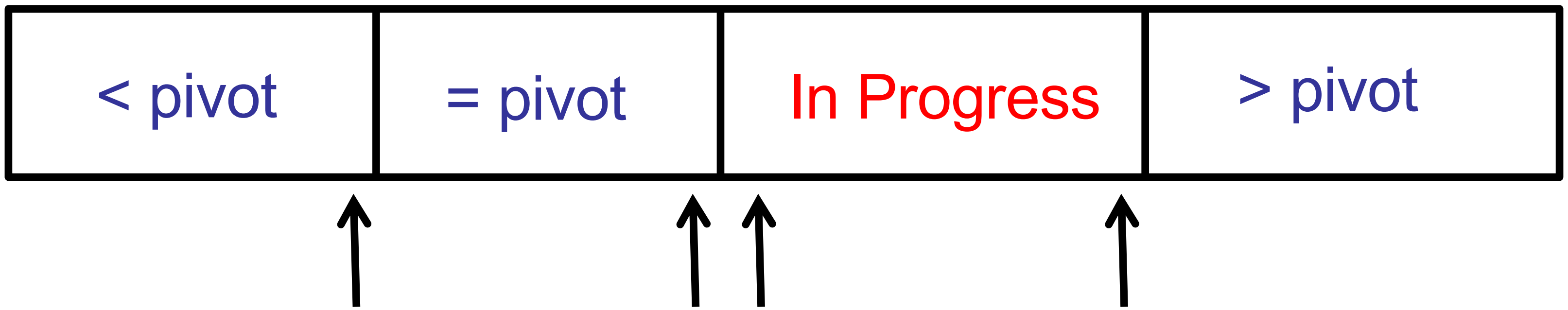
Duplicates

3-Way Partitioning

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

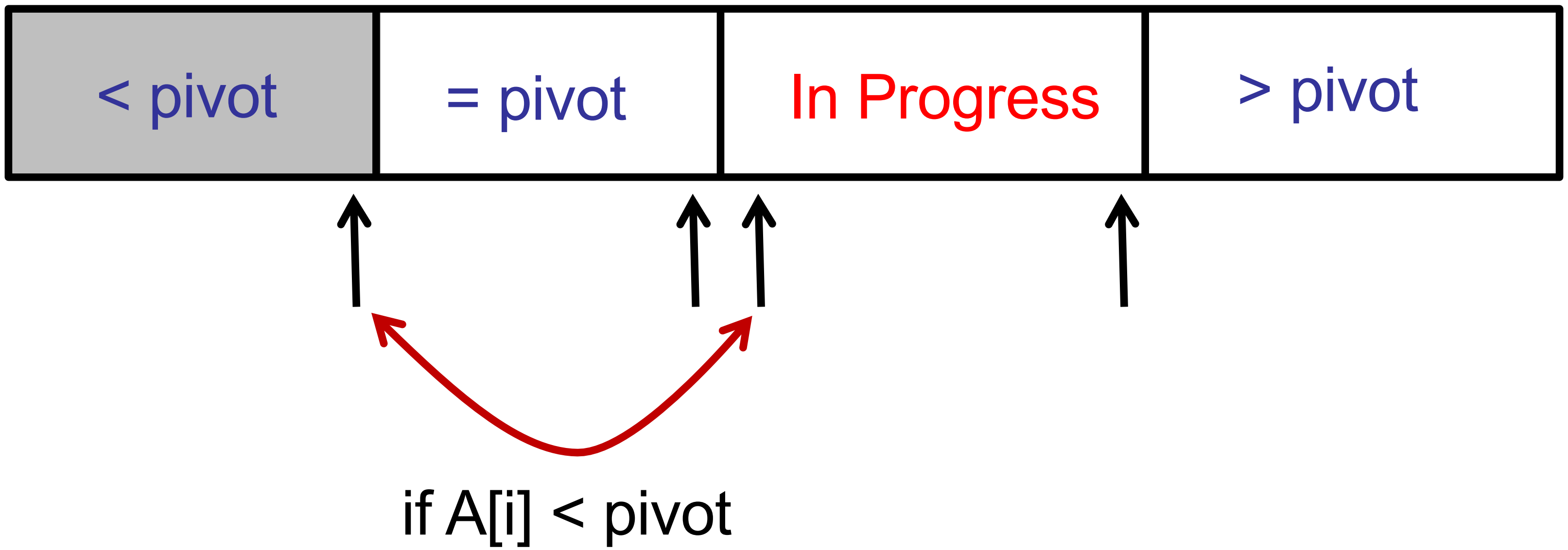
- Option 2: one pass partitioning
  - More complicated.
  - Maintain four regions of the array

3-Way Partitioning

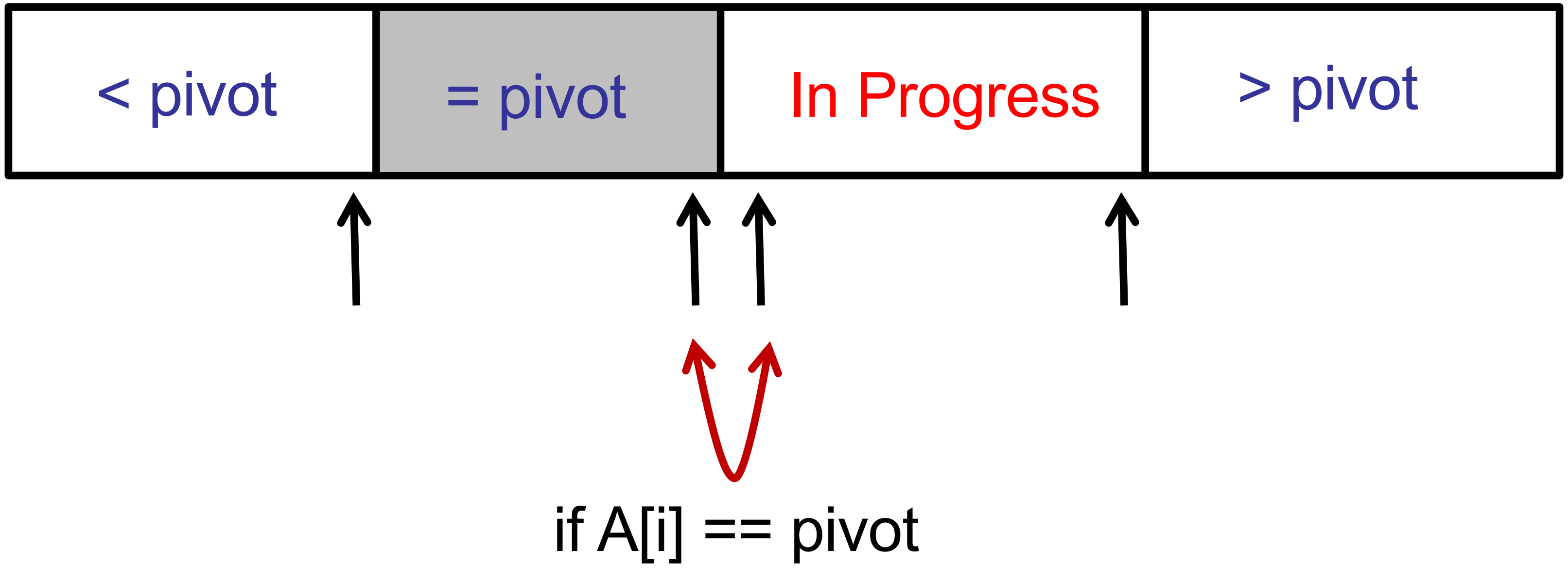


(Note: think about array in terms of invariants!)

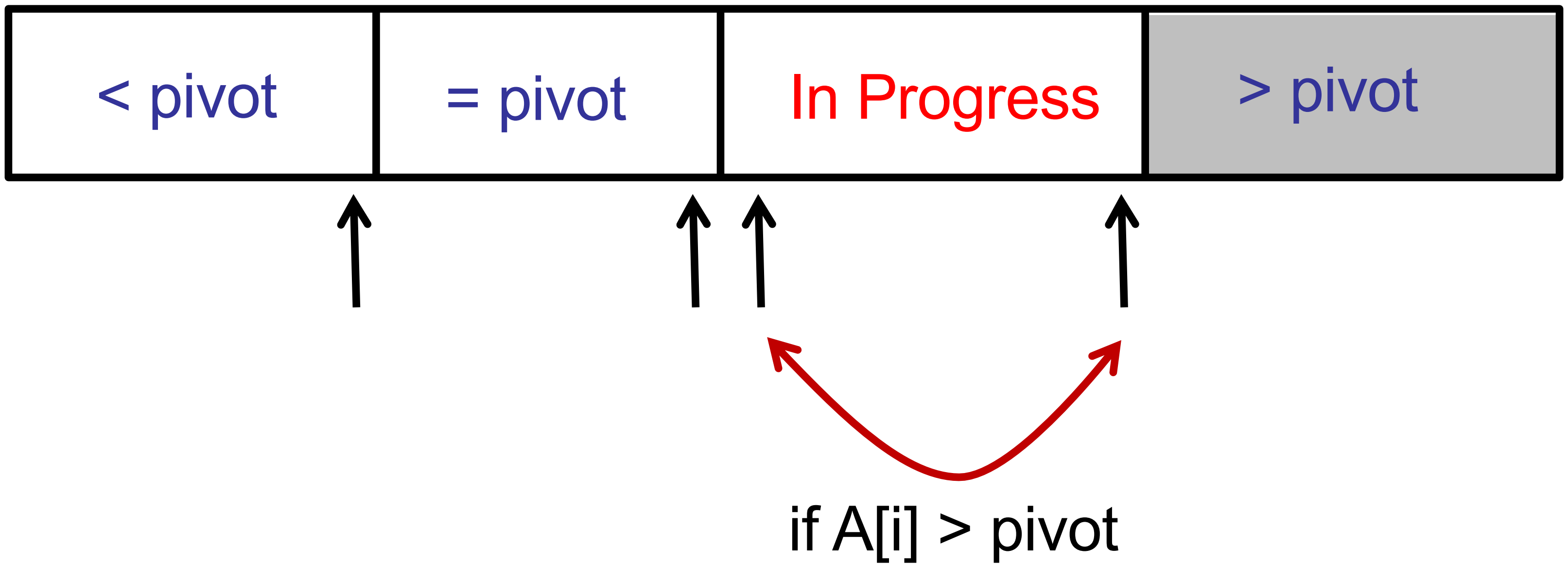
3-Way Partitioning



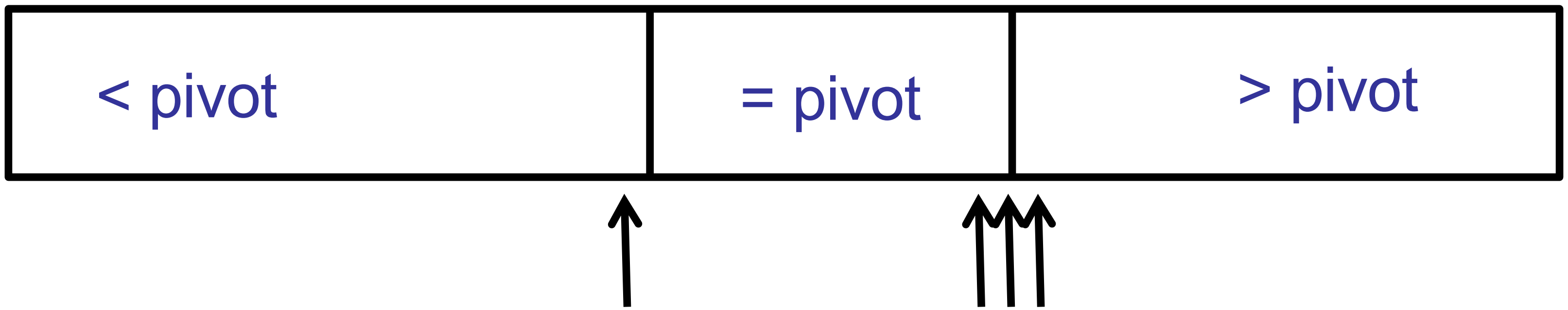
3-Way Partitioning



3-Way Partitioning



3-Way Partitioning



Duplicates

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

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

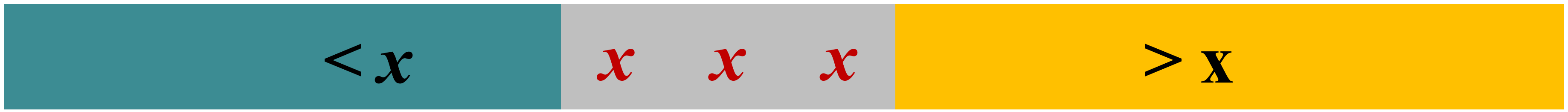
else

Choose pivot index $pIndex$.

p = 3wayPartition(A[1..n], n, $pIndex$)

x = QuickSort(A[1..p–1], p–1)

y = QuickSort(A[p+1..n], n–p)

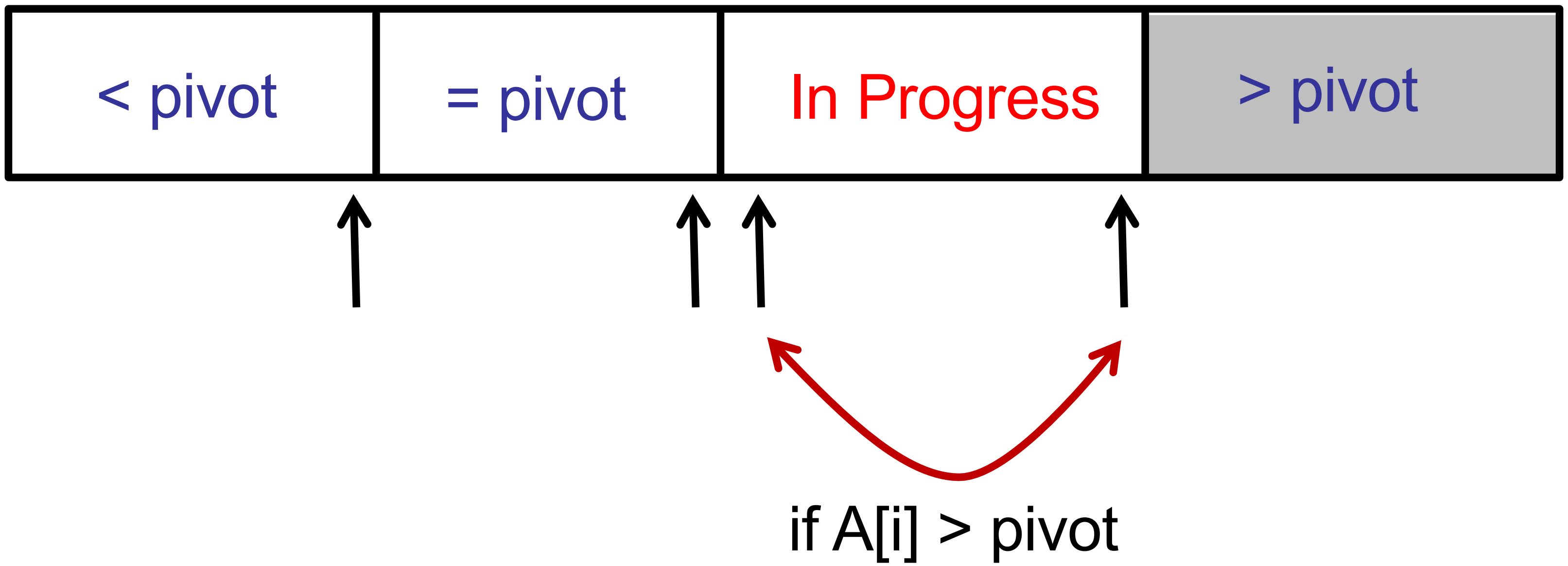


Is QuickSort stable?

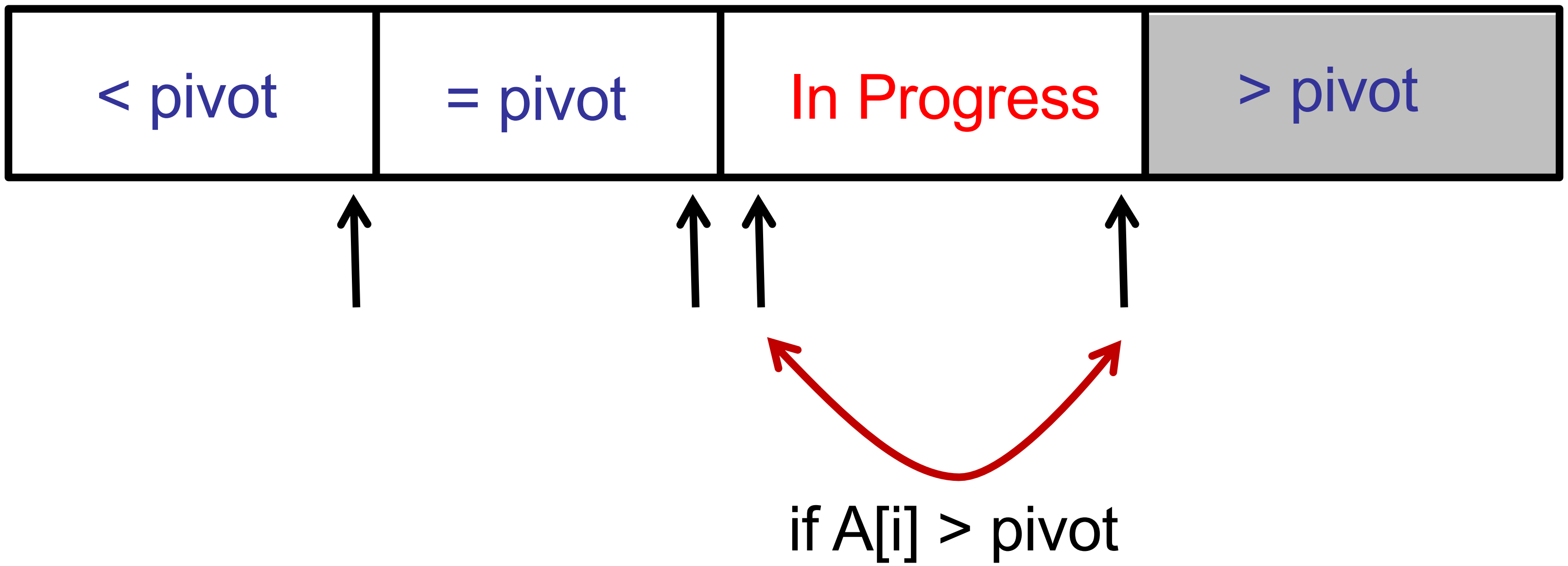
ARCHIPELAGO

is open

QuickSort is not stable



QuickSort is not stable



Very tricky to achieve all desirable properties at once:

- Stable
- In-place
- Efficient (time)

Sorting, continued

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])

Which option is best?

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

$$\Downarrow$$

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

$\Downarrow$

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

98	97	96	95	94	93	92	99	100

$\Downarrow$

97	96	95	94	93	92	98	99	100

Deterministic QuickSort

QuickSort Recurrence:

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

Cost of partition on n elements

Cost of QuickSort on 1 element

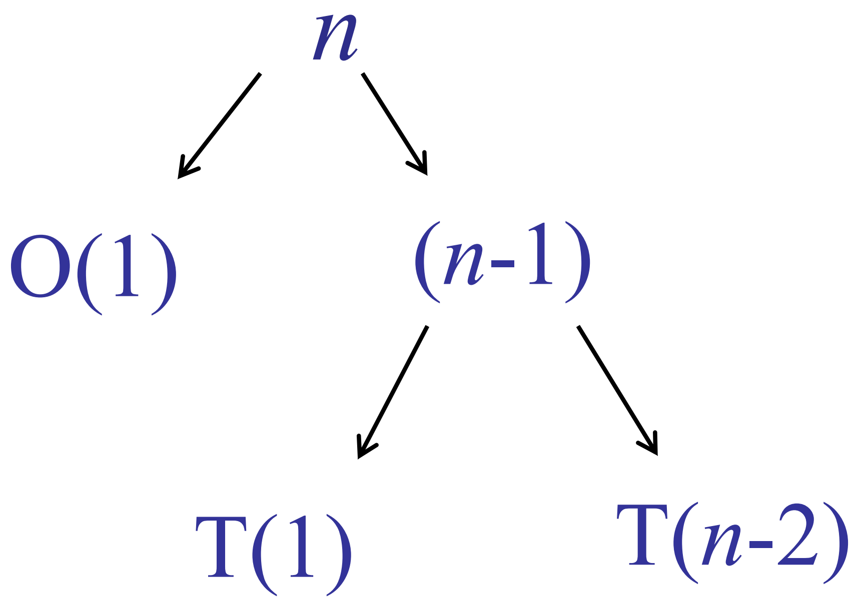
Cost of QuickSort on n – 1 elements

Cost of QuickSort on n elements

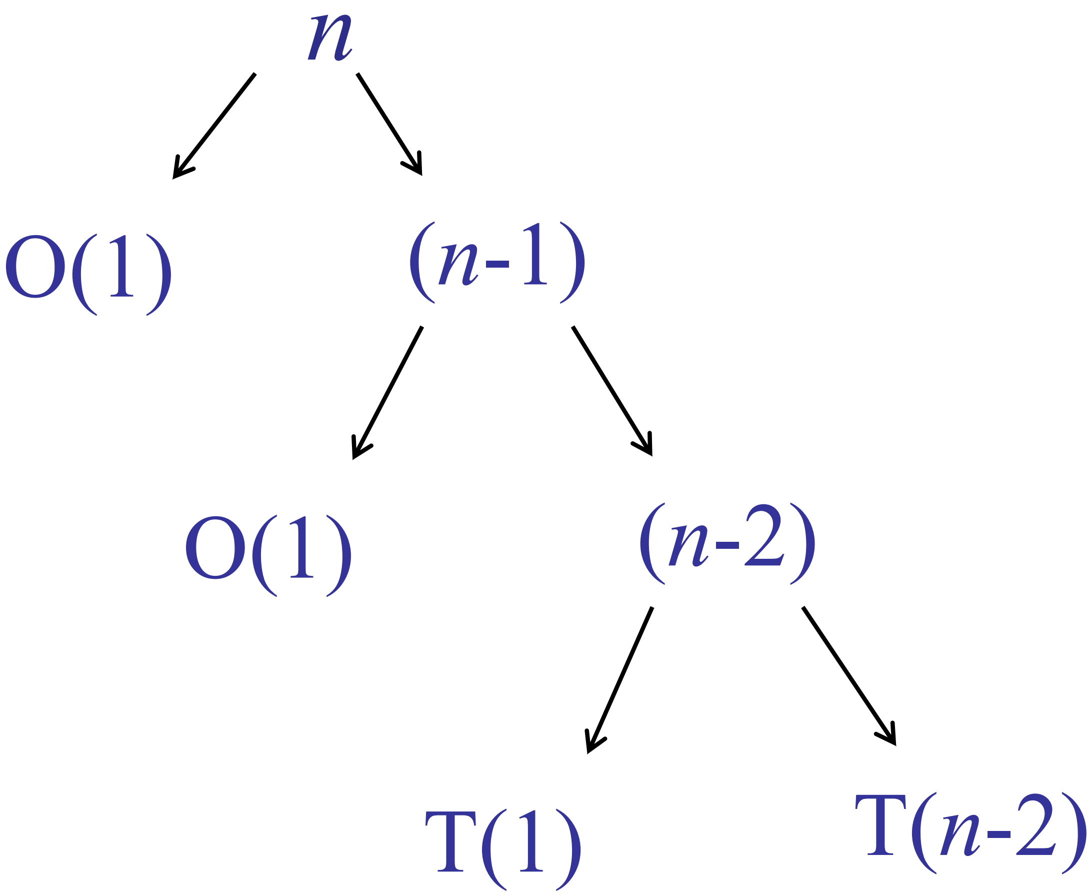
Deterministic QuickSort

$$\begin{tikzcd} & n \arrow[dl] \arrow[dr] & \\ \mathrm{T}(1) & & \mathrm{T}(n\text{-}1) \end{tikzcd}$$

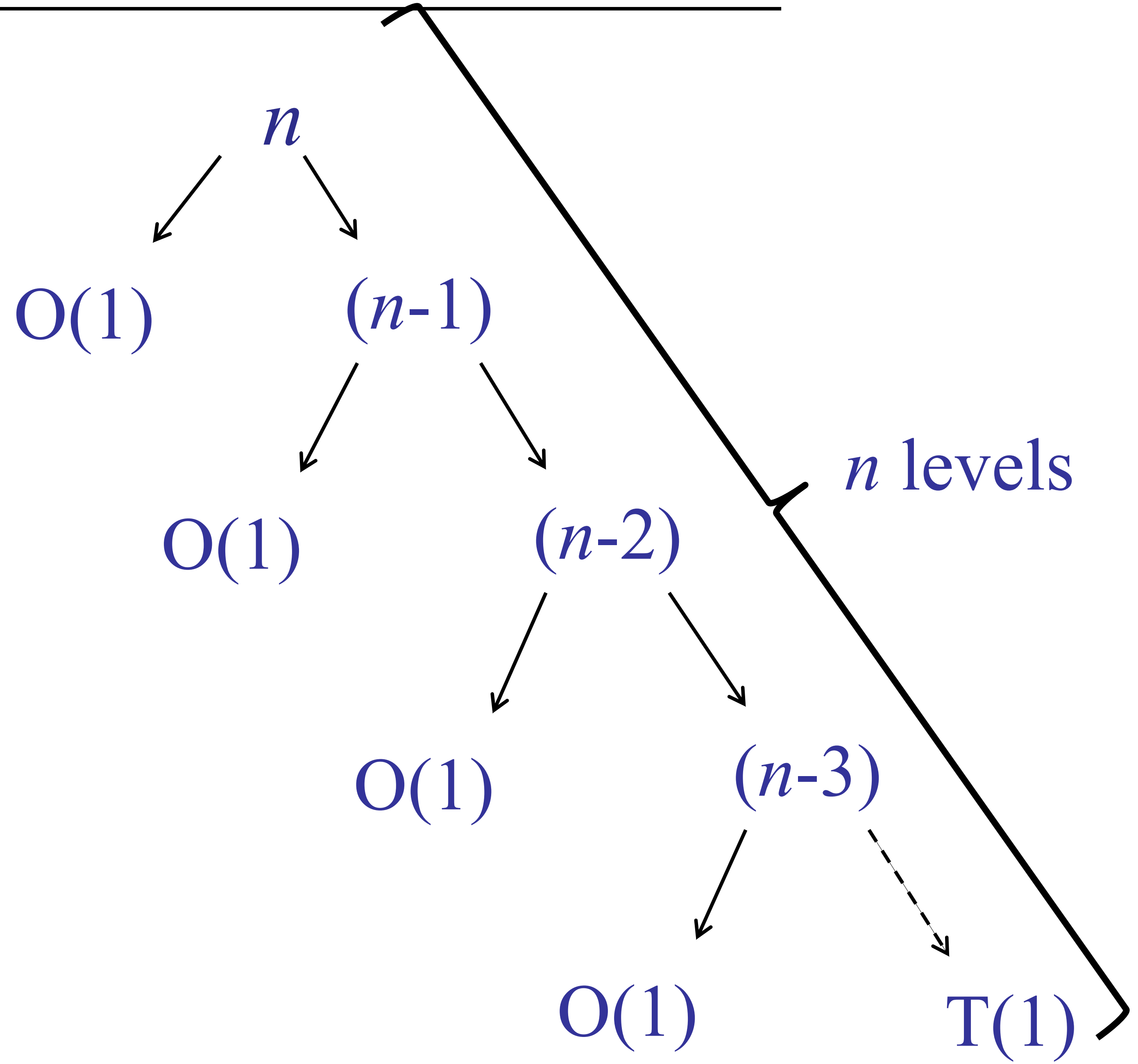
Deterministic QuickSort



Deterministic QuickSort

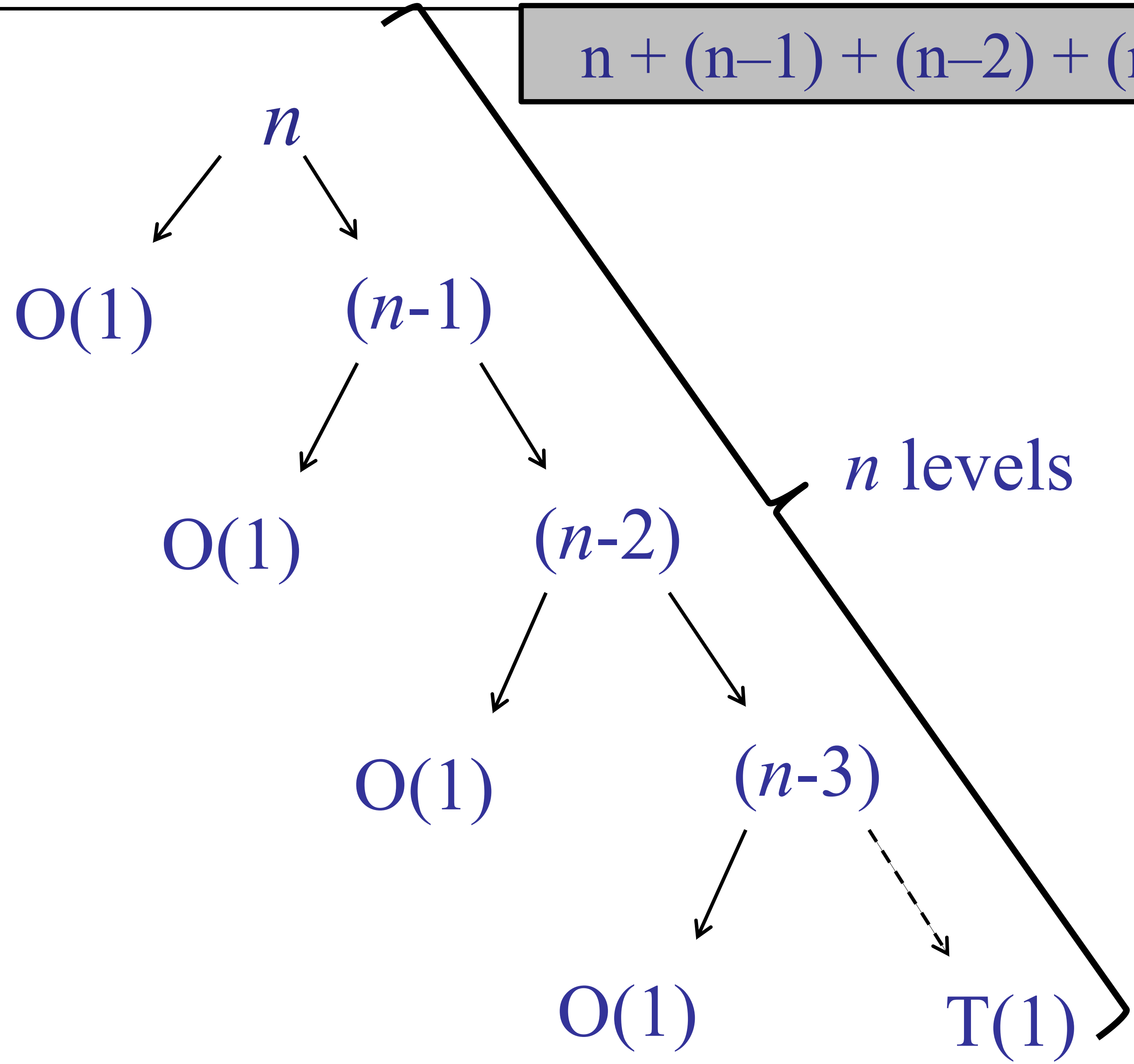


Deterministic QuickSort



Deterministic QuickSort

$$n + (n-1) + (n-2) + (n-3) + \ldots = \mathbf{O(n^2)}$$



QuickSort

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

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

else

Choose pivot index *pIndex*.

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

x = QuickSort(A[1..p–1], p–1)

y = QuickSort(A[p+1..n], n–p)

Ideally: partition should split the array evenly.

n/2 elements	*x*	n/2 elements

Better QuickSort

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

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

Cost of partition on n elements

Cost of QuickSort on *high* elements

Cost of QuickSort on *low* elements

Cost of QuickSort on n elements

Better QuickSort

If we split the array evenly:

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

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

$$= O(n \log n)$$

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)
  - ??

Note: 10 is an arbitrary integer!

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. $H > n/10$

Note: 10 is an arbitrary integer!

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:

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

$$< T(9n/10) + T(9n/10) + n$$

$$< 2T(9n/10) + n$$

$$< O(n \log n)$$

What is wrong?

QuickSort

Tempting solution:

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

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

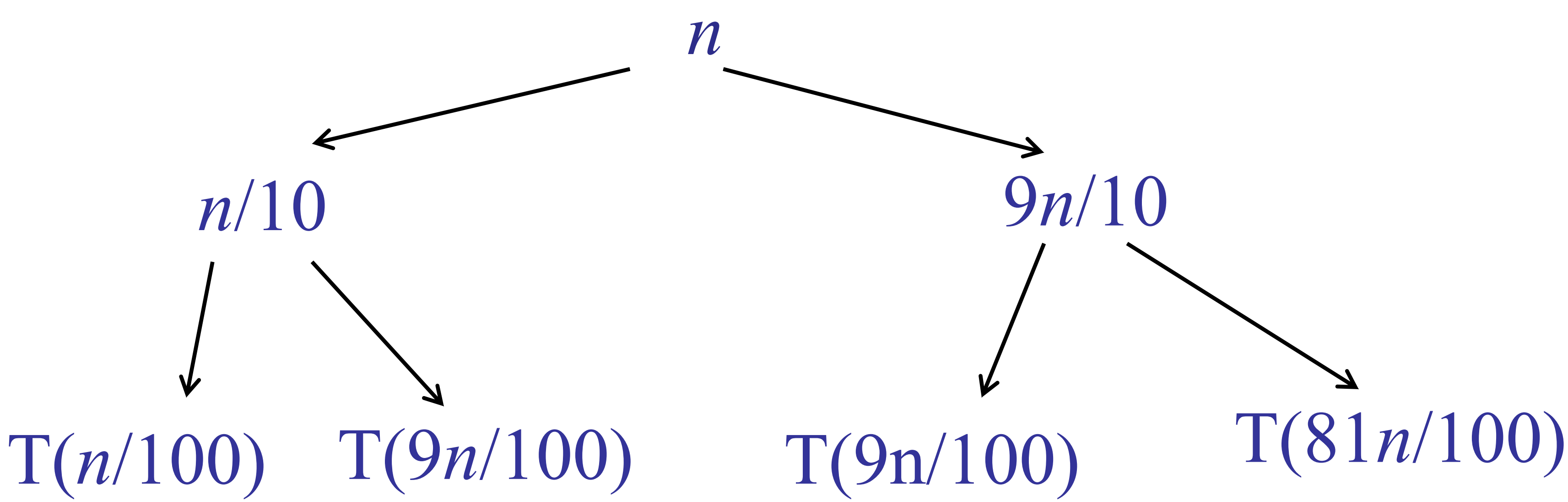
$$k = n/10$$

$$n$$

$$T(n/10) \qquad\qquad T(9n/10)$$

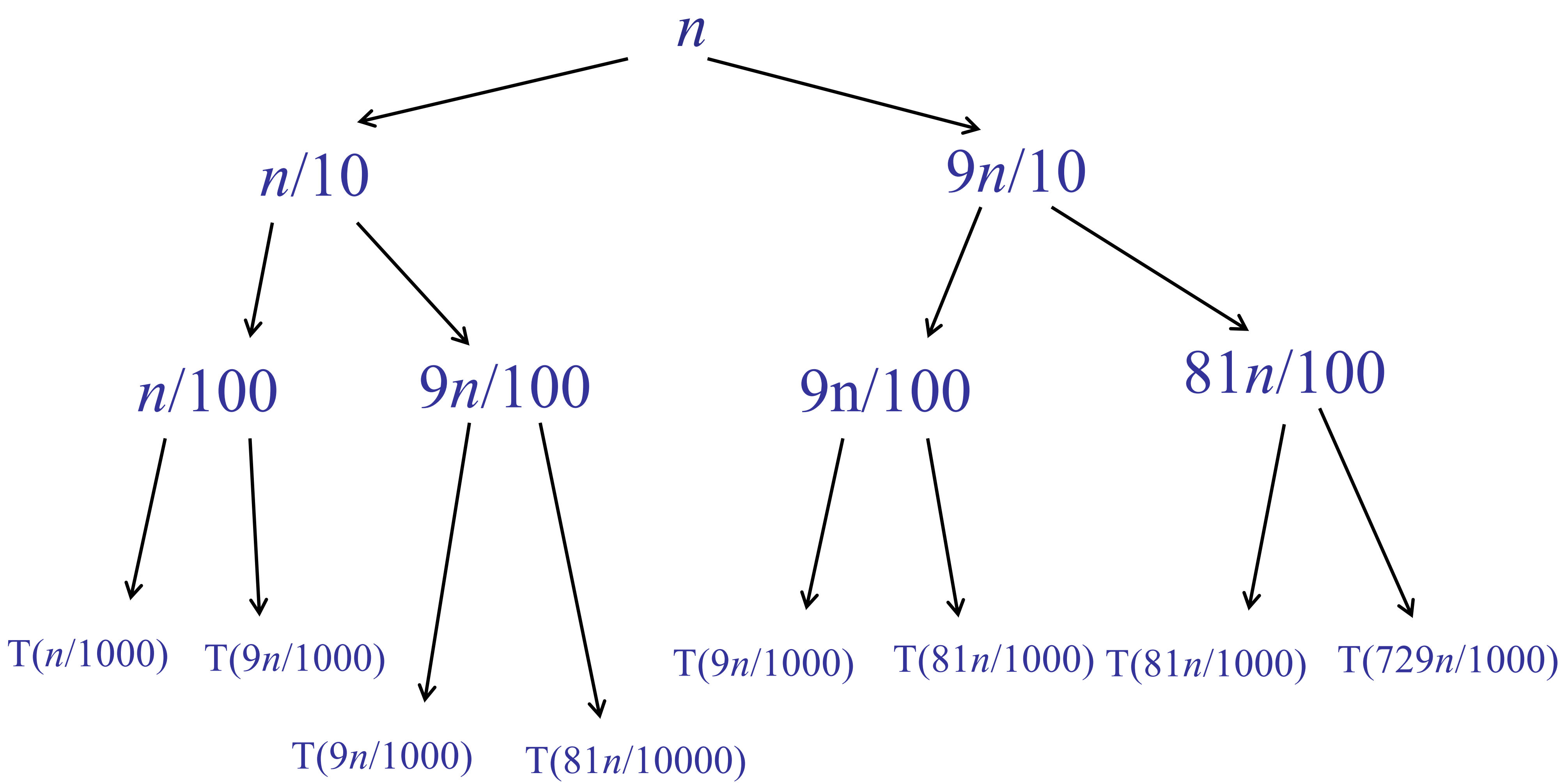
QuickSort Analysis

$k = n/10$



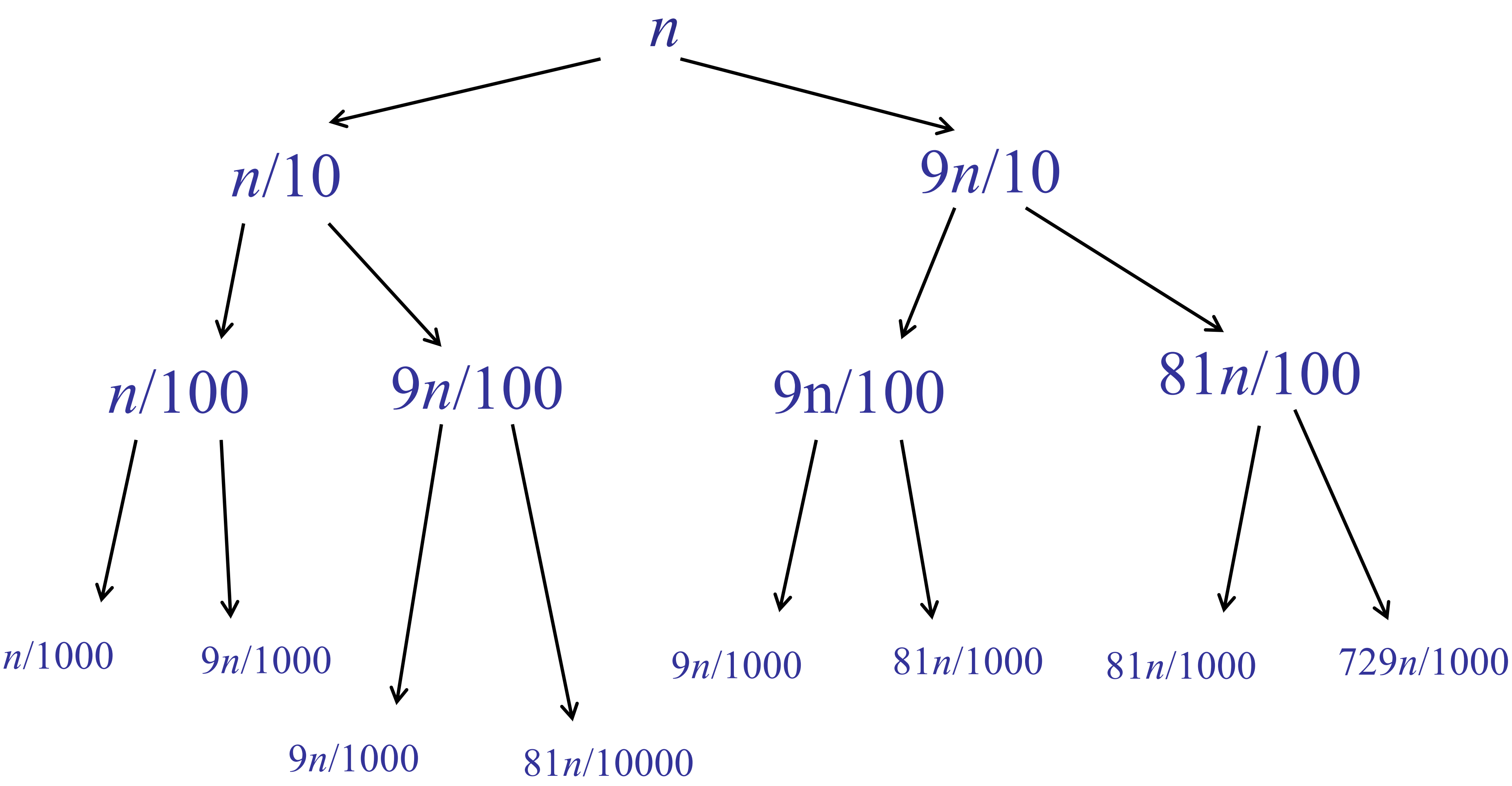
QuickSort Analysis

$k = n/10$

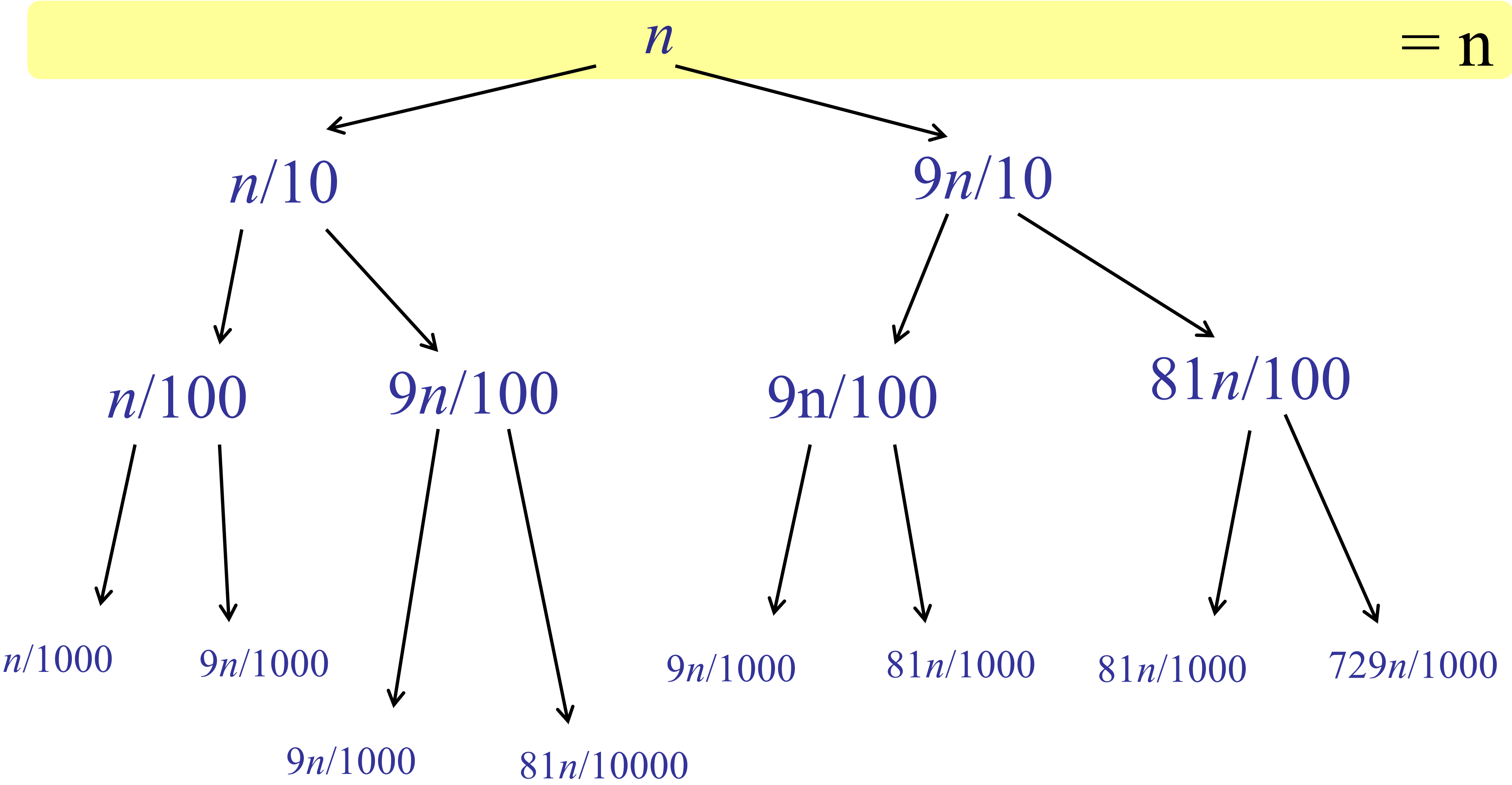


QuickSort Analysis

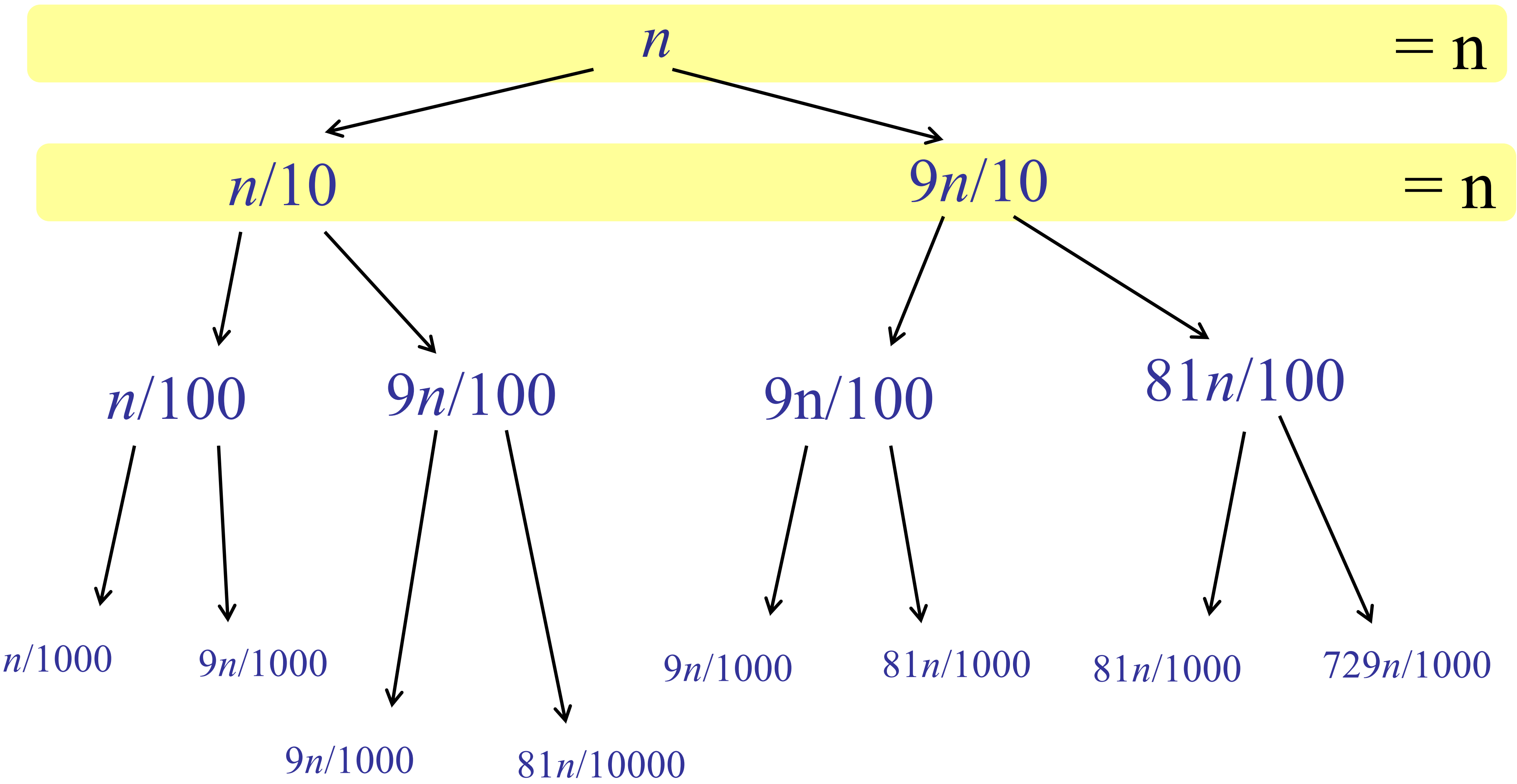
$k = n/10$



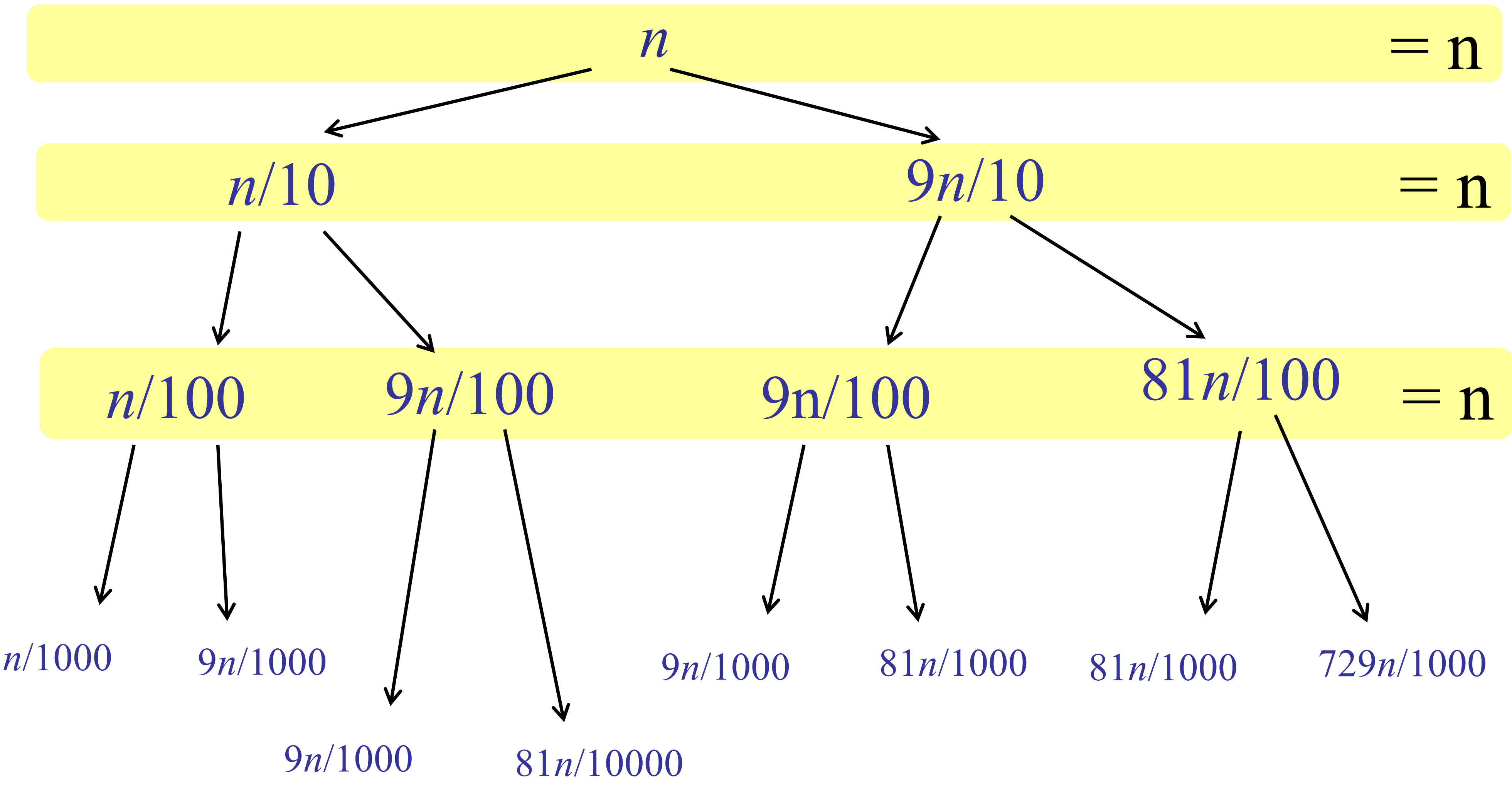
QuickSort Analysis



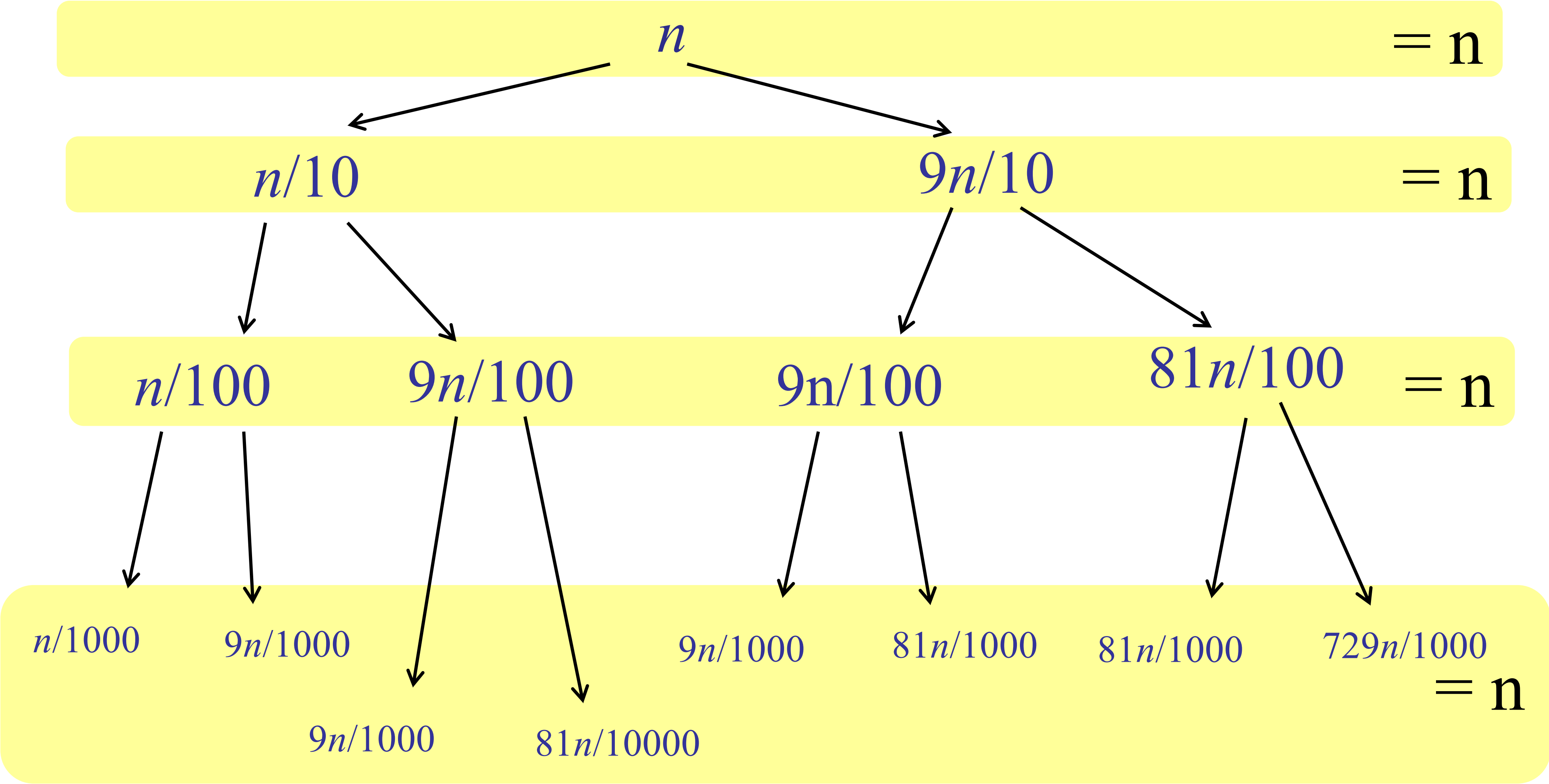
QuickSort Analysis



QuickSort Analysis

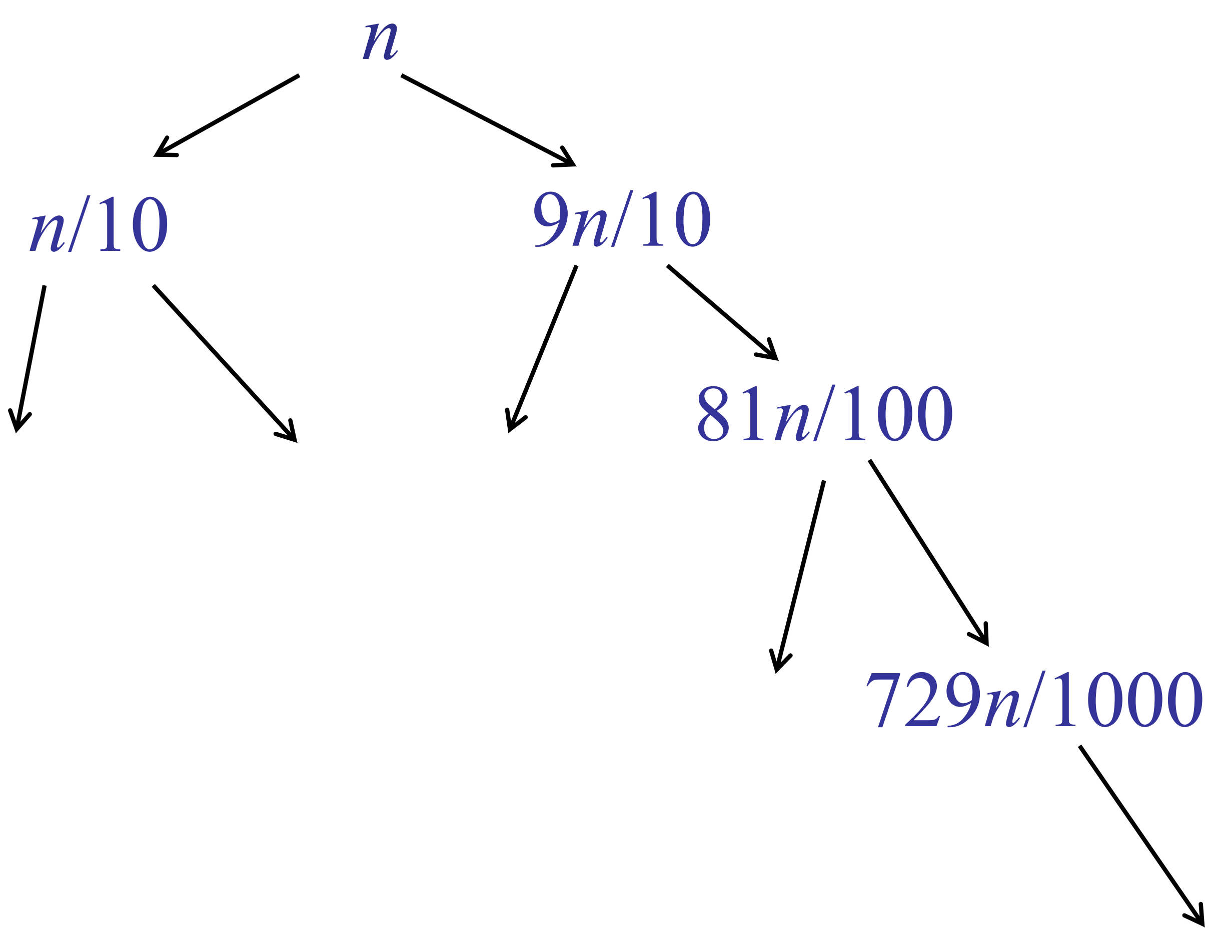


QuickSort Analysis



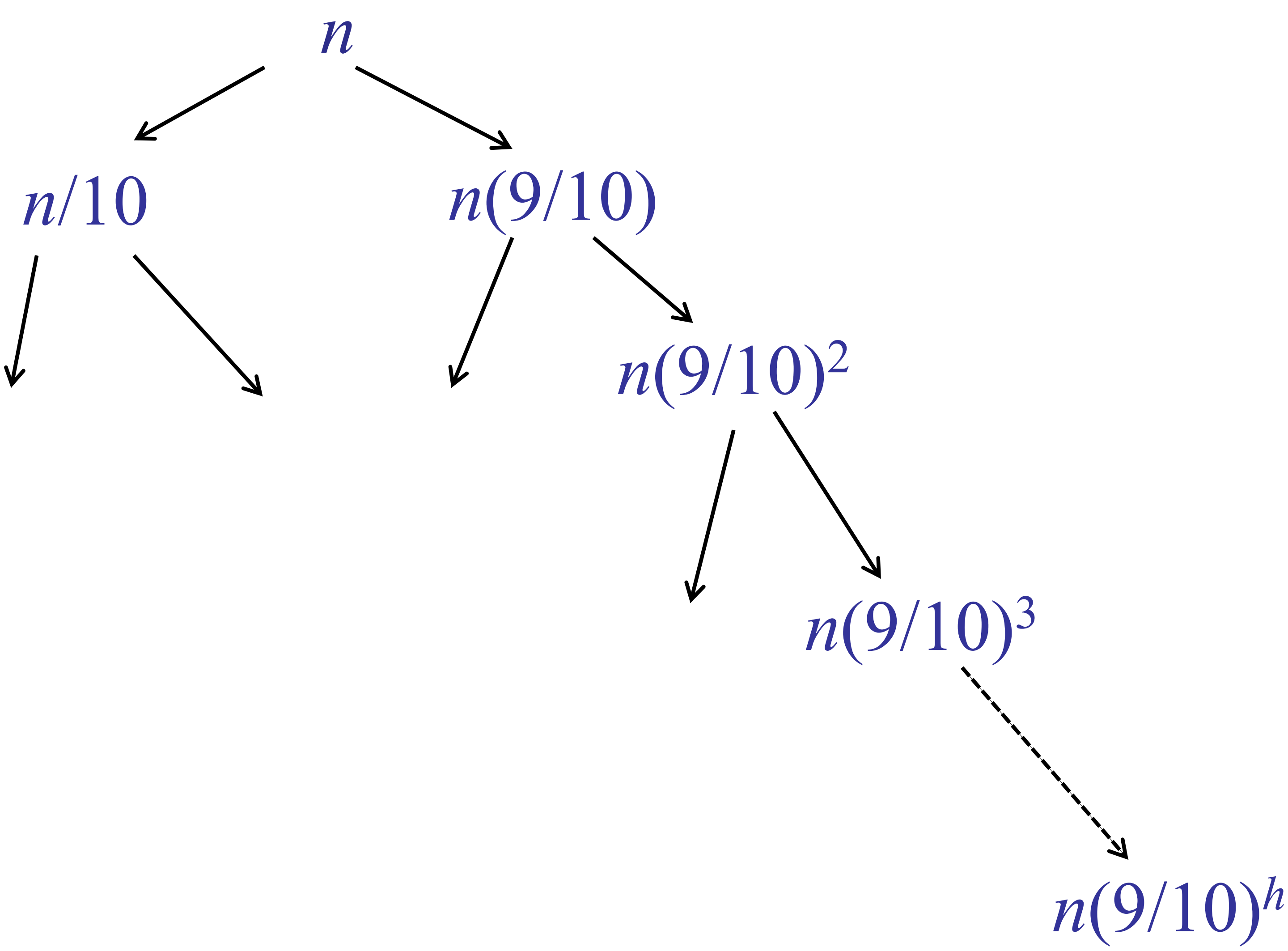
How many levels??

QuickSort Analysis



How many levels??

QuickSort Analysis



How many levels??

QuickSort Analysis

Maximum number of levels:

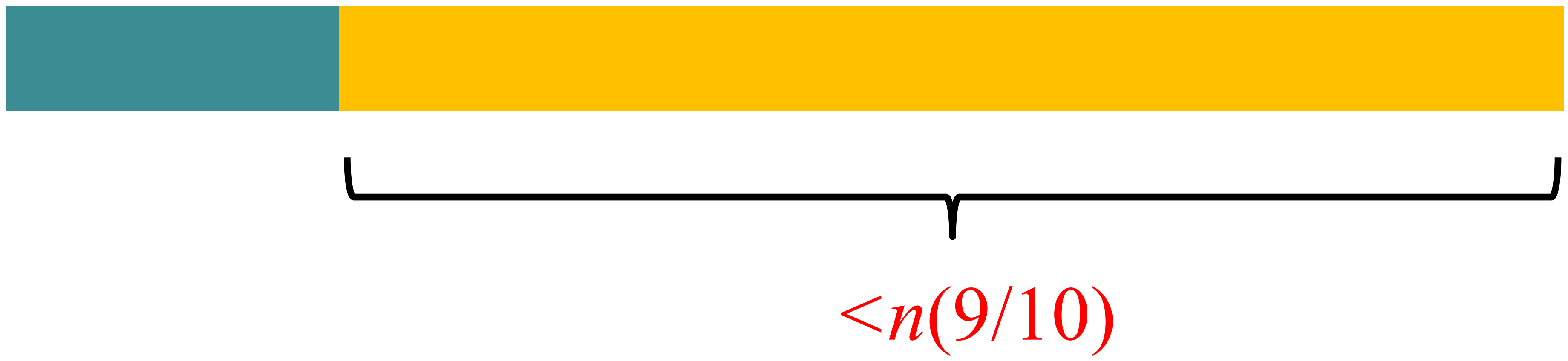
$$1 = n(9/10)^h$$

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

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

QuickSort Analysis

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



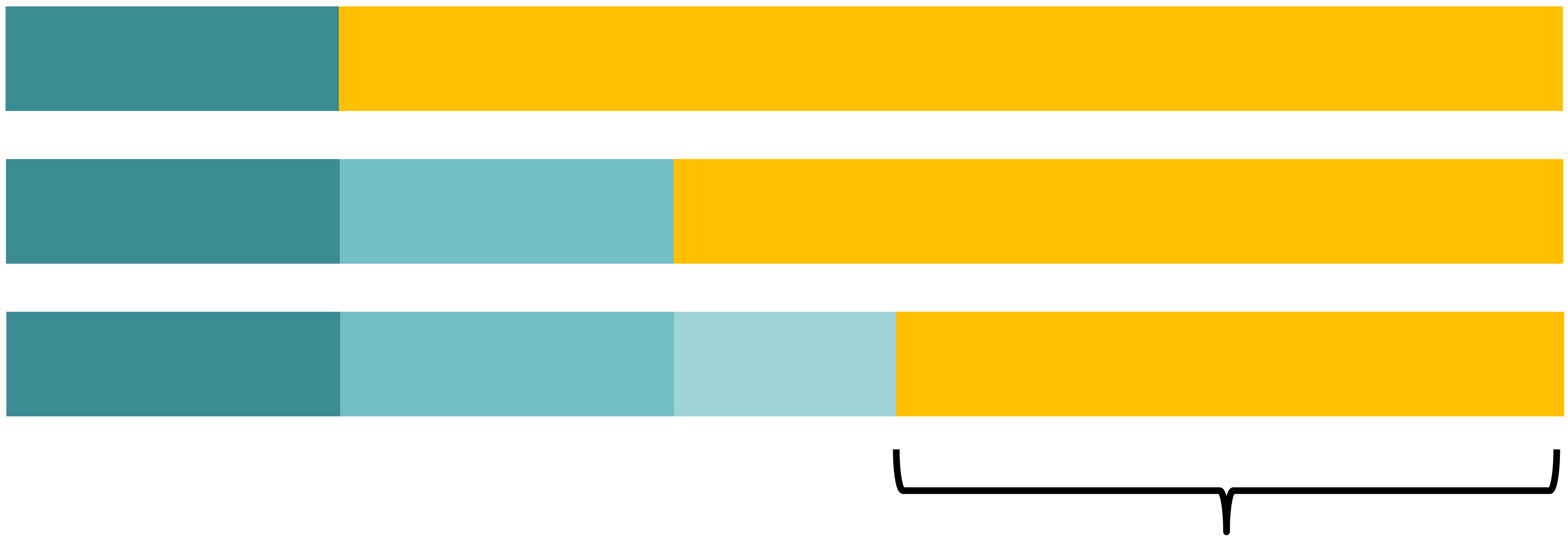
QuickSort Analysis

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

$< n(9/10)^2$

QuickSort Analysis

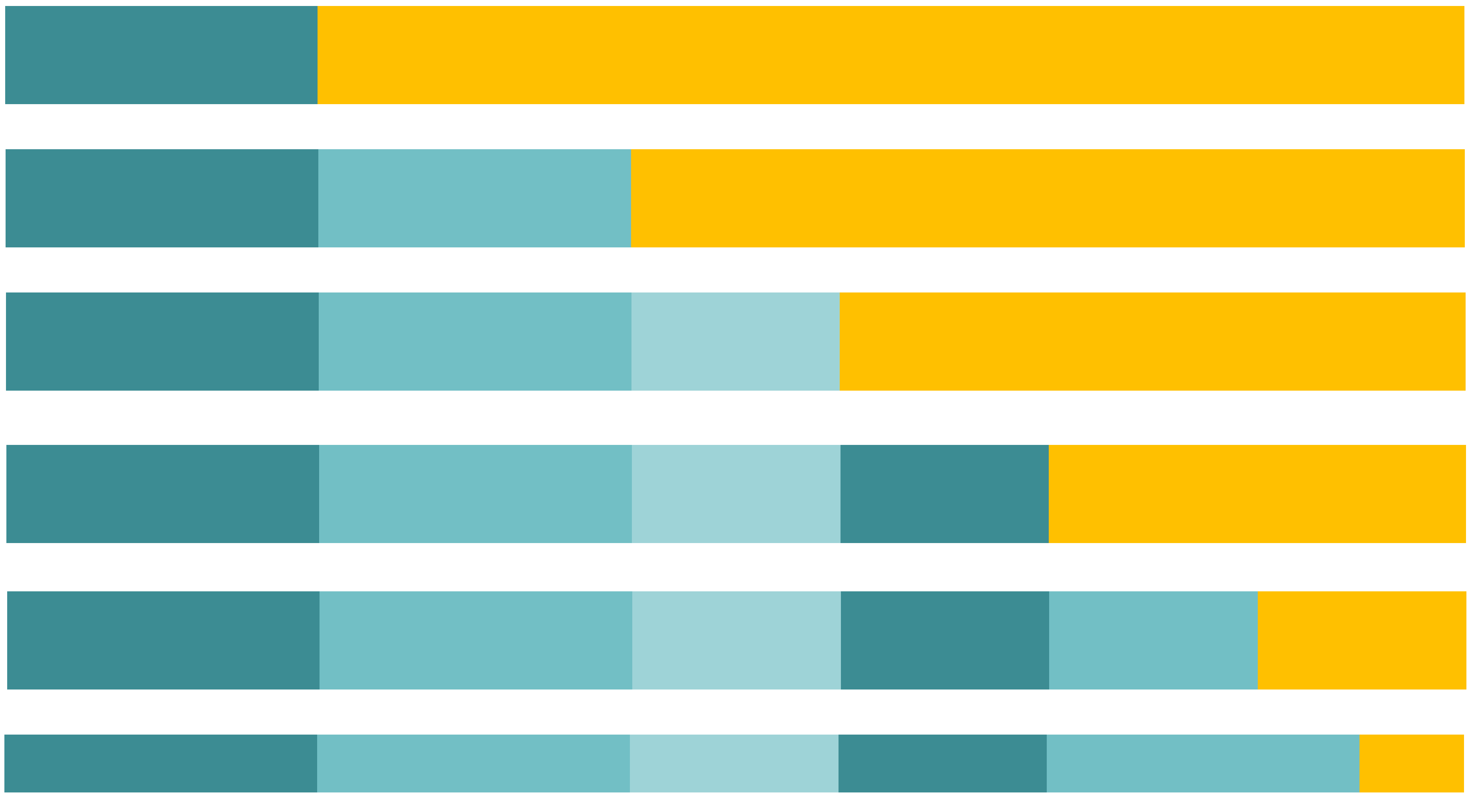
Assume larger part shrinks by at least 9/10 every iteration:



$< n(9/10)^3$

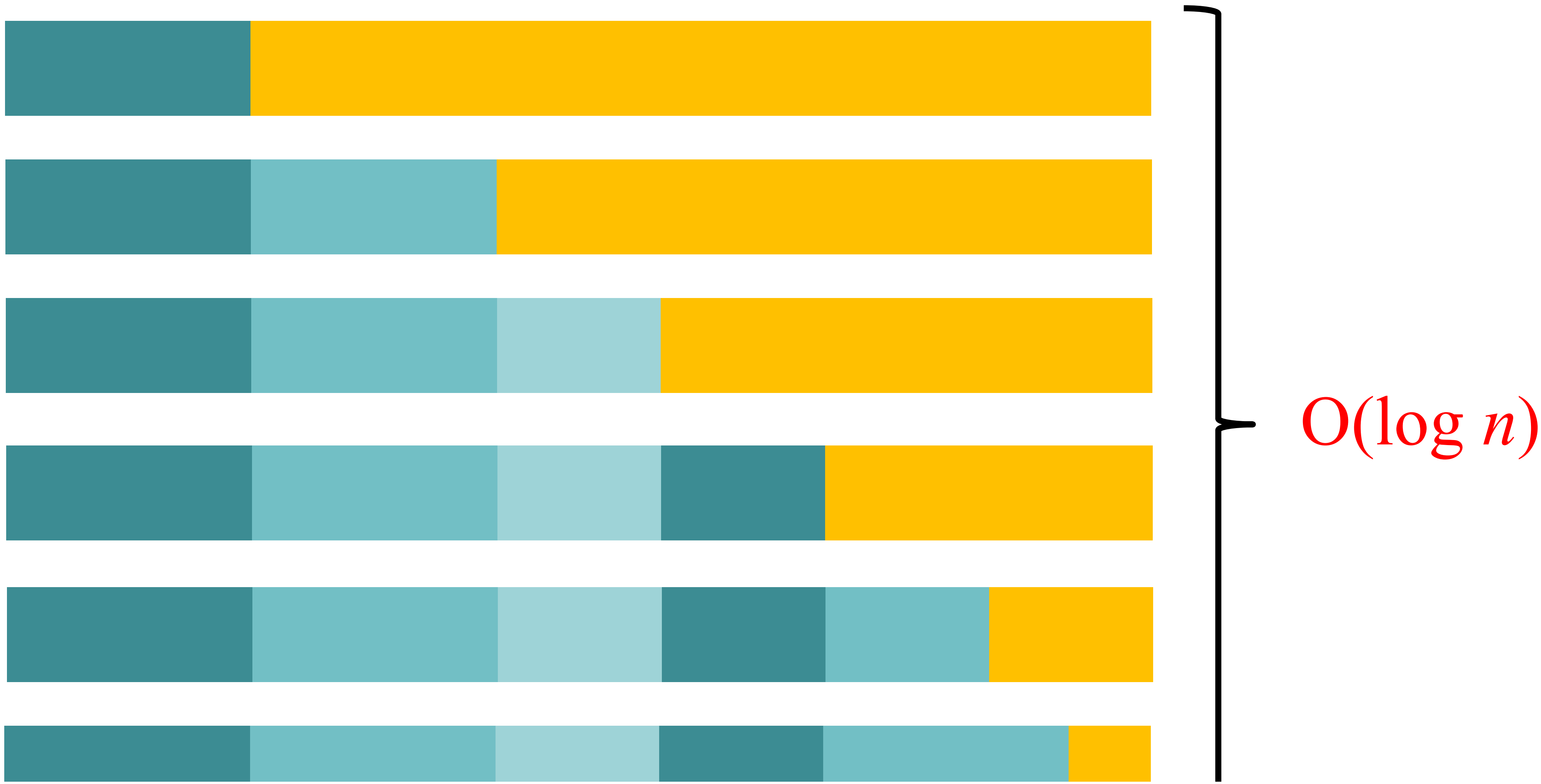
QuickSort Analysis

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

x = QuickSort(A[1..p–1], p–1)

y = QuickSort(A[p+1..n], n–p)

QuickSort

Key Idea:

- Choose the pivot at random.
- Most of the time: split will be at least $\frac{9}{10} : \frac{1}{10}$

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

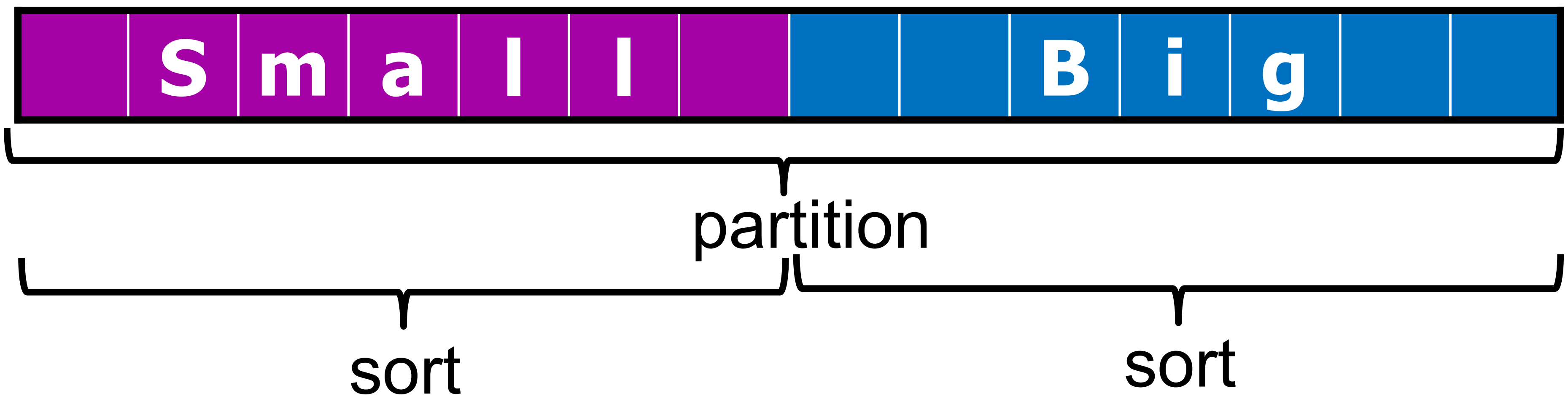
Randomized algorithm:

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

        x = QuickSort(A[1..p–1], p–1)
        y = 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)

1

??

T(k)

T(n-k)

9n/10 > k > n/10

```
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], n
```

Paranoid QuickSort

T(n)

1

O(n)

T(k)

T(n-k)

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

```
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], n
```

Paranoid QuickSort

Key claim:

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

Then we know:

$$T(n) <= T(n/10) + T(9n/10) + n(\text{\# iterations of } \textbf{repeat})$$

$$= O(n \log n)$$

Paranoid QuickSort

T(n) → ParanoidQuickSort(A[1..n], n)

1 → **if** (n == 1) **then** return;

else

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

T(k) → x = QuickSort(A[1..p–1], p

T(n-k) → y = QuickSort(A[p+1..n], n

9n/10 > k > n/10

Probability Theory

CS1231S review!

Probability Theory

Flipping a coin:

- $Pr(heads) = ½$
- $Pr(tails) = ½$

Coin flips are independent:

- $Pr(heads \rightarrow heads) = ½ * ½ = ¼$
- $Pr(heads \rightarrow tails \rightarrow heads) = ½ * ½ * ½ = 1/8$

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. All of the above are the same
e. Incomparable

ARCHIPELAGO is open

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

Set of uniform events (e_1, e_2, e_3, …, e_k):

- Pr(e_1) = 1/k
- Pr(e_2) = 1/k
- …
- Pr(e_k) = 1/k

Probability Theory

Events **A**, **B**:

- Pr(**A**), Pr(**B**)
- **A** and **B** are independent
  (e.g., unrelated random coin flips)

Then:

- Pr(**A** and **B**) = Pr(**A**)Pr(**B**)

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:

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

Expected value of A:

$$E[A] = (1/4)12 + (3/4)60 = 48$$

Probability Theory

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

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

Expected outcome:

$$E[X] = e_1 p_1 + e_2 p_2 + \ldots + e_k p_k$$

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

In two coin flips: I <u>expect</u> one heads.

Probability Theory

Define event **A**:

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

In two coin flips: I <u>expect</u> one heads.

– Pr(heads, heads) = ¼	2 * ¼	=	½
– Pr(heads, tails) = ¼	1 * ¼	=	¼
– Pr(tails, heads) = ¼	1 * ¼	=	¼
– Pr(tails, tails) = ¼	0 * ¼	=	0
			1

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

In two coin flips: I <u>expect</u> one heads.

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

Goal: calculate <u>expected</u> time of QuickSort

Worst-Case Analysis

Randomized algorithm:

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

Expected worst-case running time:

- For all possible inputs, which one yields the maximum *expected* running time.

Probability Theory

Linearity of Expectation:

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

Example:

- A = # heads in 2 coin flips: E[A] = 1
- B = # heads in 2 coin flips: A[B] = 1
- A + B = # heads in 4 coin flips

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

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

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(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

E[X]= Pr(heads after 1 flip)*1 +
Pr(heads after 2 flips)*2 +
Pr(heads after 3 flips)*3 +
Pr(heads after 4 flips)*4 +
…

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

E[X]= Pr(H)*1 +

Pr(T H)*2 +

Pr(T T H)*3 +

Pr(T T T H)*4 +

…

Probability Theory

Flipping an (unfair) coin:

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

How many flips to get at least one head?

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

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(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 <u>more</u> flips to get a head?

Idea: If I flip "tails," the expected number of additional flips to get a "heads" is <u>still</u> **E**[X]!!

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

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

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

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(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(heads) = p
- Pr(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(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

If p = ½, the expected number of flips to get one head equals:

$$\mathbf{E}[X] = 1/p = 1/½ = 2$$

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)
```

Today: more sorting!

QuickSort:

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