CS2040S
Data Structures and Algorithms

Welcome!

Last Time: Sorting, Part I

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

Today: Sorting, Part II

QuickSort

- Divide-and-Conquer
- Paranoid QuickSort
- Randomized Analysis

Sorting

Problem definition:

Input: array A[1..n] of words / numbers

Output: array B[1..n] that is a permutation of A such that:

$B[1] \leq B[2] \leq \ldots \leq B[n]$

Example:

$A = [9, 3, 6, 6, 6, 4] \rightarrow [3, 4, 6, 6, 6, 9]$

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)

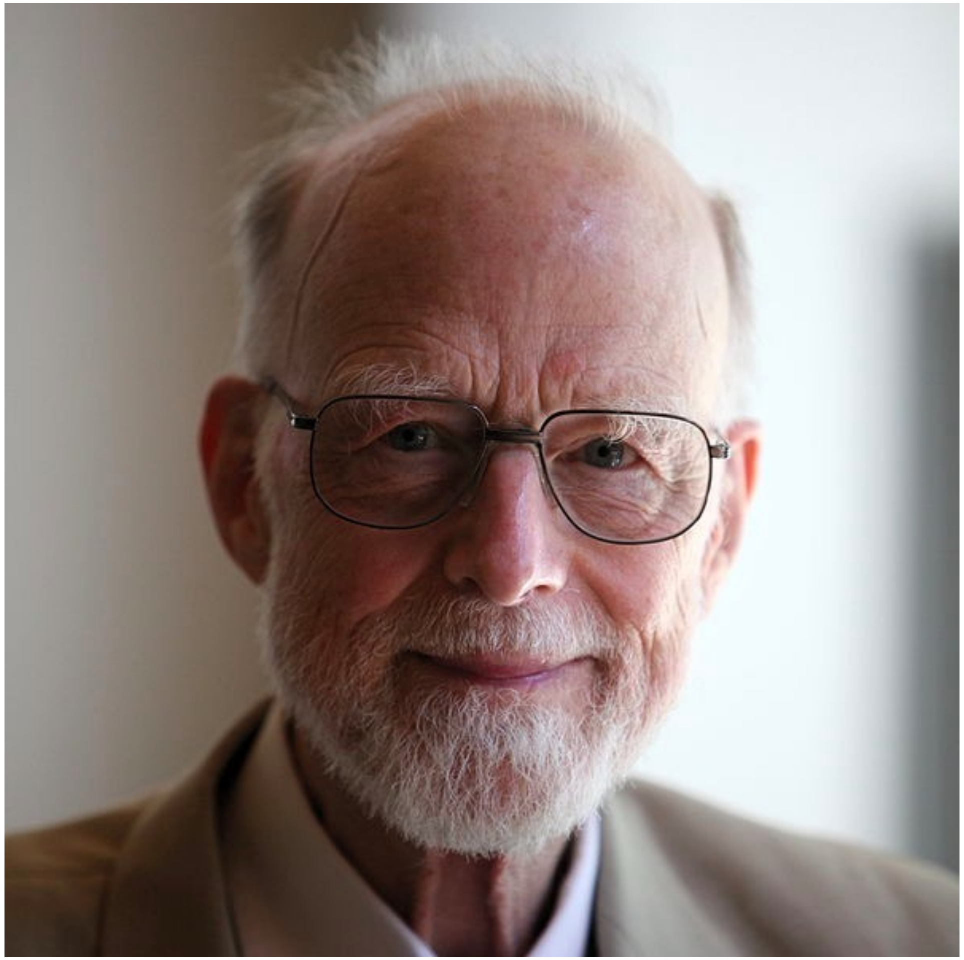


Photo: Wikimedia Commons (Rama)

Hoare

Quote:

"There are two ways of constructing a software design:

One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult."

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

"Engineering a sort function"

Yet in the summer of 1991 our colleagues Allan Wilks and Rick Becker found that a qsort run that should have taken a few minutes was chewing up hours of CPU time. Had they not interrupted it, it would have gone on for weeks. They found that it took n^2 comparisons to sort an 'organ-pipe' array of 2n integers: 123..nn.. 321.

QuickSort Today

1960: Invented by Hoare

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

1993: Bentley & McIlroy improvements

”Ok, QuickSort is done,” said everyone.

Every algorithms class since 1993:

Punk in the front row:

“But what if we used more pivots?”

Every algorithms class since 1993:

Punk in the front row:

“But what if we used more pivots?”

Professor:

“Doesn’t work. I can prove it.
Let’s get back to the syllabus....”

In 2009:

Punk in the front row:

"But what if we used more pivots?"

Professor:

"Doesn't work. I can prove it.
Let's get back to the syllabus...."

Punk in the front row:

"Huh... let me try it. Wait a sec, it's faster!"

QuickSort Today

1960: Invented by Hoare

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

1993: Bentley & McIlroy improvements

2009: Vladimir Yaroslavskiy

- Dual-pivot Quicksort !!!
- Now standard in Java
- 10% faster!

QuickSort Today

1960: Invented by Hoare

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

1993: Bentley & McIlroy improvements

2009: Vladimir Yaroslavskiy

- Dual-pivot Quicksort !!!
- Now standard in Java
- 10% faster!

2012: Sebastian Wild and Markus E. Nebel

- "Average Case Analysis of Java 7's Dual Pivot…"
- Best paper award at ESA

Moral of the story:

1) Don’t just listen to me. Go try it!

2) Even “classical” algorithms change. QuickSort in 5 years may be different than QuickSort I am teaching today.

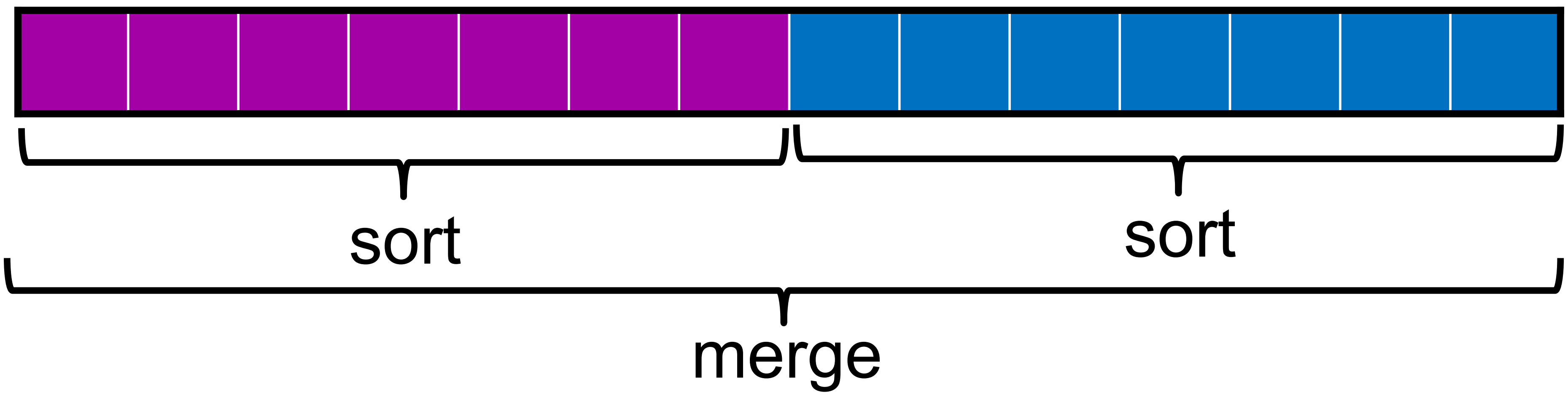
QuickSort

In class:

- Easy to understand! (divide-and-conquer…)
- Moderately hard to implement correctly.
- Harder to analyze. (Randomization…)
- Challenging to optimize.

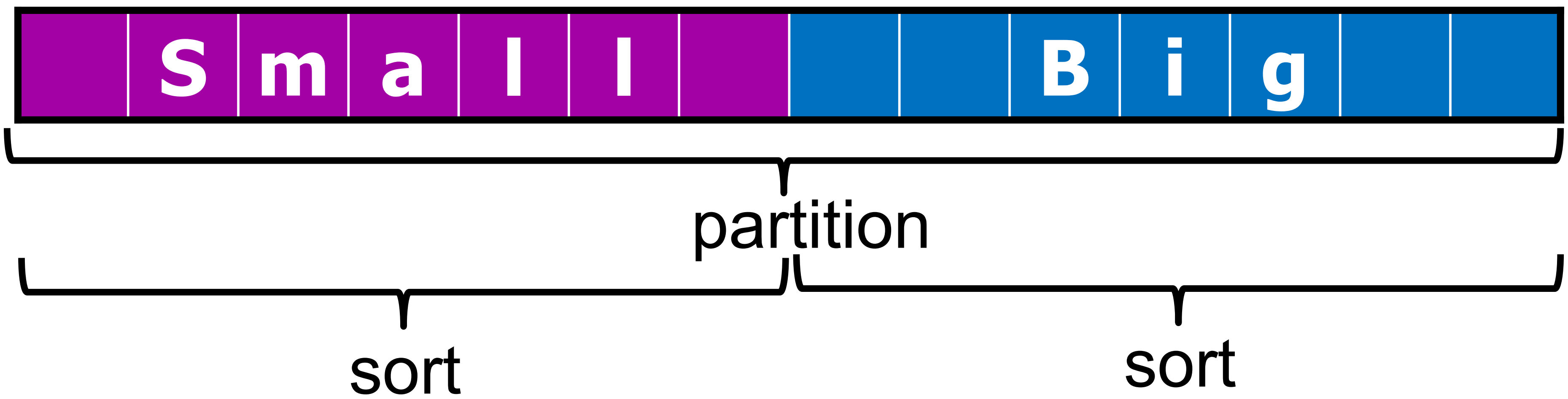
Recall: MergeSort

```
MergeSort(A[1..n], n)
    if (n==1) then return;
    else
        x = MergeSort(A[1..n/2], n/2)
        y = MergeSort(A[n/2+1..n], n/2)
        return merge(x, y, n/2)
```



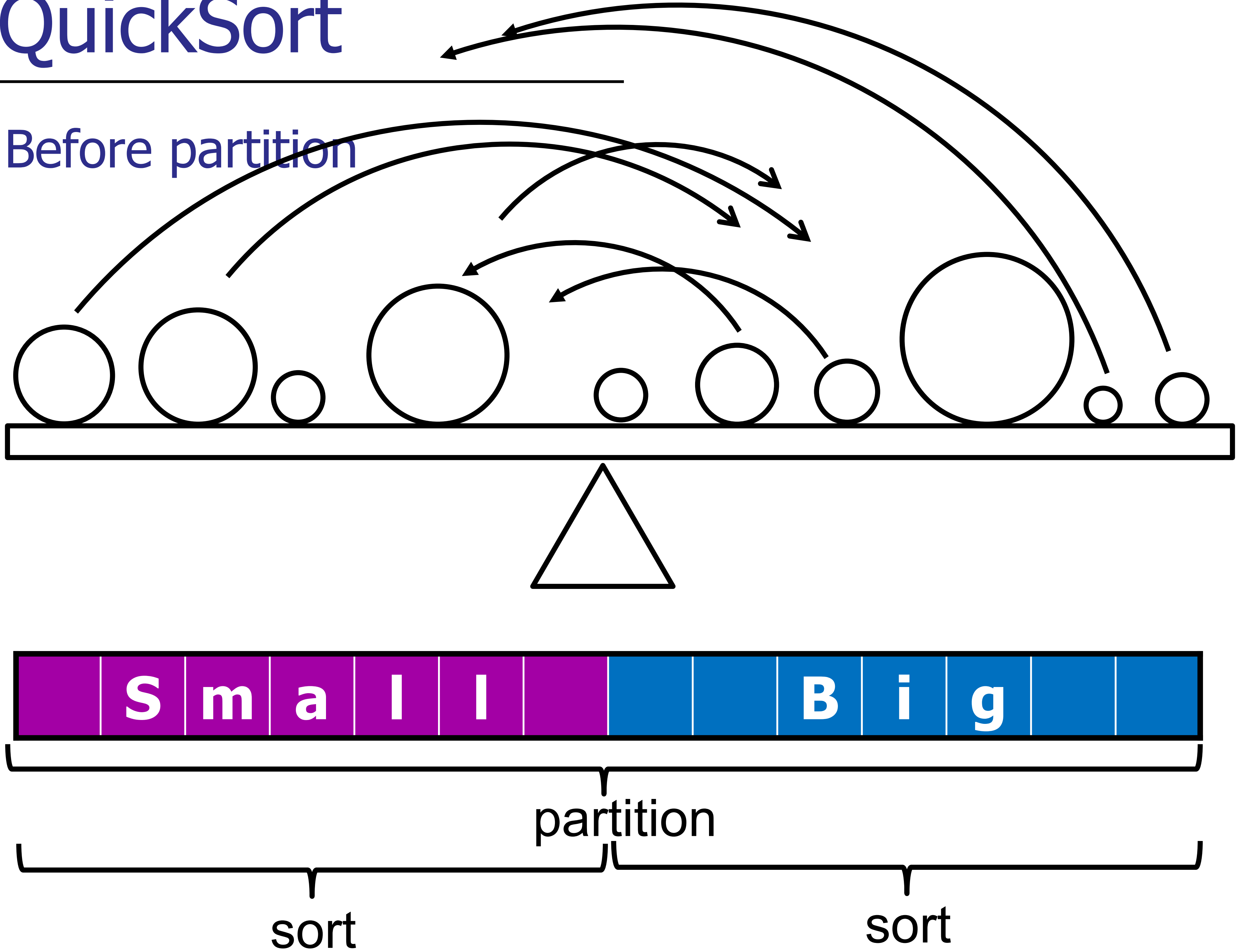
QuickSort

```
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)
```



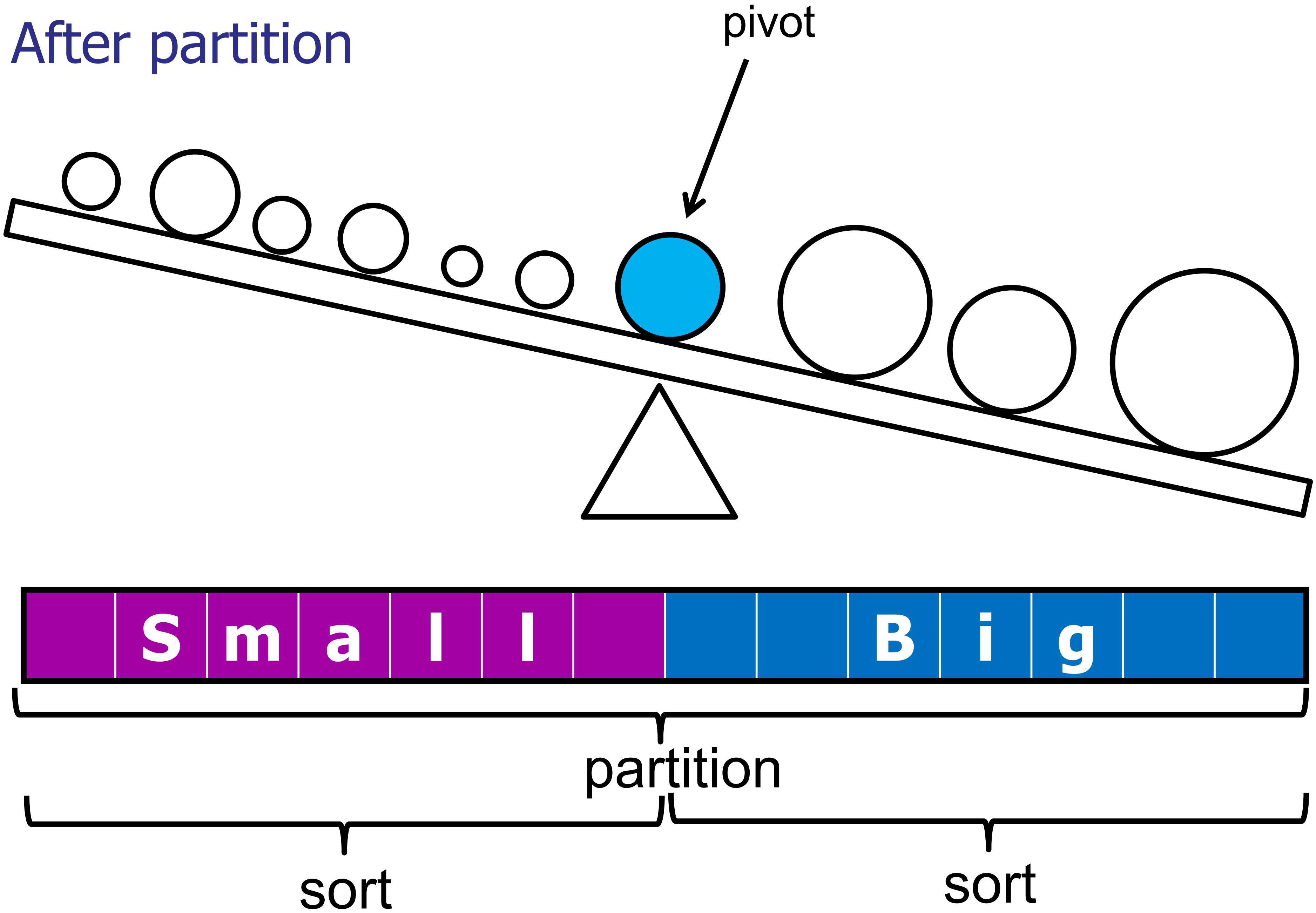
QuickSort

Before partition

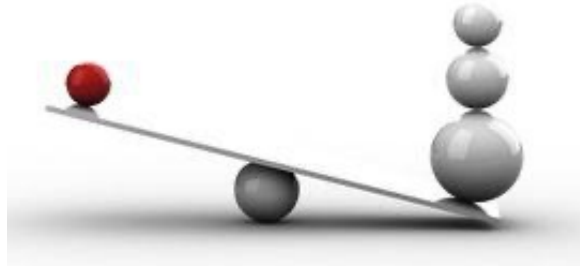


QuickSort

After partition



QuickSort



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

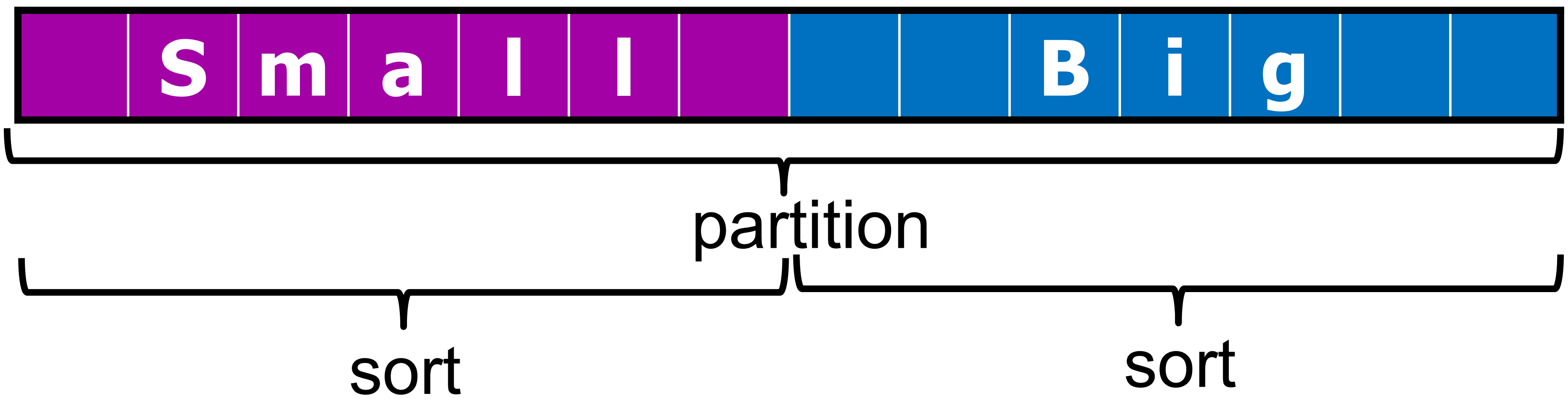
if (n==1) **then** return;

else

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

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

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



QuickSort

Given: n element array $A[1..n]$

1. Divide: Partition the array into two sub-arrays around a ***pivot*** x such that elements in lower subarray $\leq x \leq$ elements in upper sub-array.

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

2. Conquer: Recursively sort the two sub-arrays.
3. Combine: Trivial, do nothing.

Key: efficient *partition* sub-routine

Partitioning an Array

Three steps:

1. Choose a pivot.
2. Find all elements smaller than the pivot.
3. Find all elements larger than the pivot.

$< x$	x	$> x$

Quicksort

Example:

6 3 9 8 4 2

Quicksort

Example:

6	3	9	8	4	2
3	4	2	6	9	8

Quicksort

Example:

6	3	9	8	4	2
3	4	2	6	9	8
2	3	4			

Quicksort

Example:

6	3	9	8	4	2
3	4	2	6	9	8
2	3	4		8	9

Quicksort

Example:

6	3	9	8	4	2
3	4	2	$\boxed{6}$	9	8
2	3	4	6	8	9

Quicksort

Example:

6	3	9	8	4	2
3	4	2	6	9	8
2	3	4	6	8	9

The following array has been partitioned around which element?

18	5	6	1	10	22	40	32	50

a. 6

b. 10

c. 22

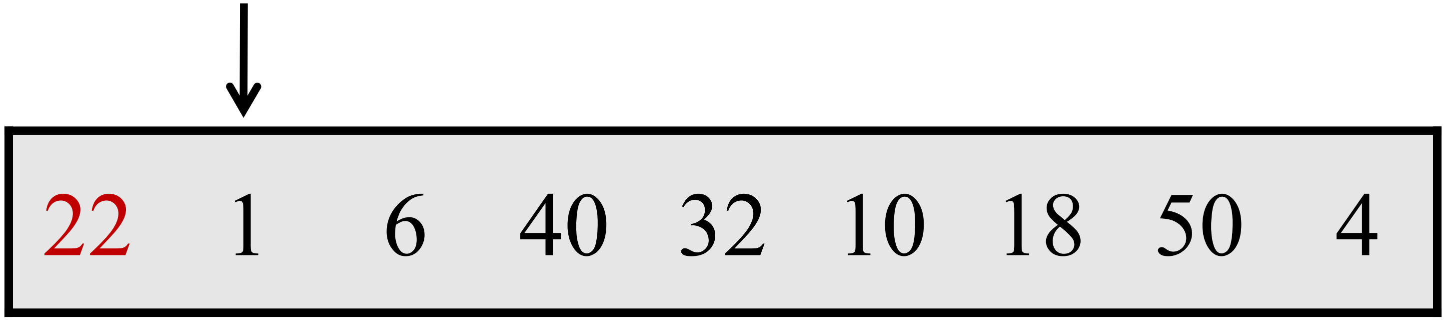
d. 40

e. 32

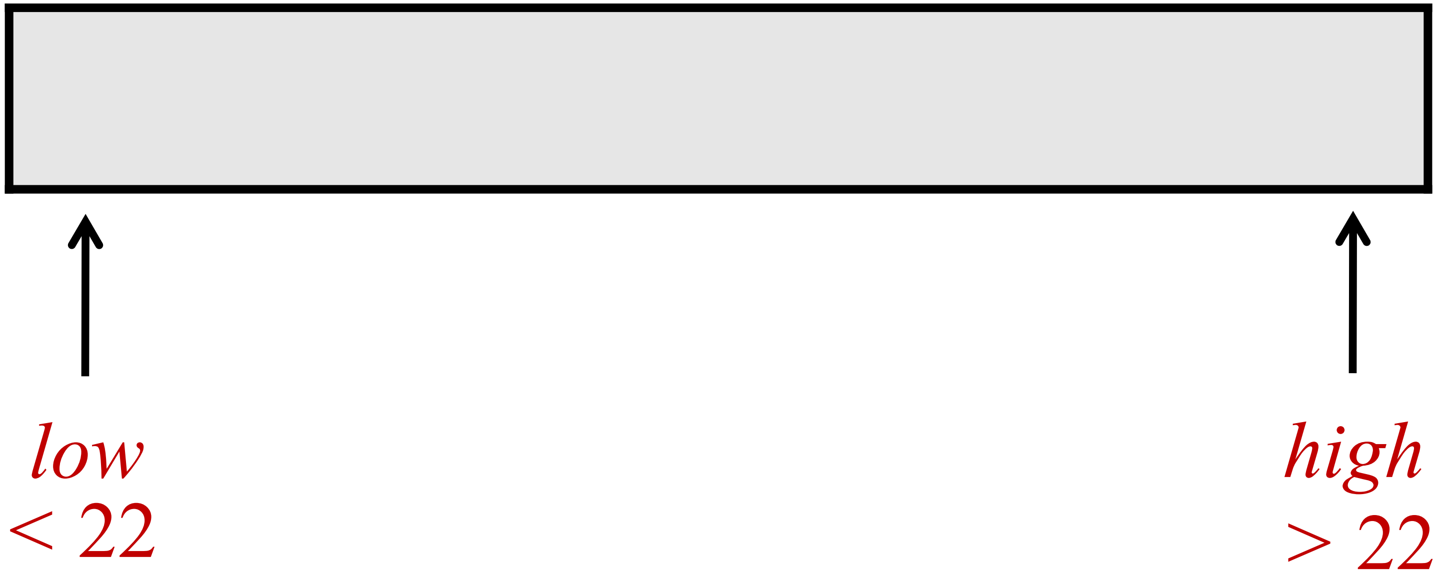
f. I don’t know.

Partitioning an Array

Example: partition around 22

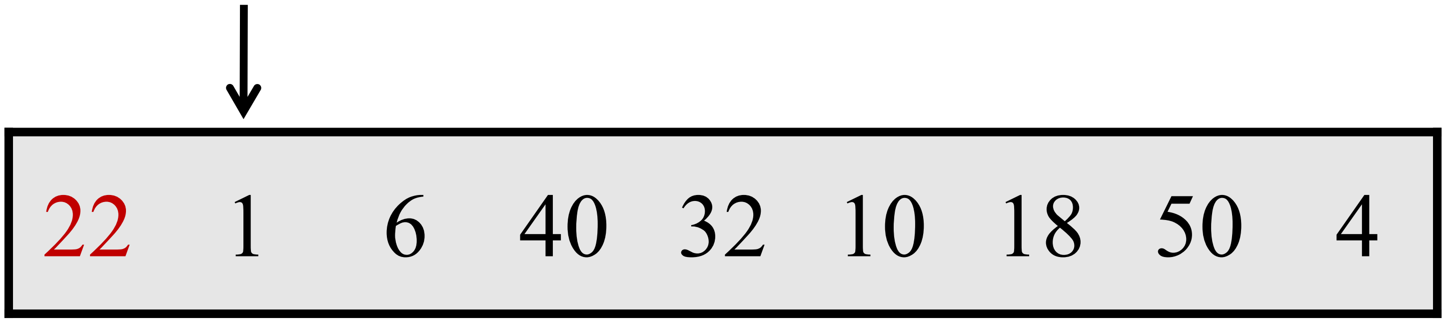


Output array:

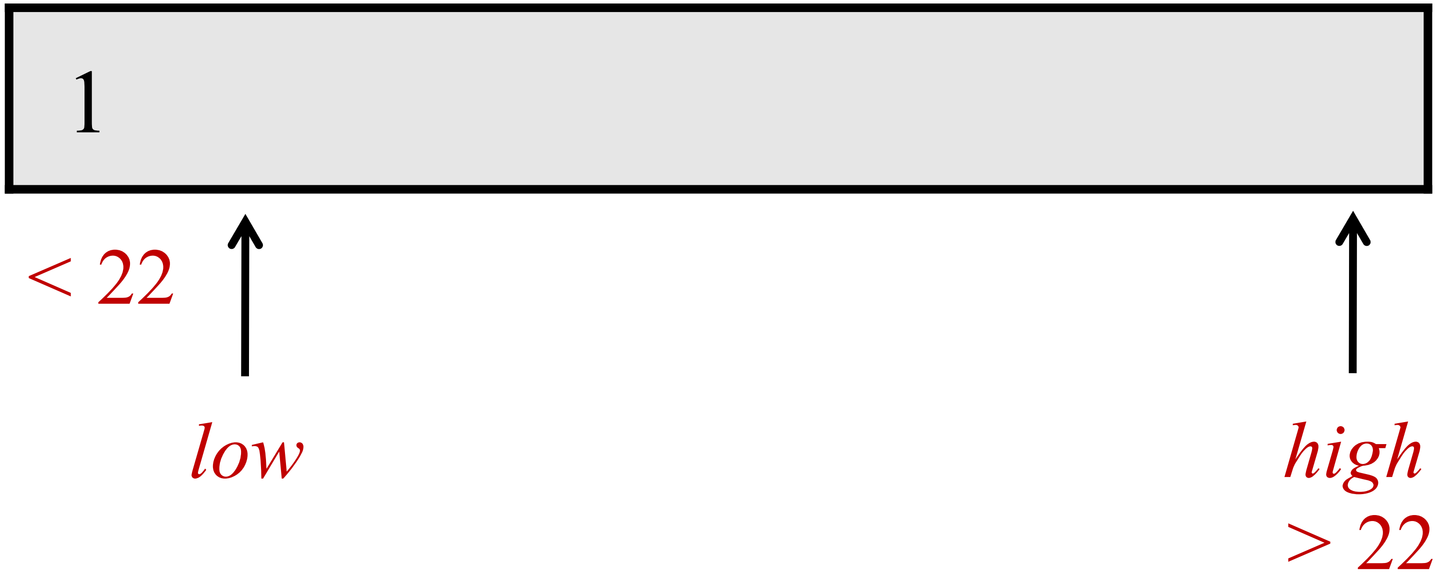


Partitioning an Array

Example: partition around 22

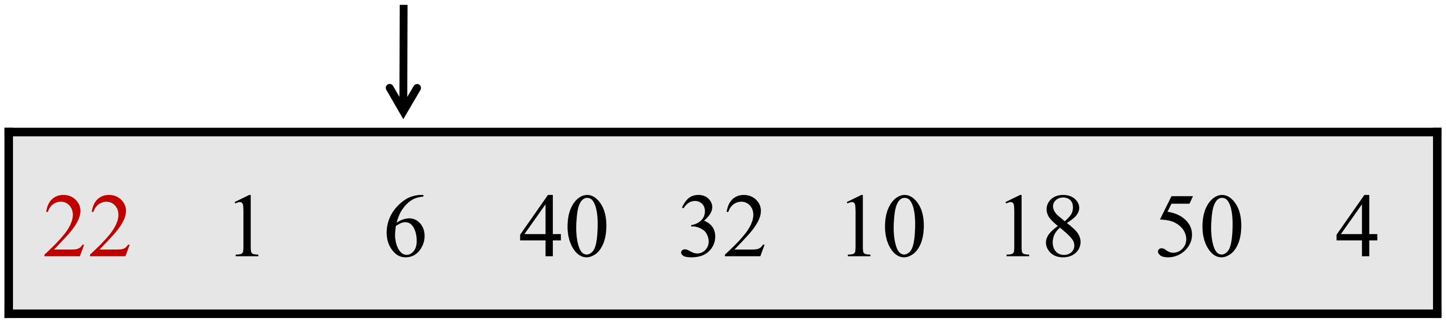


Output array:

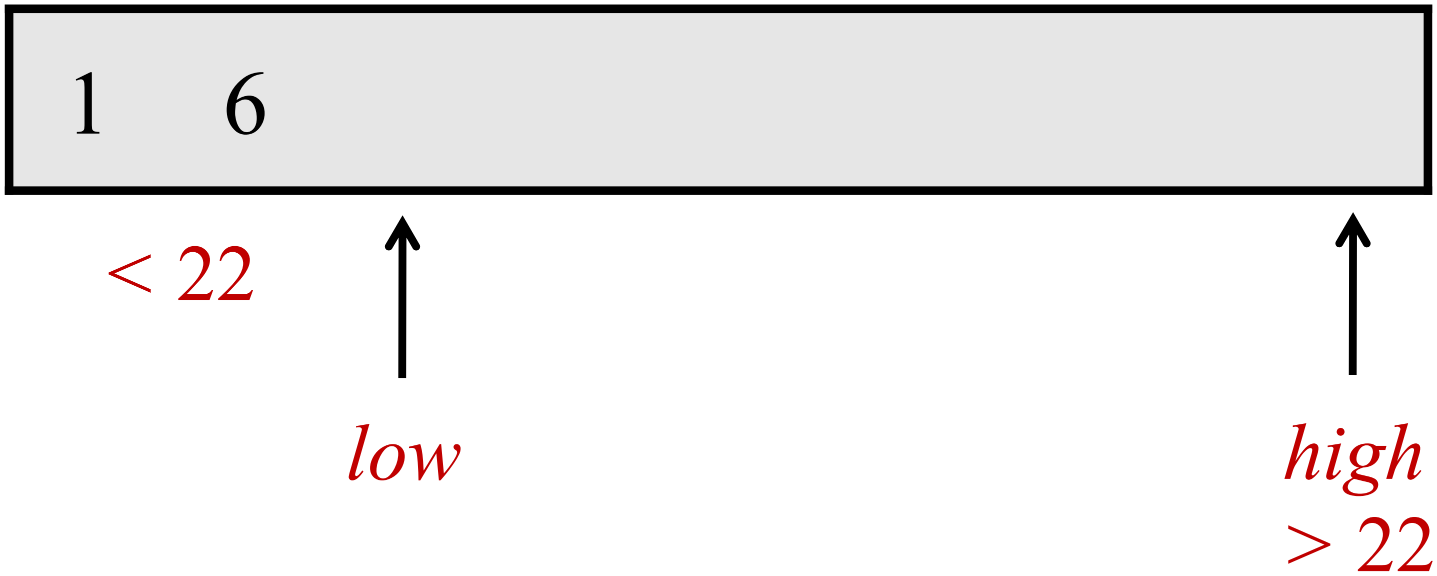


Partitioning an Array

Example: partition around 22

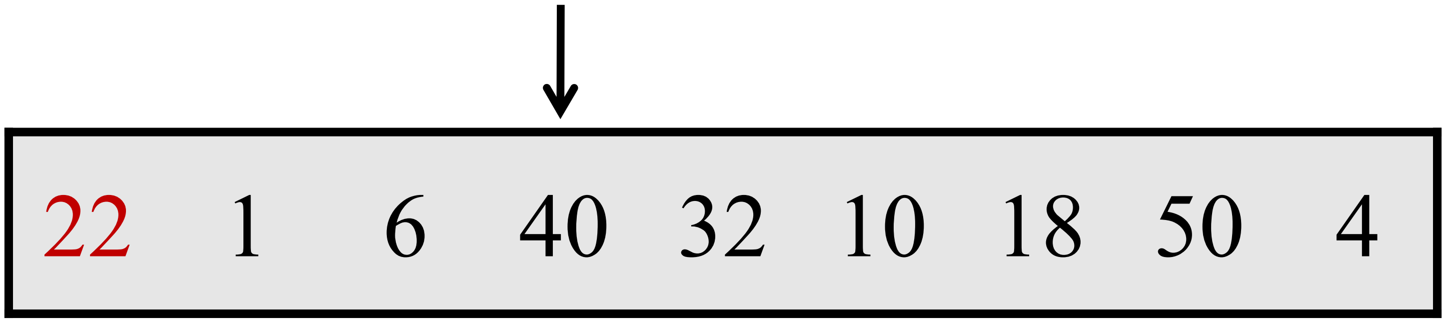


Output array:

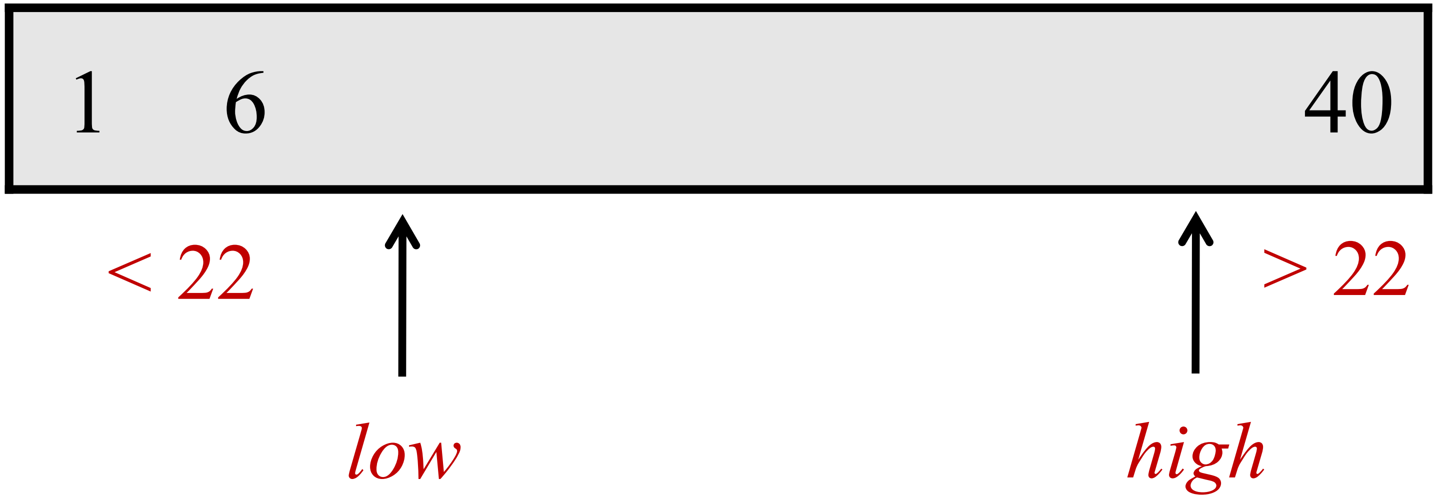


Partitioning an Array

Example: partition around 22

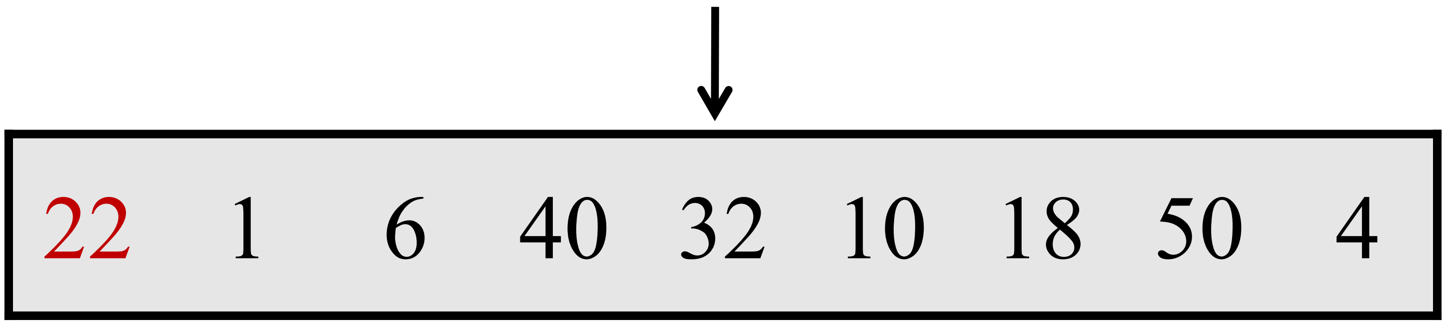


Output array:

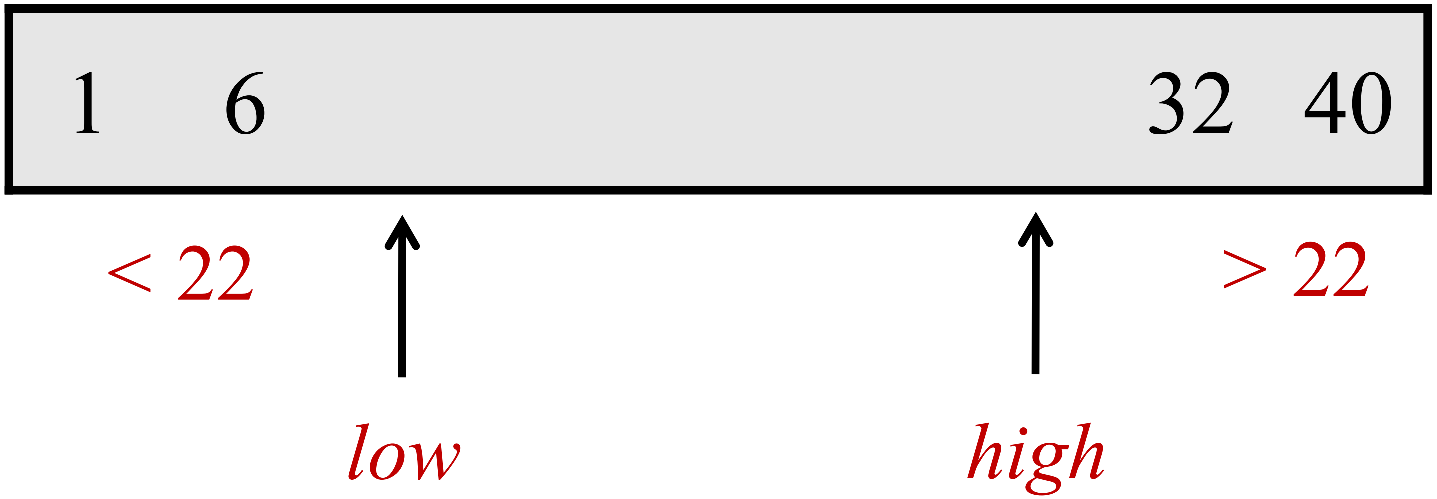


Partitioning an Array

Example: partition around 22

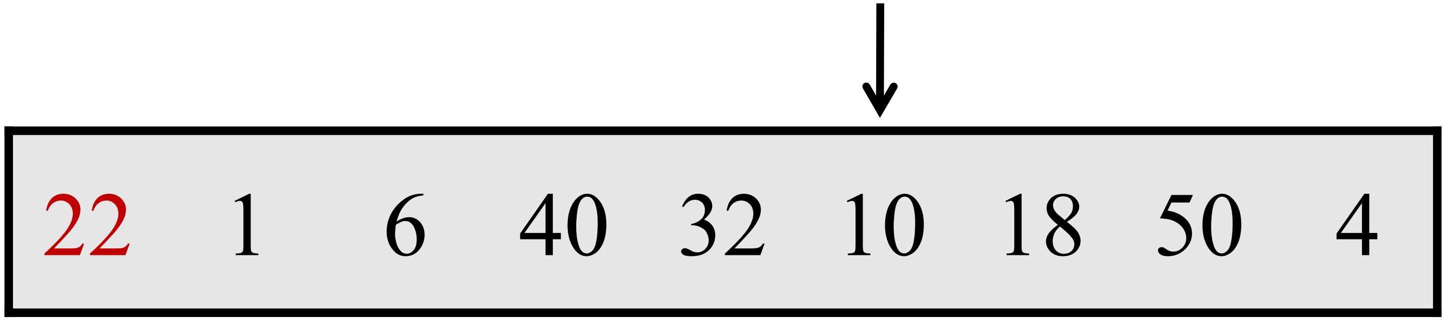


Output array:

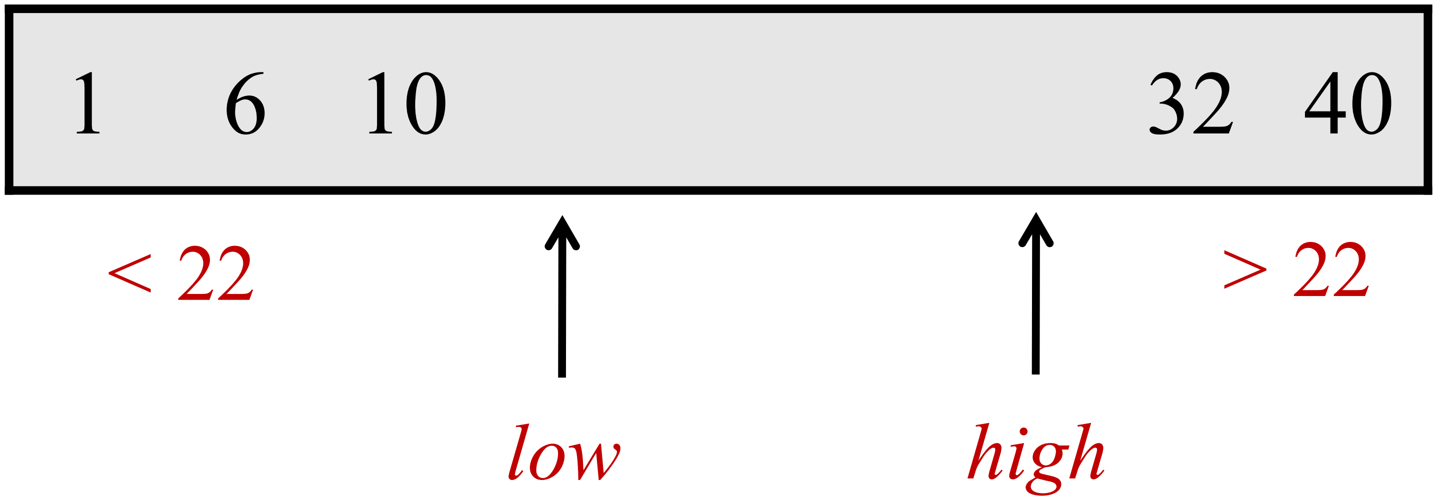


Partitioning an Array

Example: partition around 22

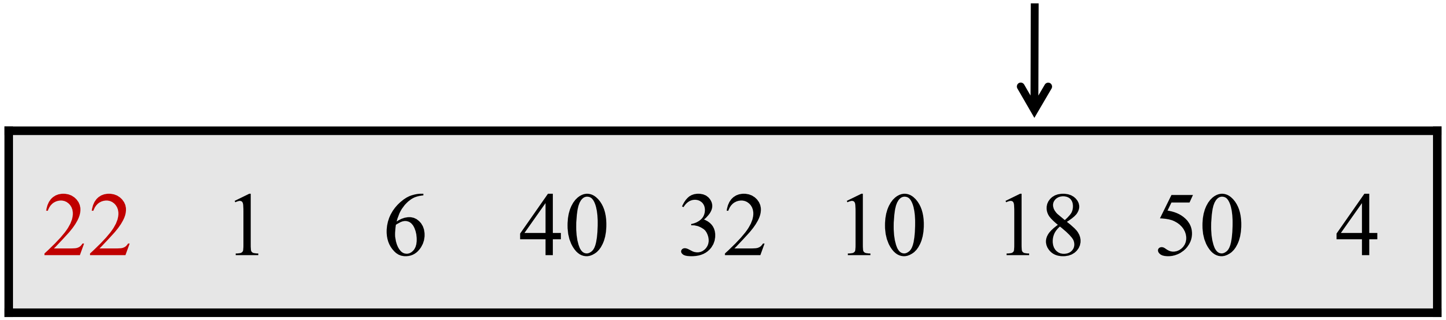


Output array:

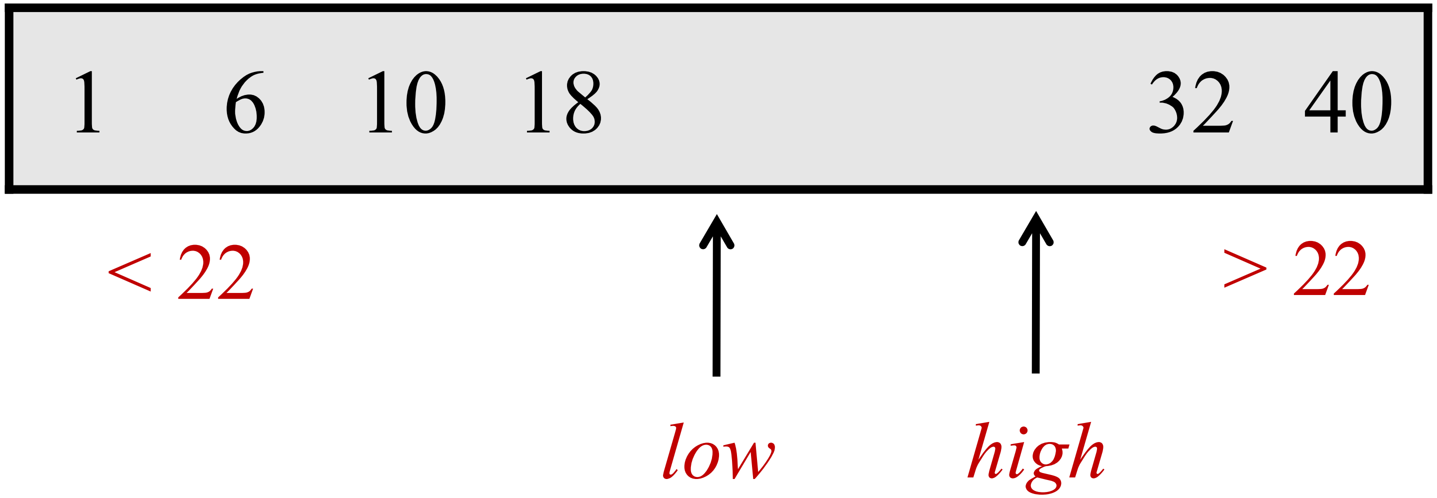


Partitioning an Array

Example: partition around 22



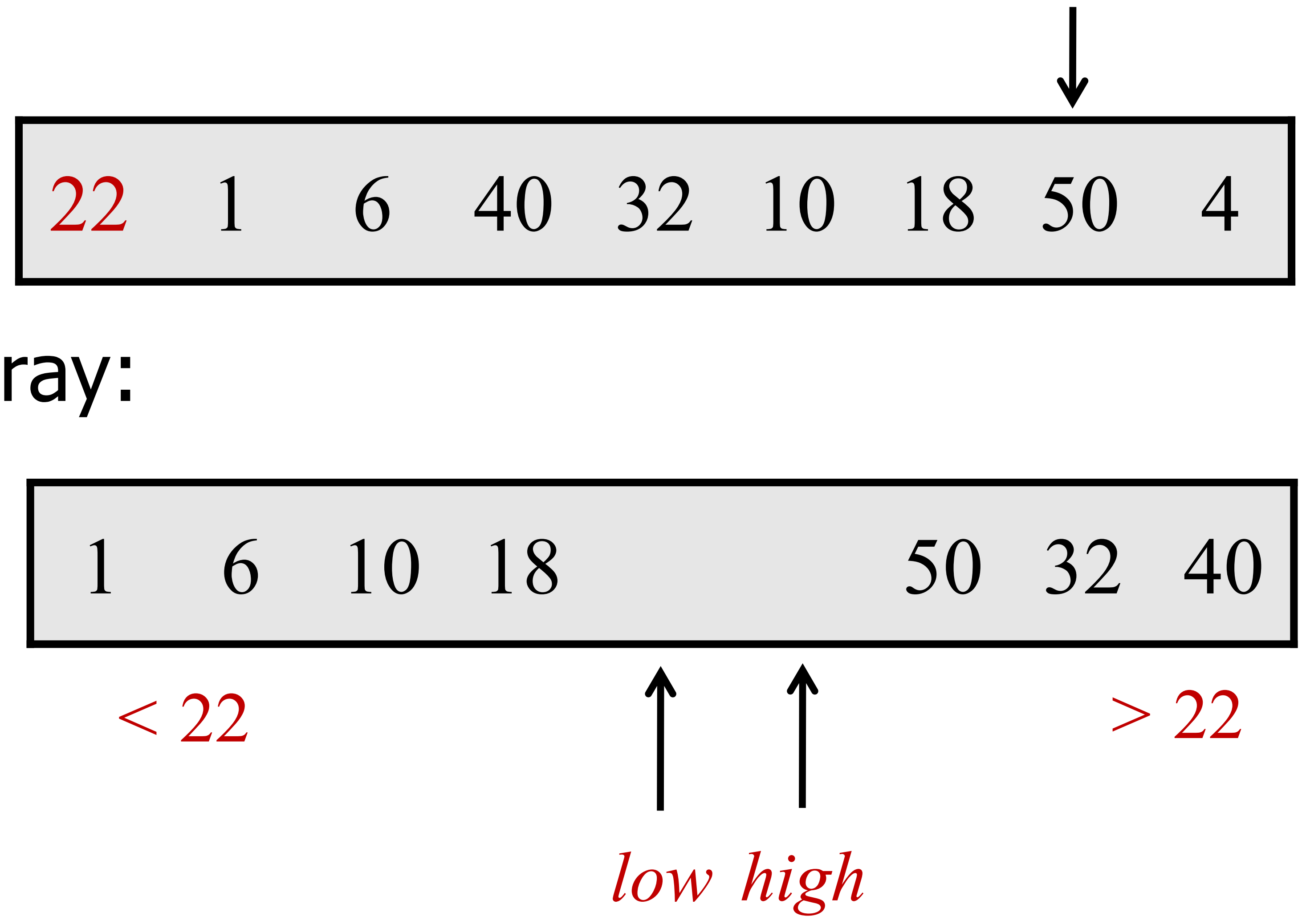
Output array:



Partitioning an Array

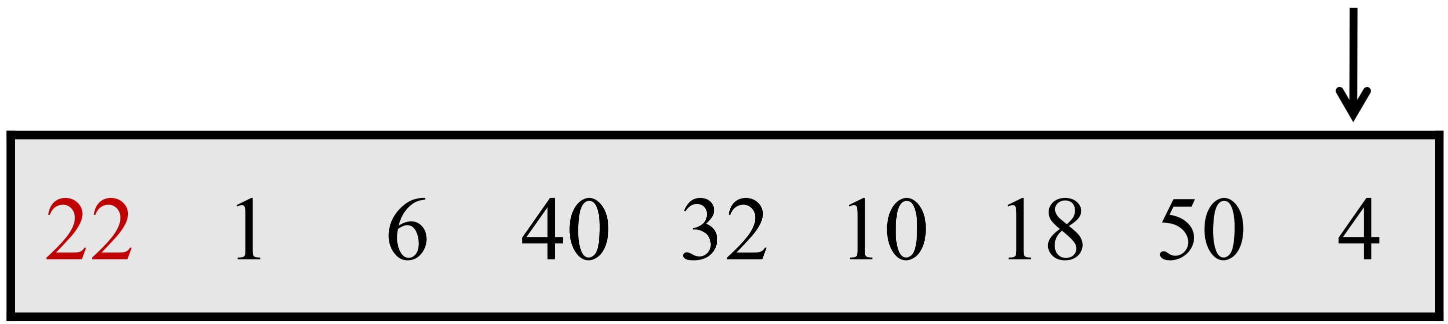
Example: partition around 22

Output array:

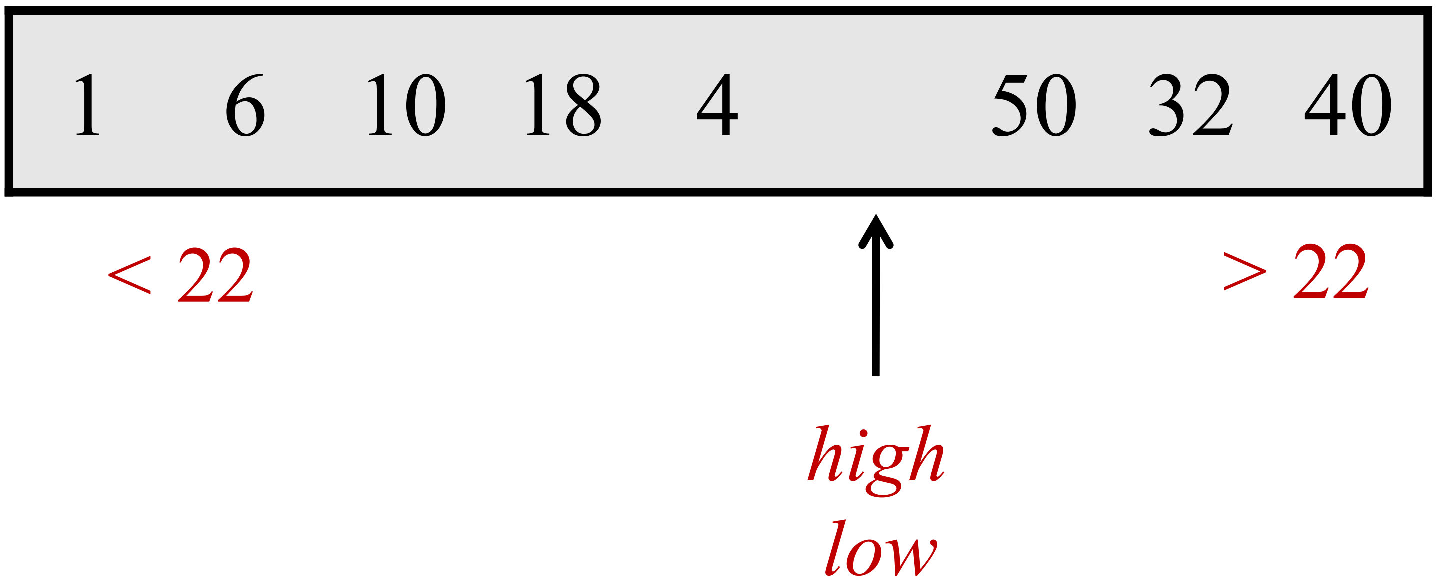


Partitioning an Array

Example: partition around 22



Output array:



Partitioning an Array

Example: partition around 22

22	1	6	40	32	10	18	50	4

Output array:

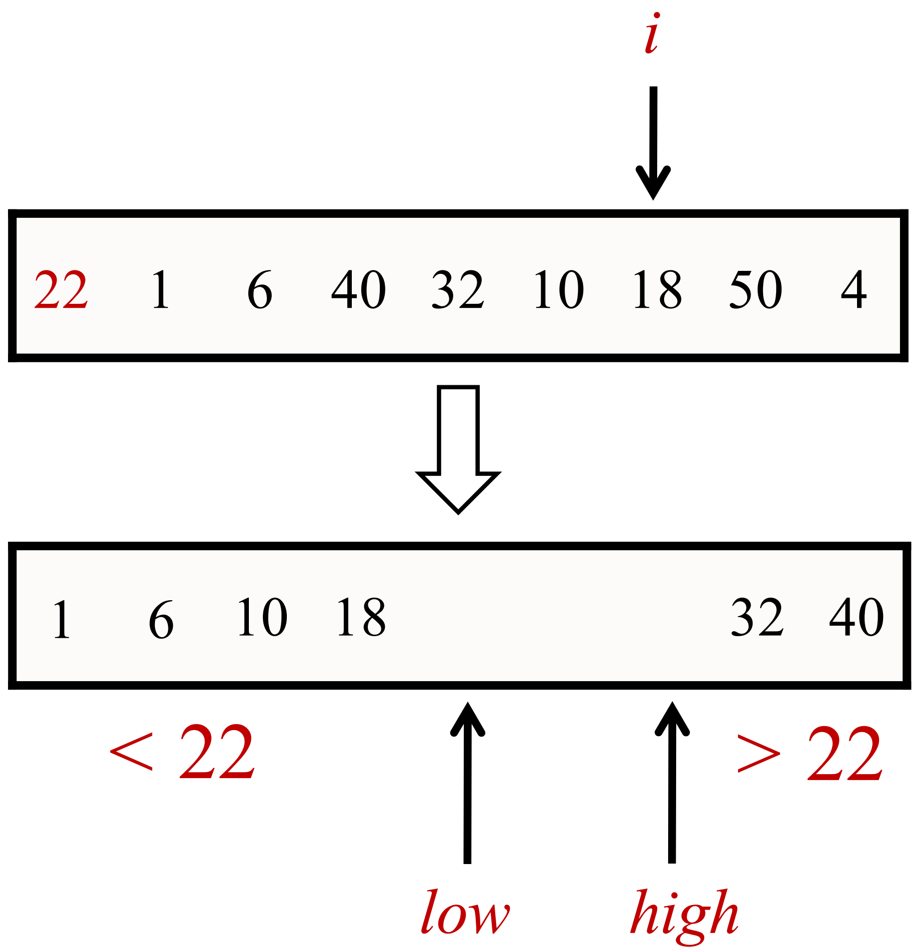
1	6	10	18	4	22	50	32	40

< 22 $\qquad$ > 22

high
low

```
partition(A[2..n], n, pivot)
    B = new n element array
    low = 1;
    high = n;
    for (i = 2; i<= n; i++)
        if (A[i] < pivot) then
            B[low] = A[i];
            low++;
        else if (A[i] > pivot) then
            B[high] = A[i];
            high– – ;
    B[low] = pivot;
    return < B, low >
```

// Assume no duplicates



Partition

Claim: array B is partitioned around the pivot

Proof:

Invariants:

1. For every $i < low : B[i] < pivot$
2. For every $j > high : B[j] > pivot$

In the end, every element from A is copied to B.

Then: $B[i] = pivot$

By invariants, B is partitioned around the pivot.

Partitioning an Array

Example:

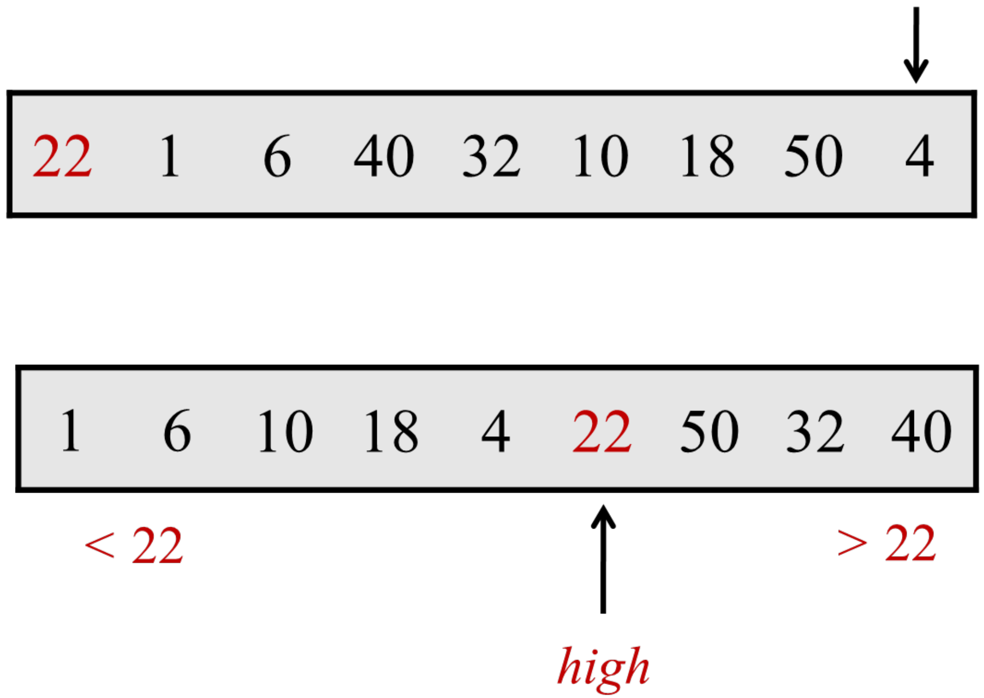
22	1	6	40	32	10	18	50	4

What is the running time of partition?

1. $O(\log n)$
2. ✓ $O(n)$
3. $O(n \log n)$
4. $O(n^2)$
5. I have no idea.

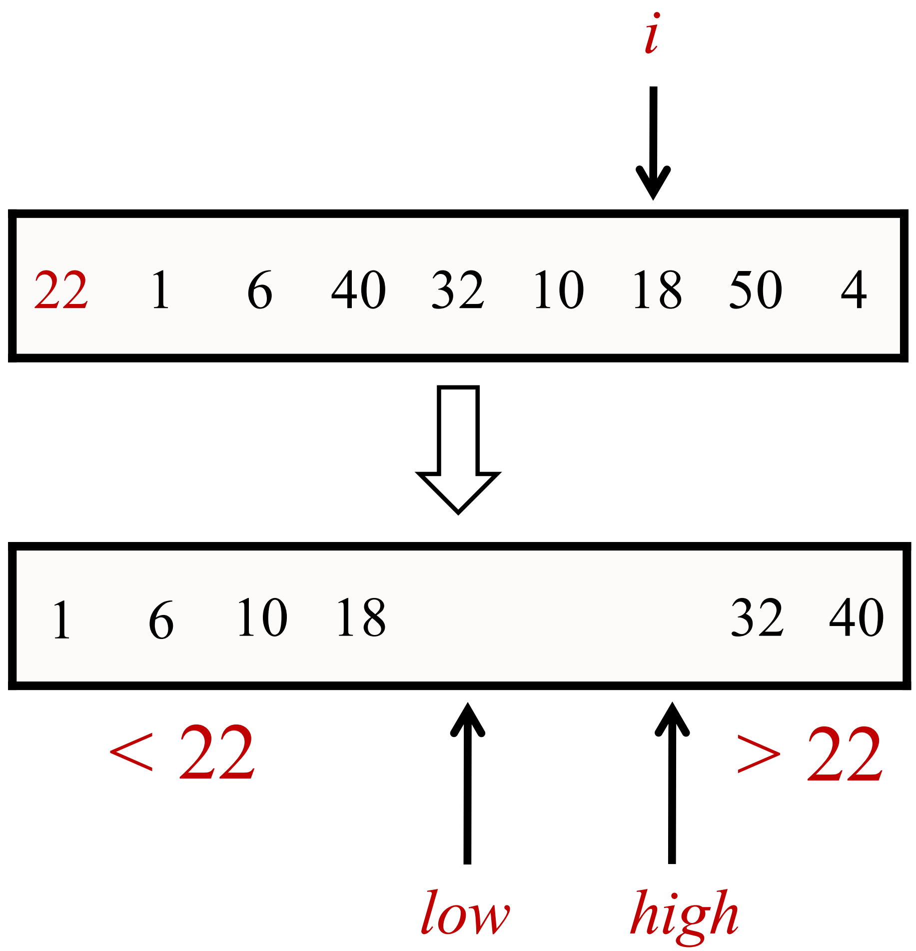
Any bugs?

Anything that can be improved?



```
partition(A[2..n], n, pivot)
    B = new n element array
    low = 1;
    high = n;
    for (i = 2; i<= n; i++)
        if (A[i] < pivot) then
                B[low] = A[i];
                low++;
        else if (A[i] > pivot) then
                B[high] = A[i];
                high– – ;
    B[low] = pivot;
     return  < B, low >
```

// Assume no duplicates



Partitioning an Array "in-place"

Example: partition around 22

22	1	6	40	32	10	18	4	50

low
< 22

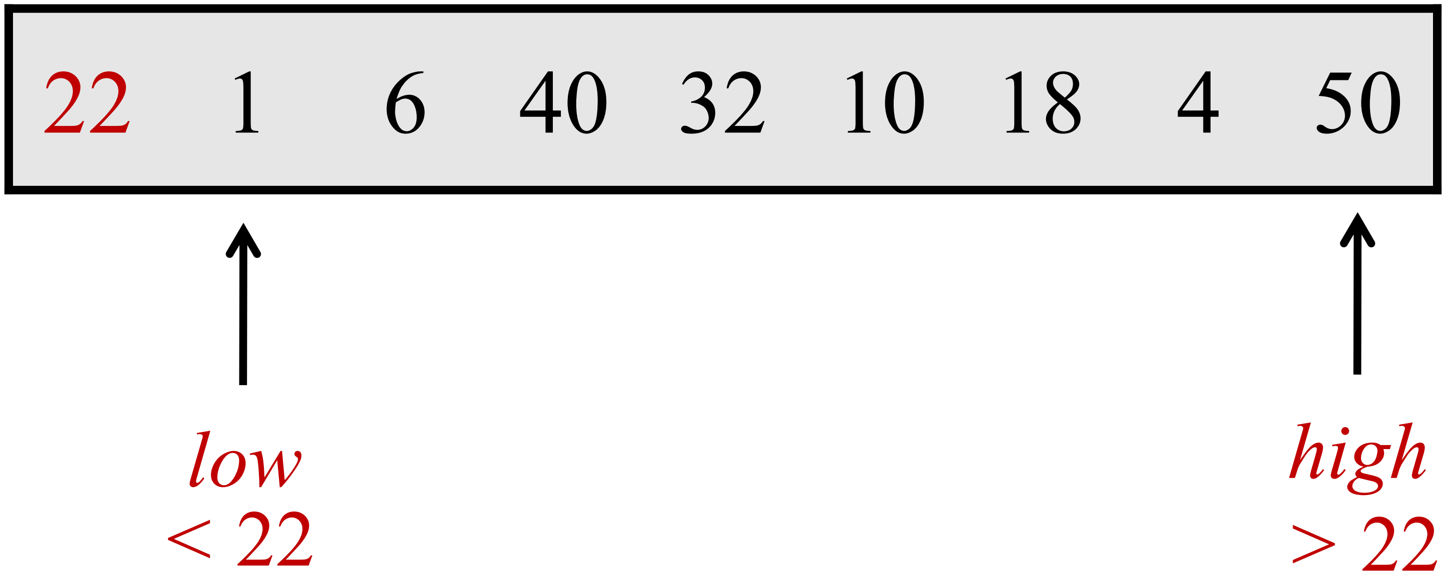
high
> 22

Move until it's bigger than the pivot

Move until it's less than the pivot

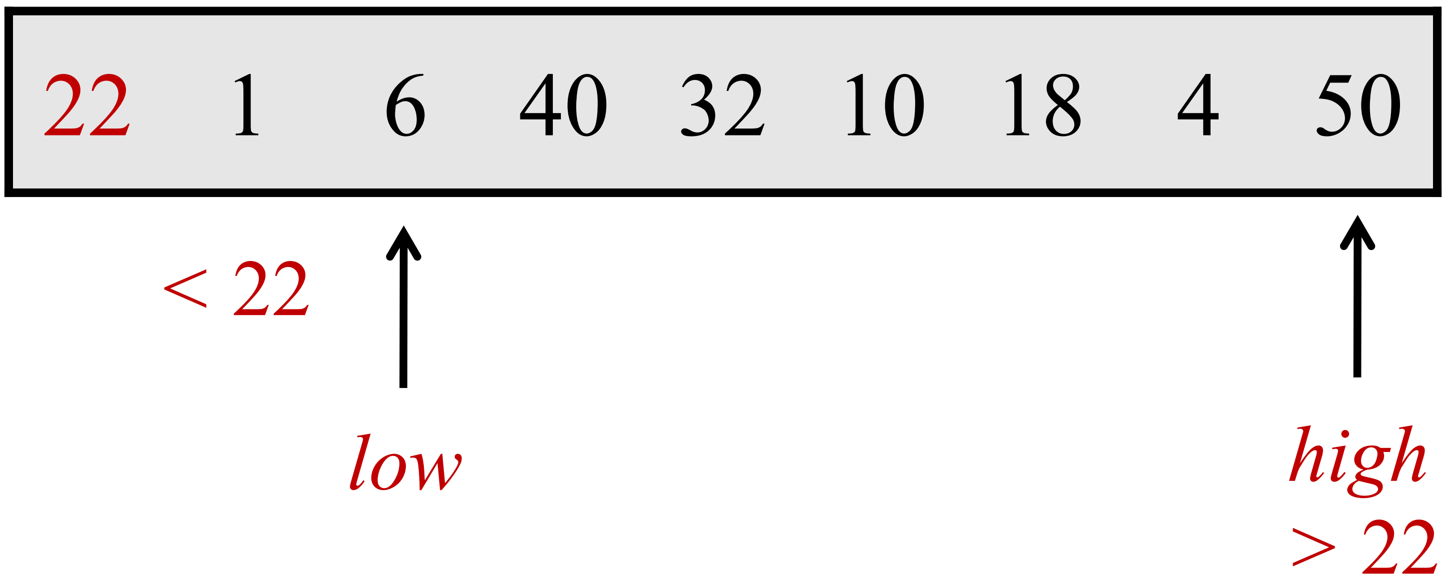
Partitioning an Array

Example: partition around 22



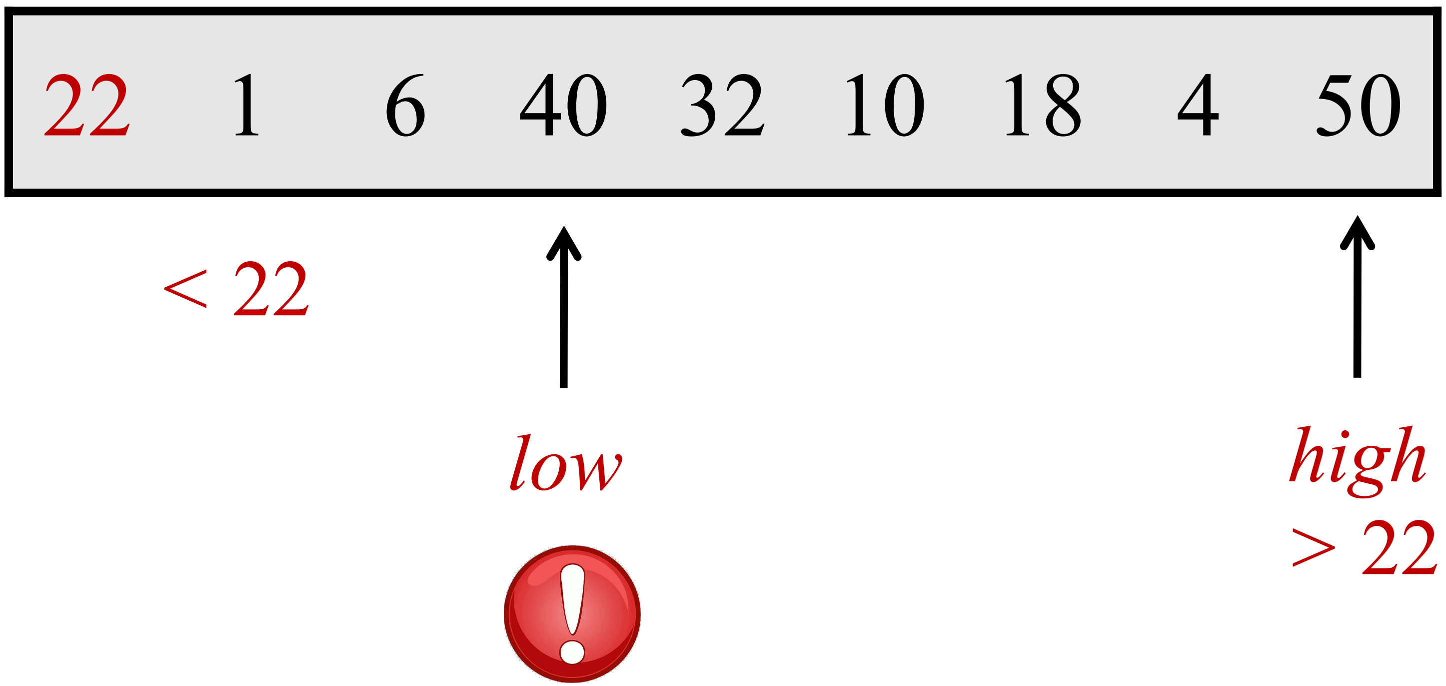
Partitioning an Array

Example: partition around 22



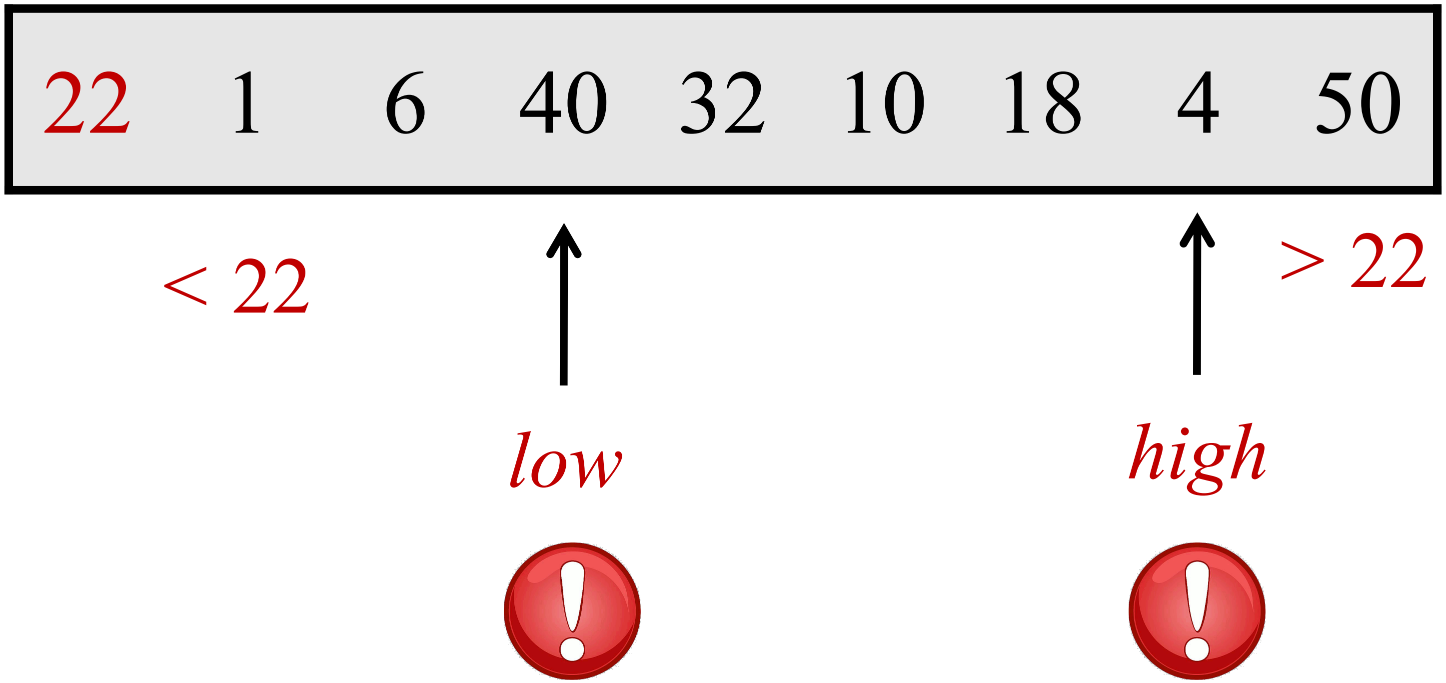
Partitioning an Array

Example: partition around 22



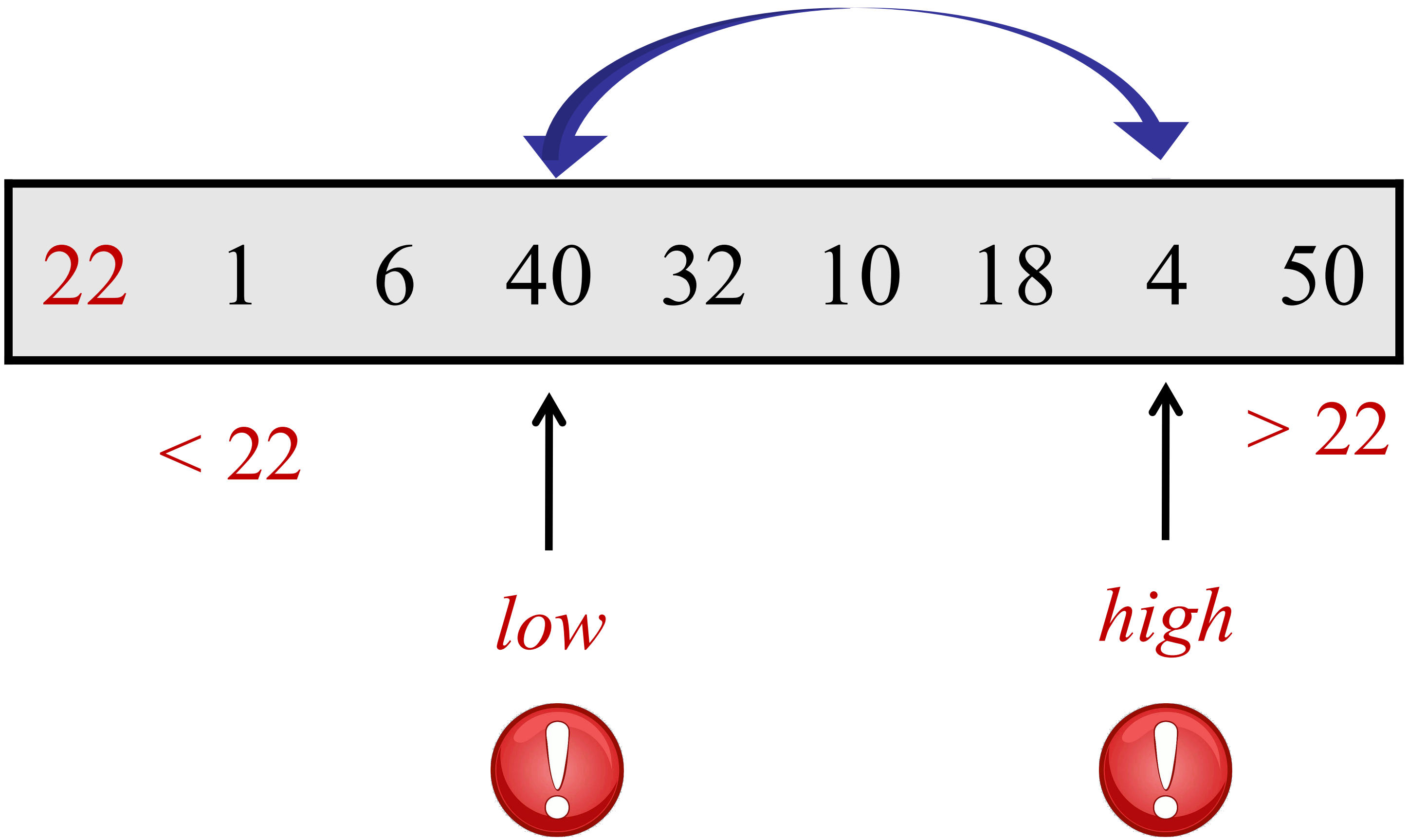
Partitioning an Array

Example: partition around 22



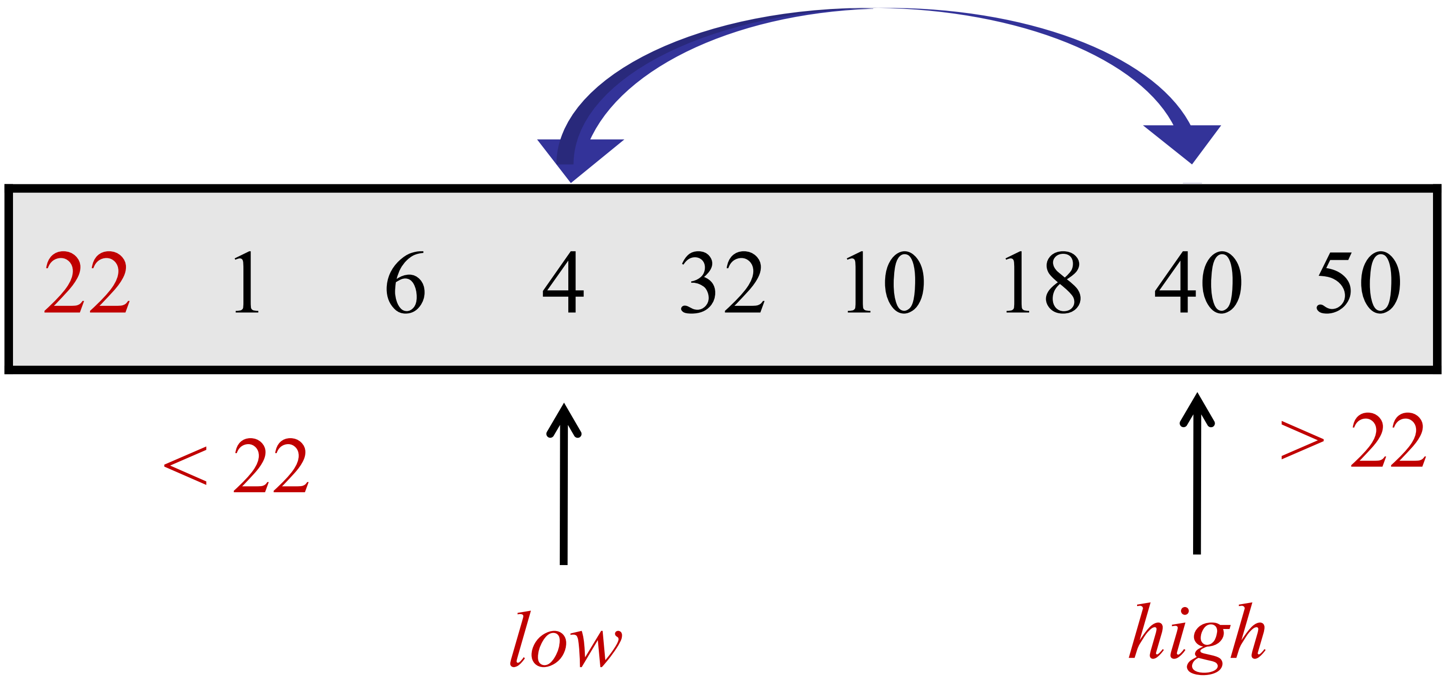
Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22

22	1	6	4	32	10	18	40	50

< 22

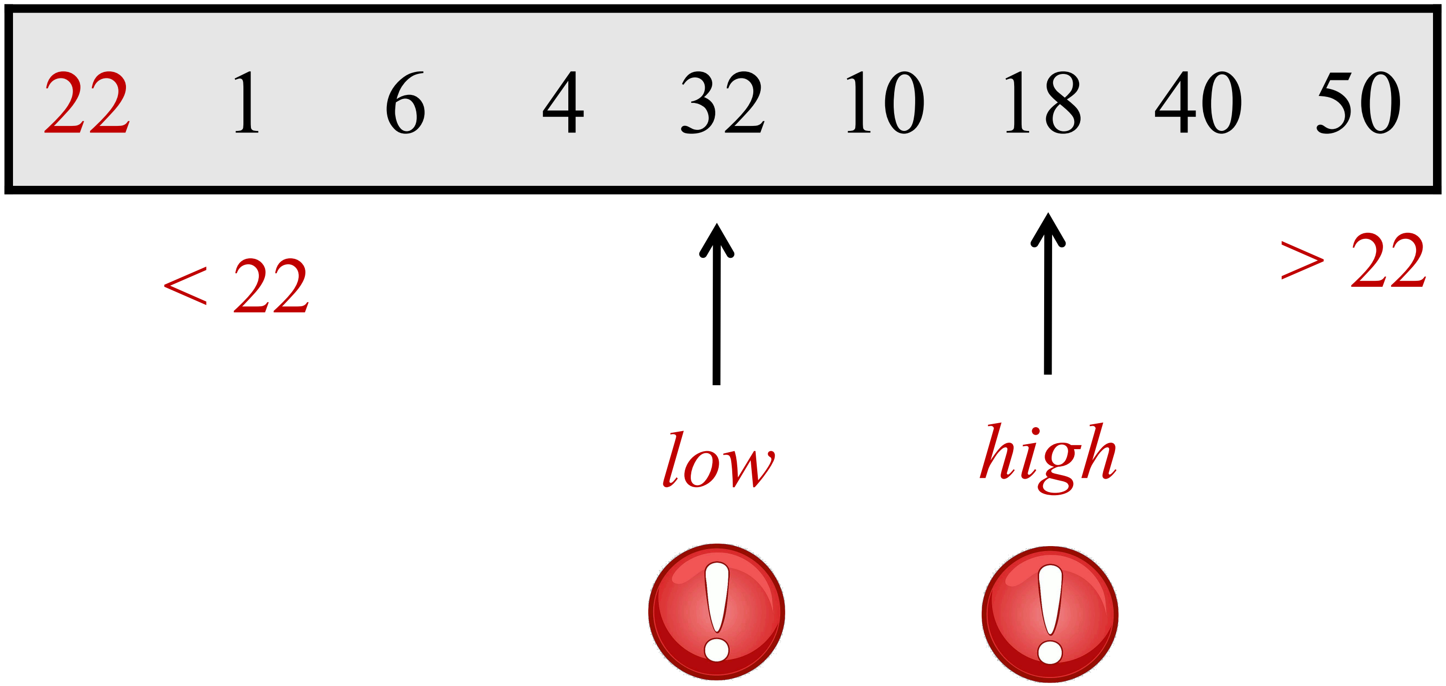
$\uparrow$ *low*

$\uparrow$ *high*

> 22

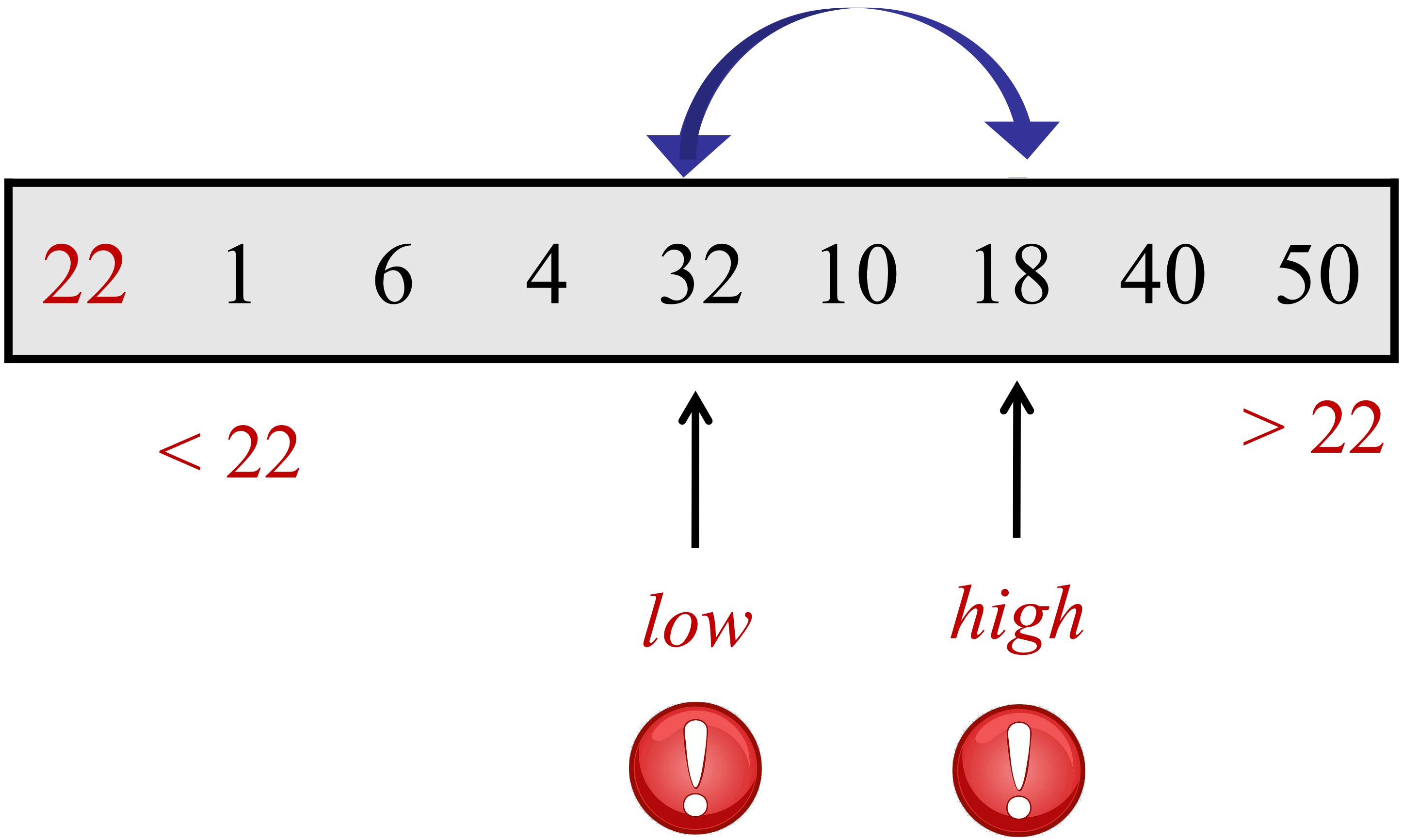
Partitioning an Array

Example: partition around 22



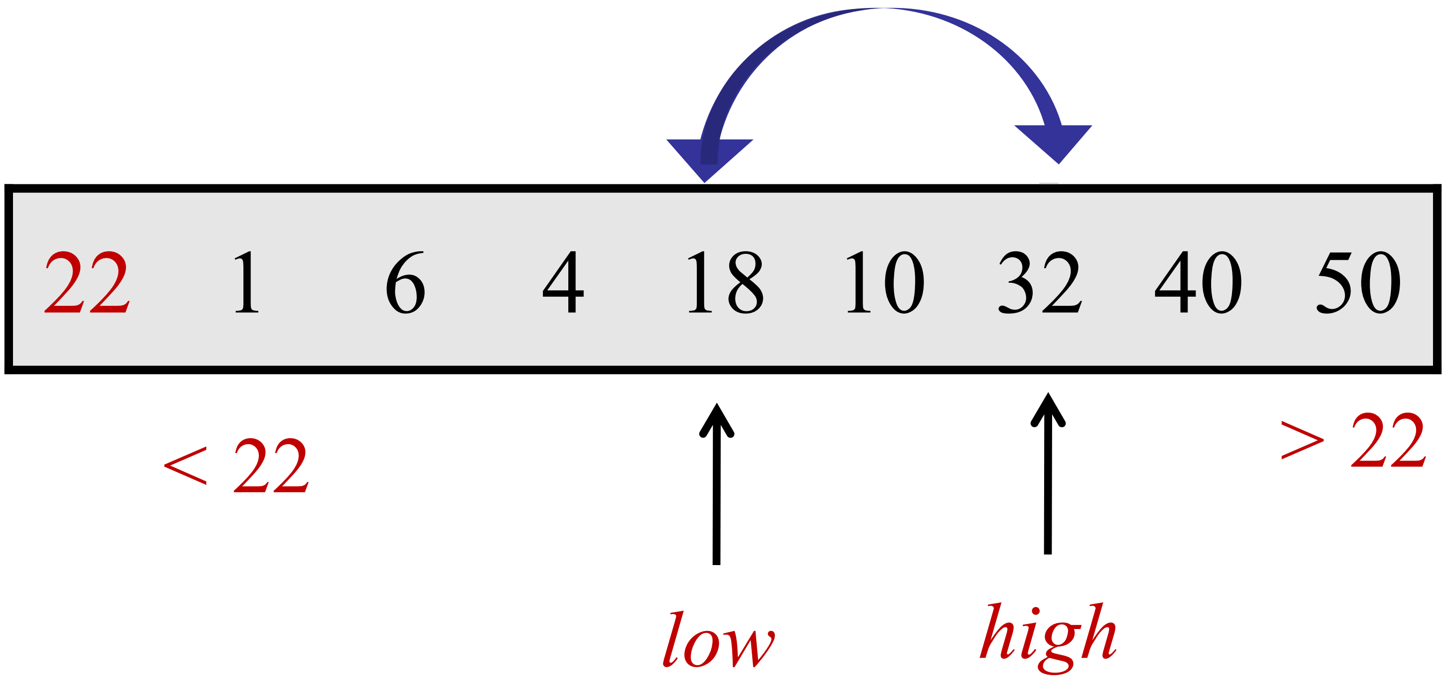
Partitioning an Array

Example: partition around 22



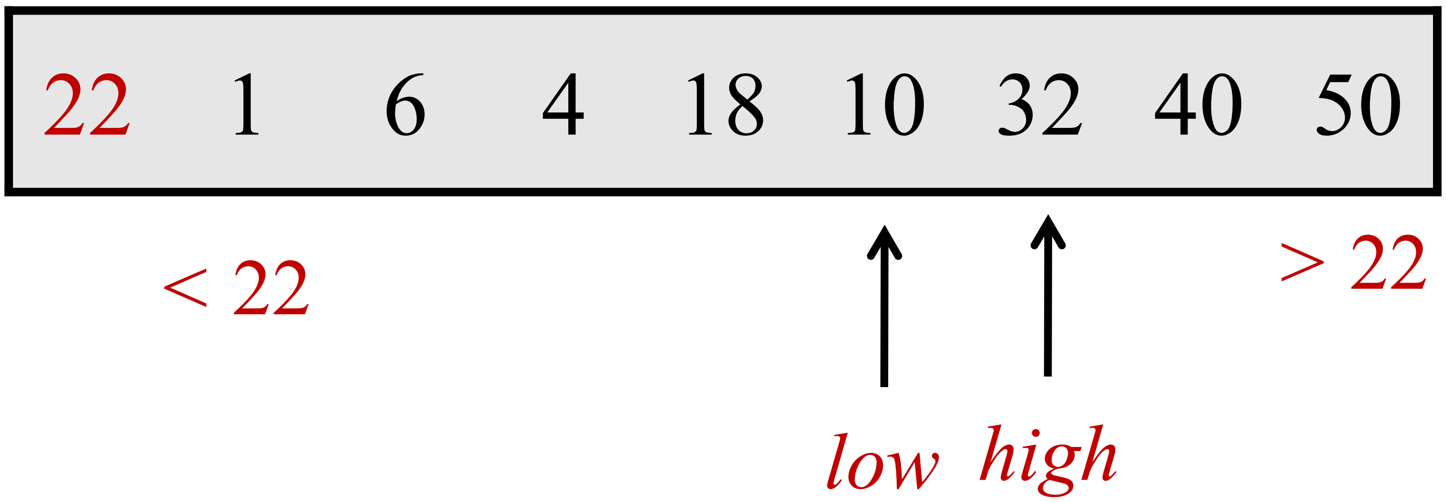
Partitioning an Array

Example: partition around 22



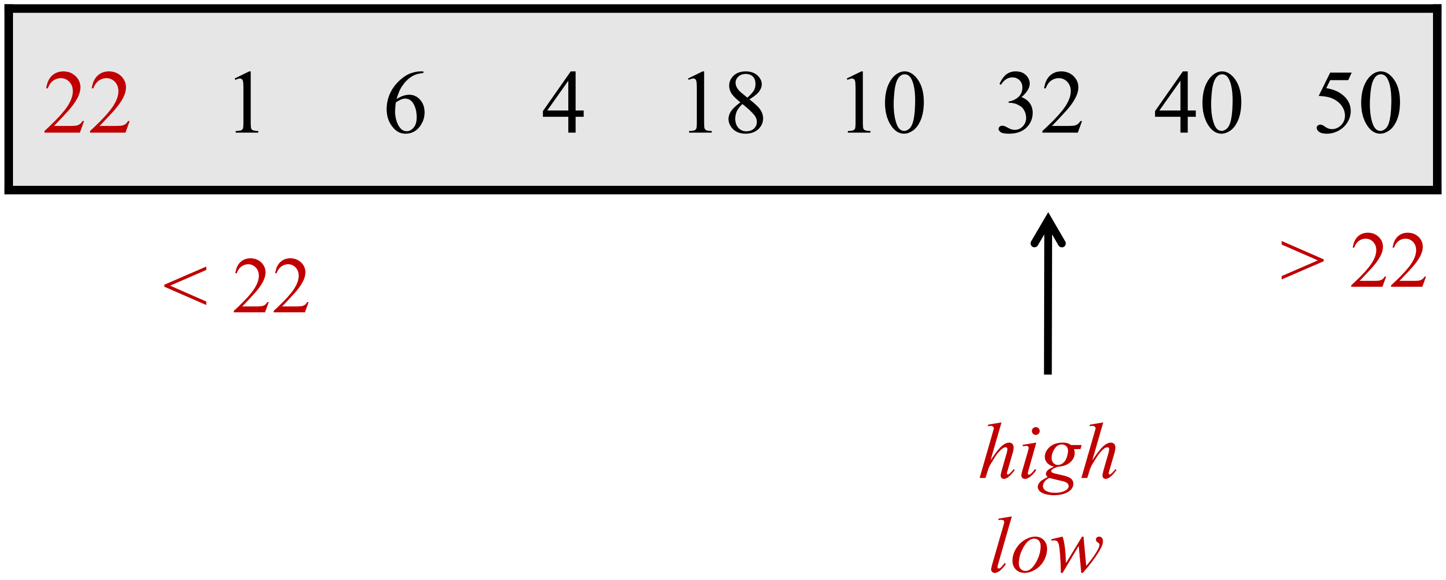
Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22

22	1	6	4	18	10	32	40	50

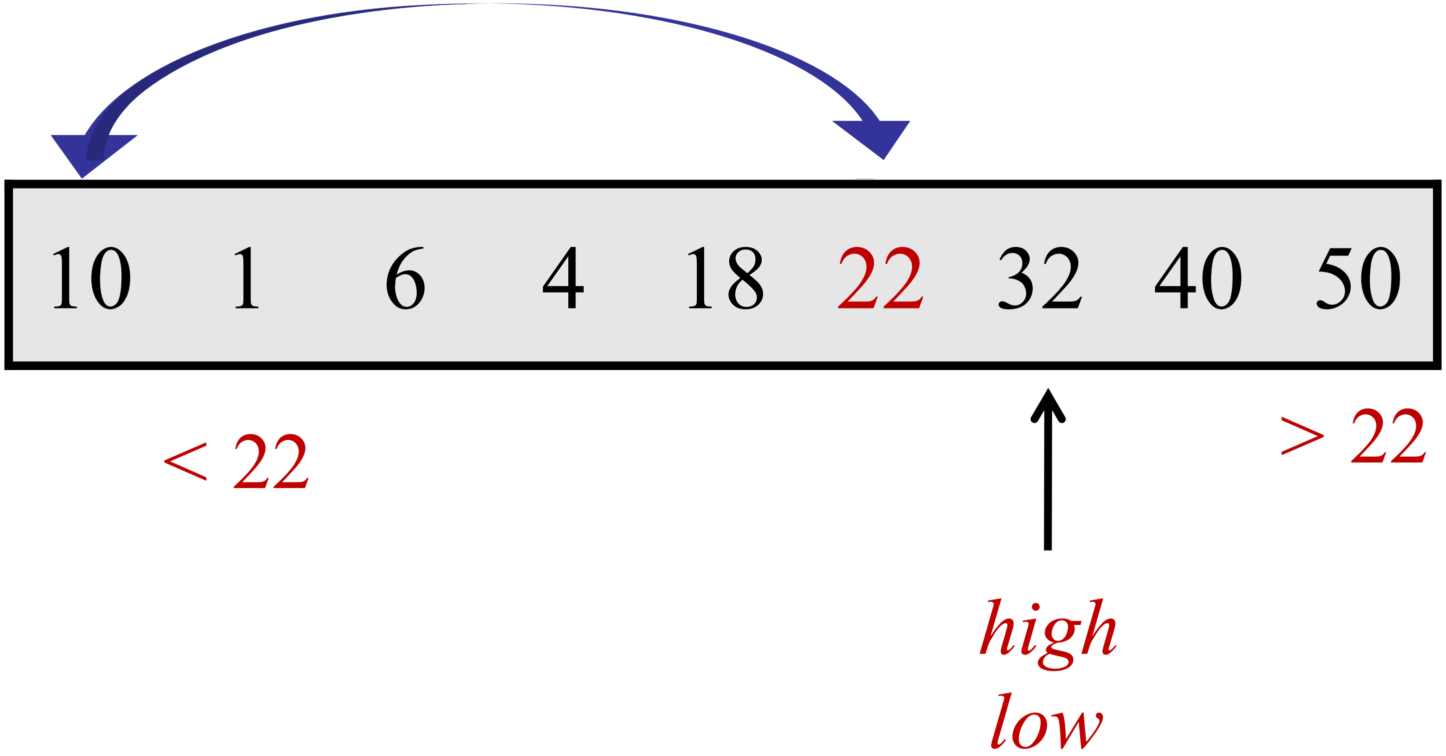
< 22

> 22

high
low

Partitioning an Array

Example: partition around 22



```
partition(A[1..n], n, pIndex)        // Assume no duplicates, n>1
    pivot = A[pIndex];               // pIndex is the index of pivot
    swap(A[1], A[pIndex]);           // store pivot in A[1]
    low = 2;                         // start after pivot in A[1]
    high = n+1;                      // Define: A[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;
```

Pseudocode

vs.

Real Code

QuickSort is notorious for off-by-one errors…

Partition

Invariant: $A[high] > pivot$ at the end of each loop.

Proof:

Initially: true by assumption $A[n+1] = \infty$

Partition

Invariant: $A[high] > pivot$ at the end of each iter:

Proof: During loop:

- When exit loop incrementing low: $A[low] > pivot$

  If ($low > high$), then by **while** condition.

  If ($low = high$), then by inductive assumption.

- When exit loop decrementing high:

  $A[high] < pivot$ OR $low = high$

- If ($high == low$), then $A[high] > pivot$
- Otherwise, swap $A[high]$ and $A[low]$>pivot.

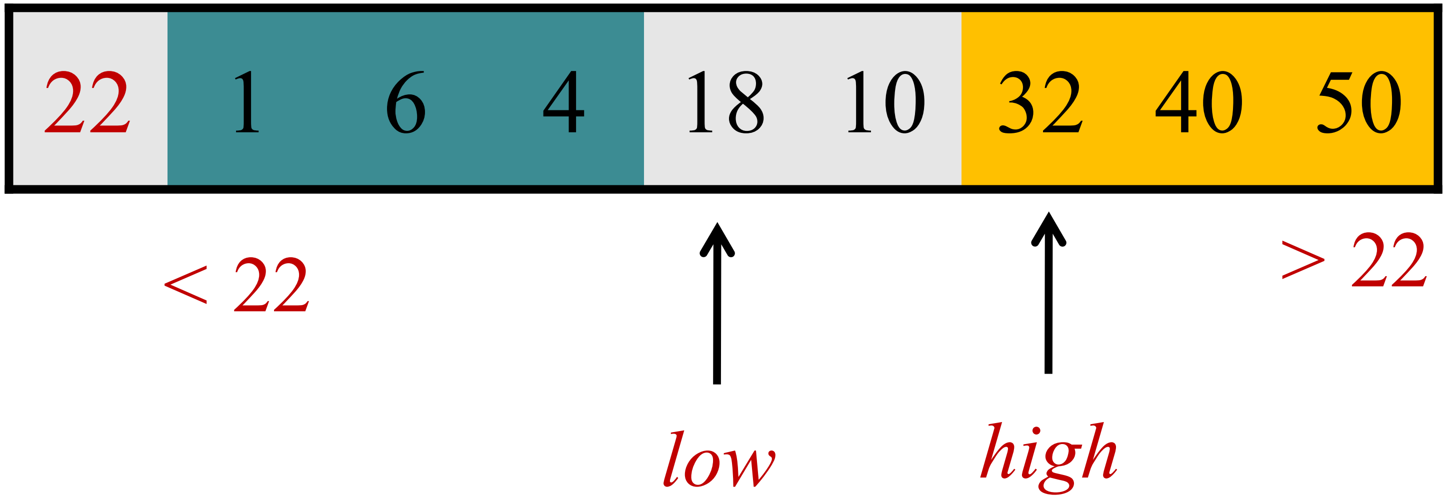
```
partition(A[1..n], n, pIndex)          // Assume no duplicates,  n>1
    pivot = A[pIndex];                 // pIndex is the index of pivot
    swap(A[1], A[pIndex]);             // store pivot in A[1]
    low = 2;                           // start after pivot in A[1]
    high = n+1;                        // Define: A[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;
```

Partition

Invariant: At the end of every loop iteration:

for all $i >= high$, $A[i] > pivot$.

for all $1 < j < low$, $A[j] < pivot$.

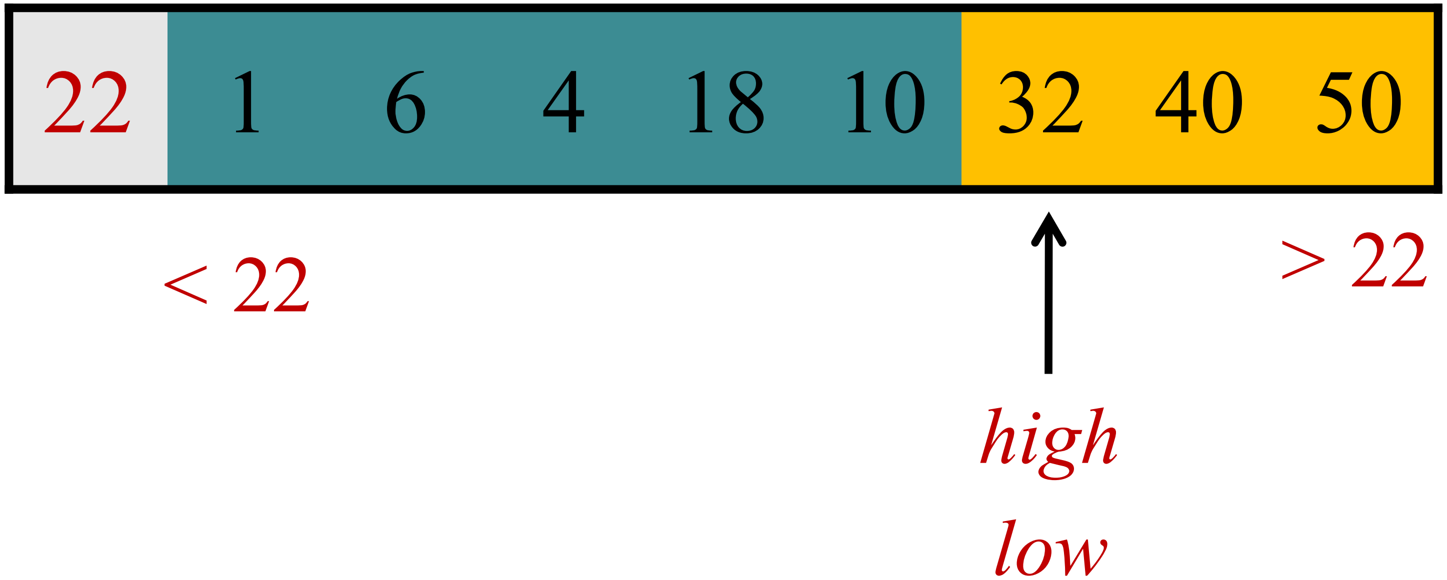


Partition

Invariant: At the end of every loop iteration:

for all $i >= high$, $A[i] > pivot$.

for all $1 < j < low$, $A[j] < pivot$.

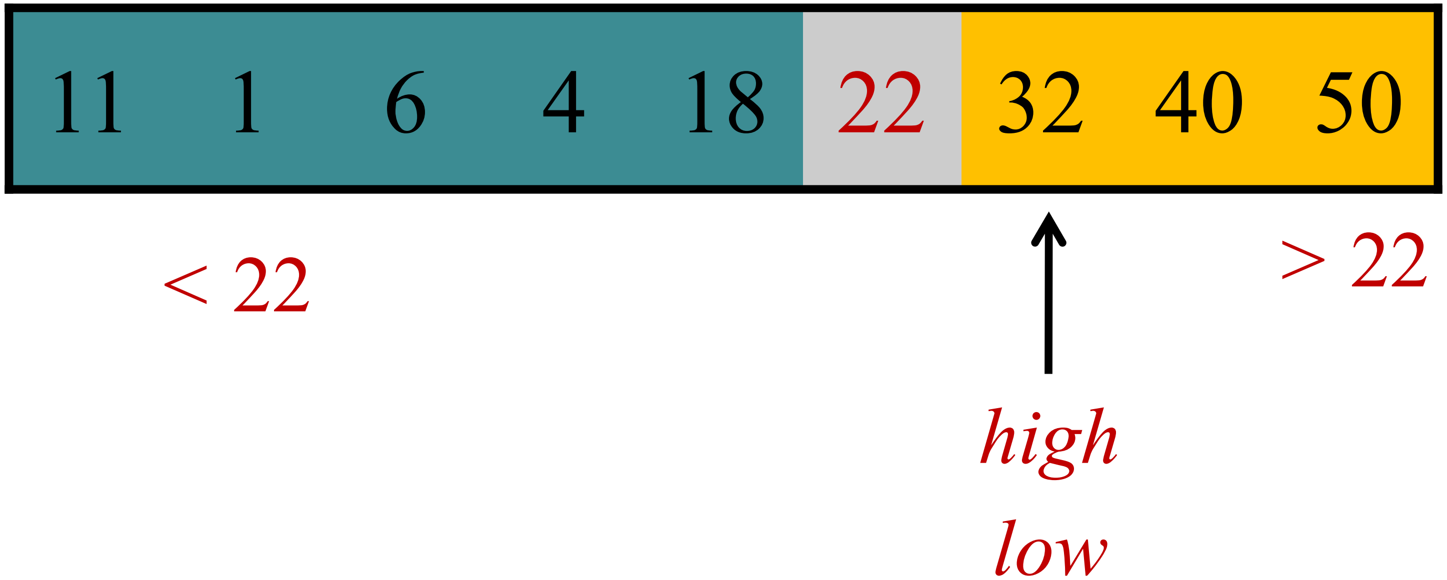


Partition

Claim: At the end of every loop iteration:

for all $i >= high$, $A[i] > pivot$.

for all $1 < j < low$, $A[j] < pivot$.



Claim: Array A is partitioned around the pivot

```
partition(A[1..n], n, pIndex)        // Assume no duplicates,  n>1
    pivot = A[pIndex];               // pIndex is the index of pivot
    swap(A[1], A[pIndex]);           // store pivot in A[1]
    low = 2;                         // start after pivot in A[1]
    high = n+1;                      // Define: A[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;
```

```
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

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($A[p+1..n]$, $n-p$)

$< x$	x	$> x$

Today: Sorting, Part II

QuickSort

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

QuickSort

What happens if there are duplicates?

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($A[p+1..\mathrm{n}]$, $n-p$)

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

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

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

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

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

```
partition(A[1..n], n, pIndex)        // Assume no duplicates, n>1
    pivot = A[pIndex];                // pIndex is the index of pivot
    swap(A[1], A[pIndex]);            // store pivot in A[1]
    low = 2;                          // start after pivot in A[1]
    high = n+1;                       // Define: A[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;
```

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;

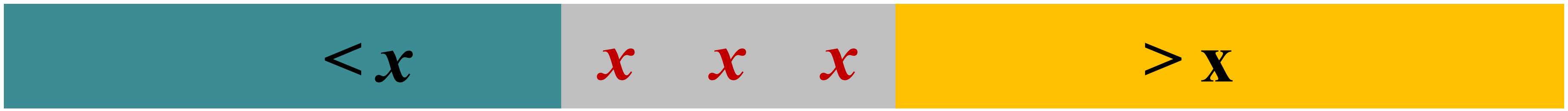
else

Choose pivot index *pIndex*.

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

x = QuickSort(A[1..p–1], p–1)

y = QuickSort(A[p+1..n], n–p)



Pivot

Duplicates

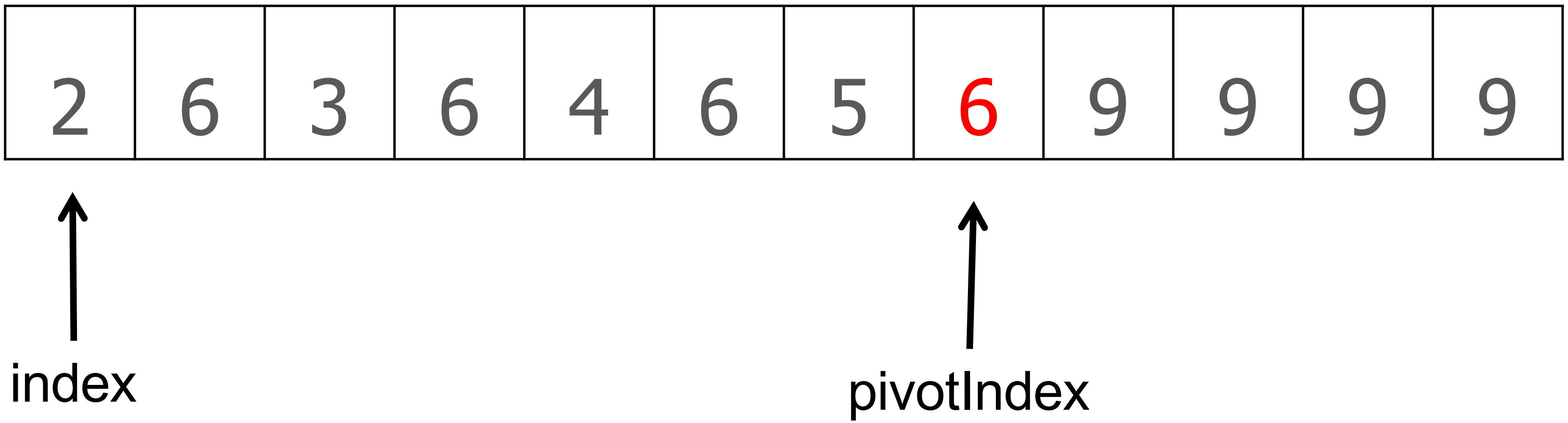
3-Way Partitioning

– Option 1: two pass partitioning

1. Regular partition.
2. Pack duplicates.

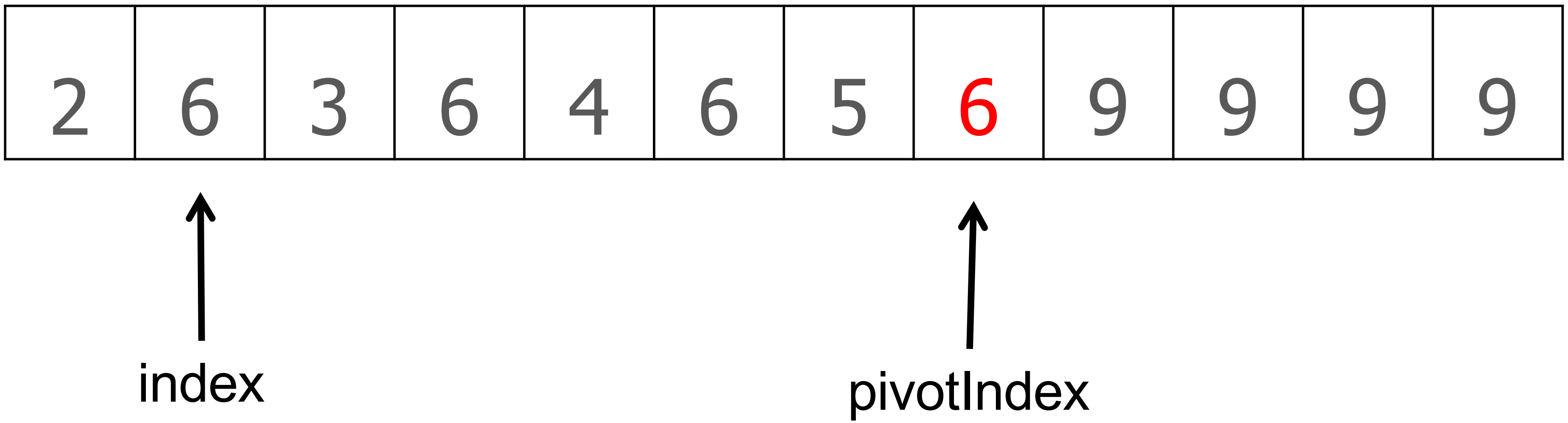
Pack Duplicates

Example:



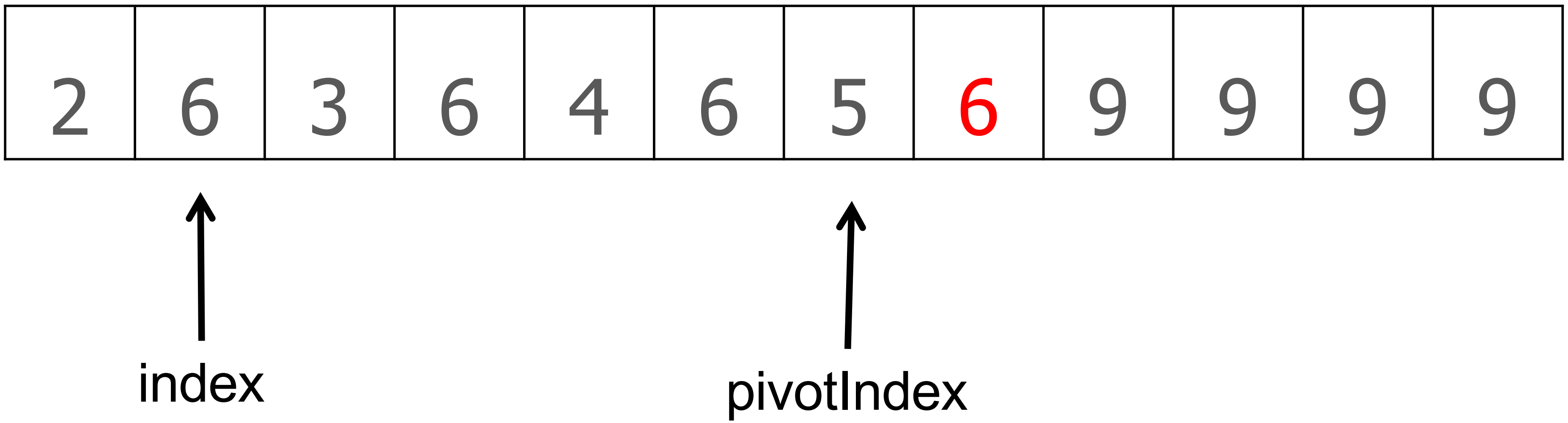
Pack Duplicates

Example:



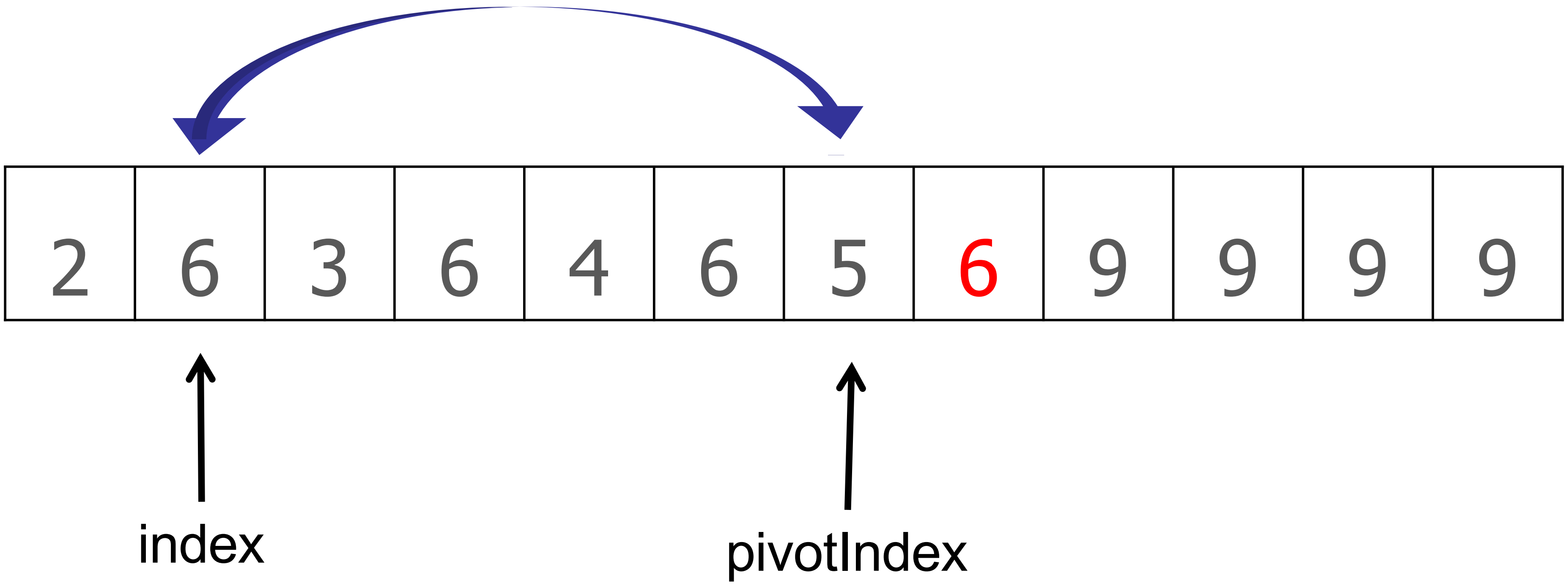
Pack Duplicates

Example:



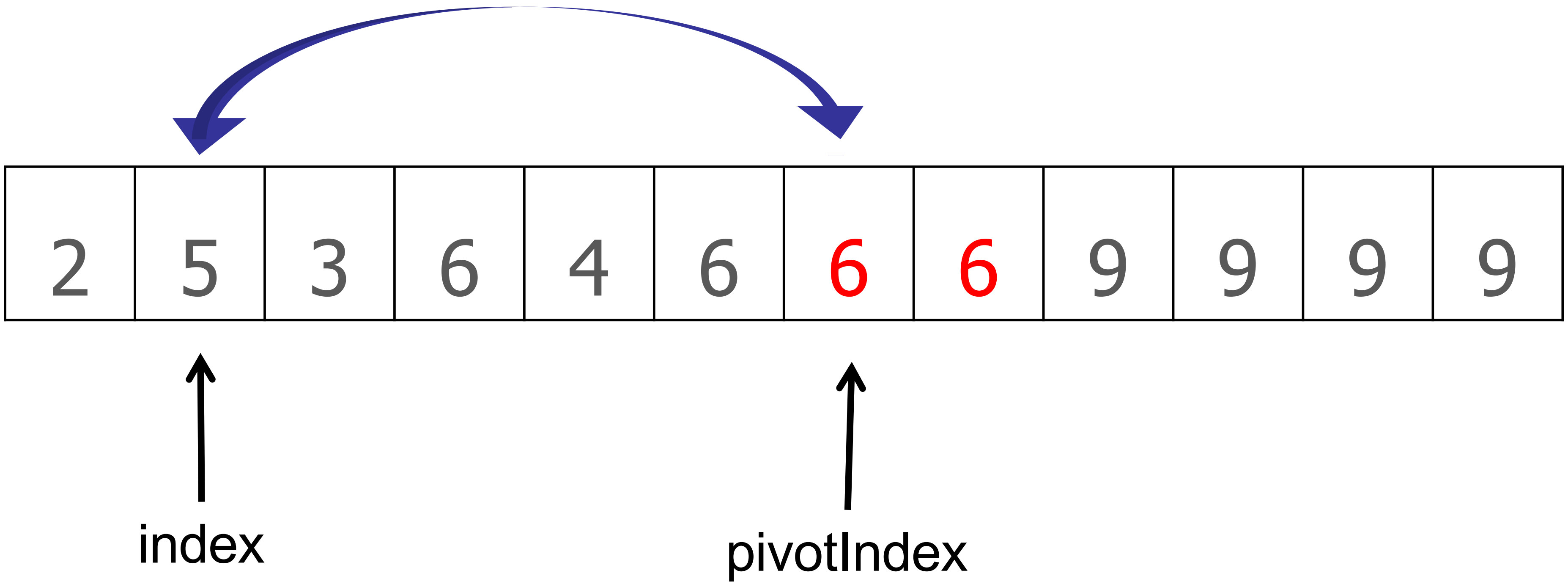
Pack Duplicates

Example:



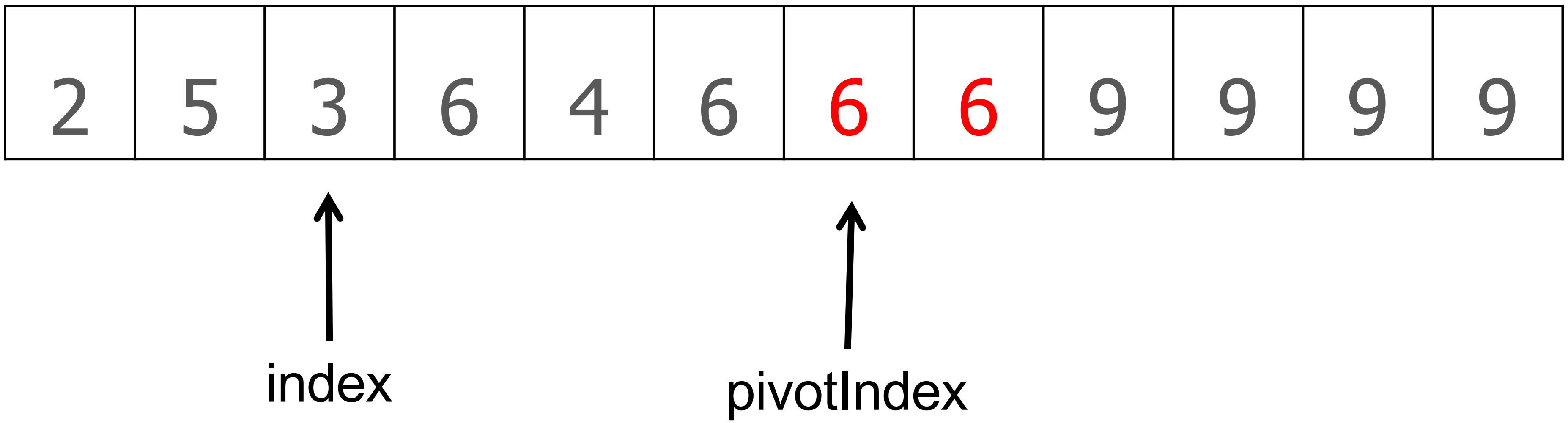
Pack Duplicates

Example:



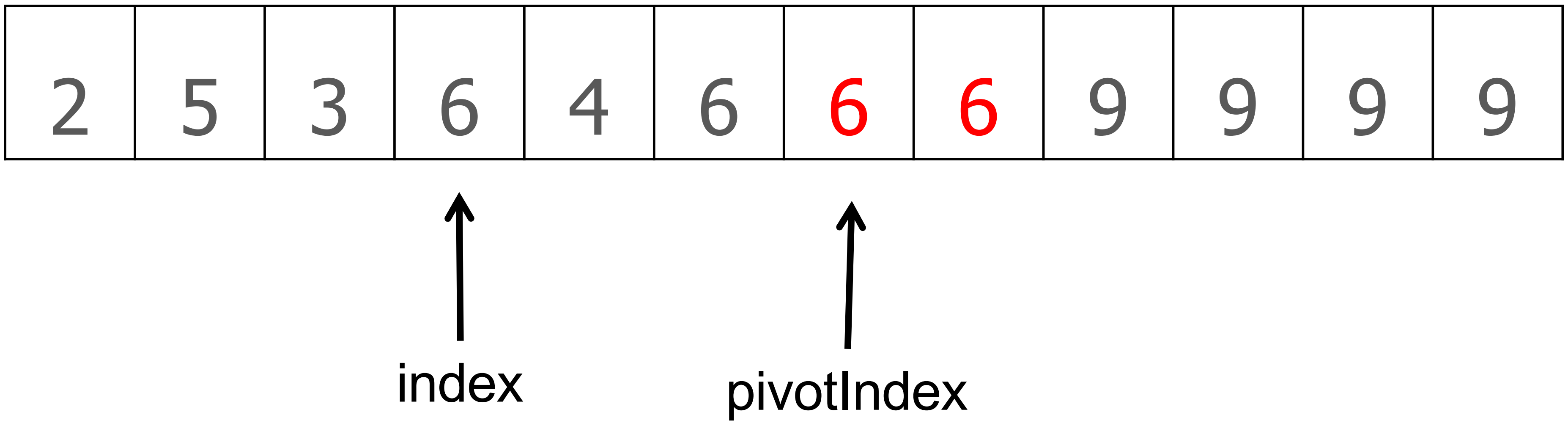
Pack Duplicates

Example:



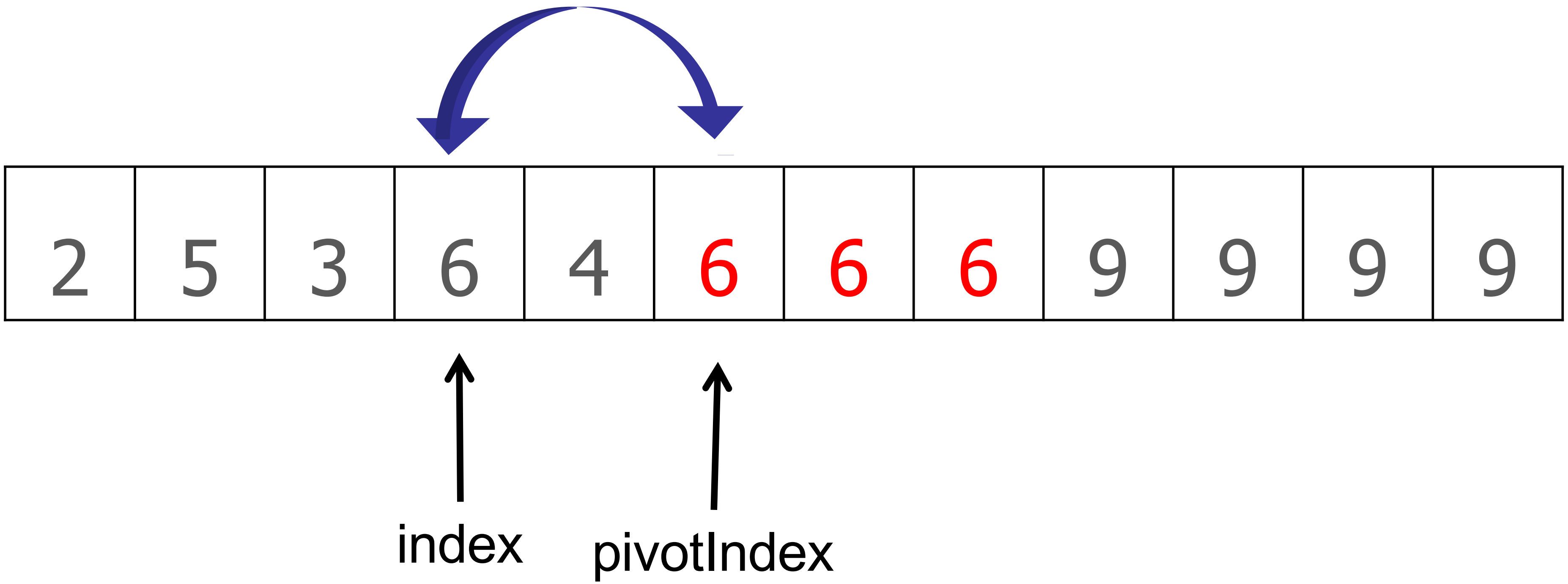
Pack Duplicates

Example:



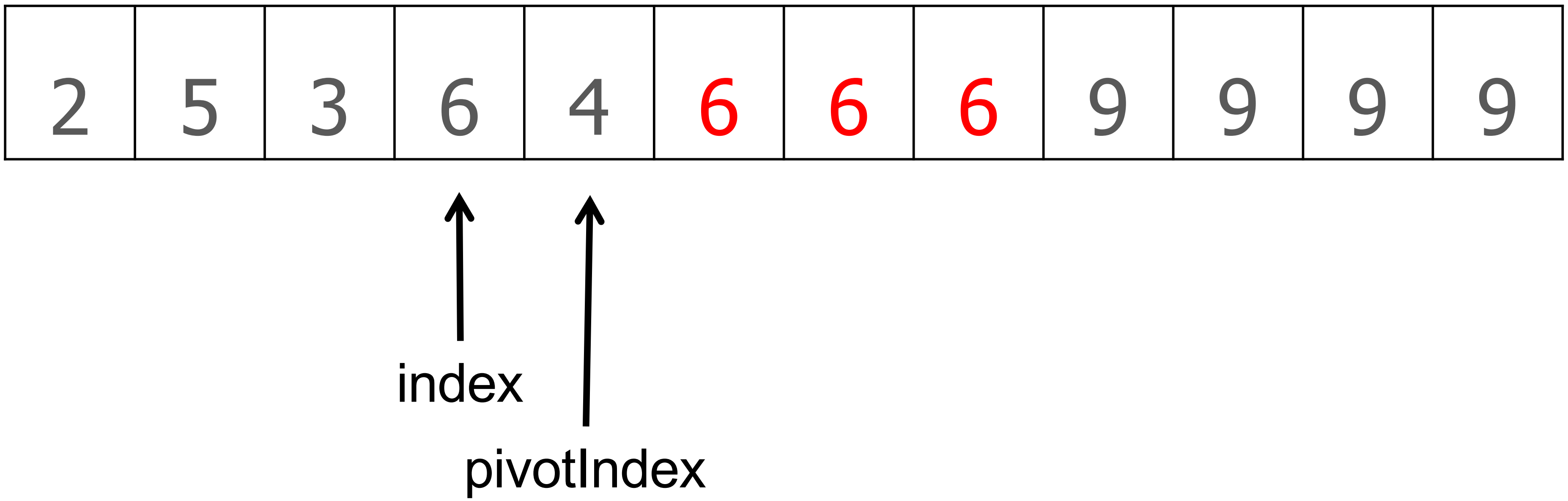
Pack Duplicates

Example:



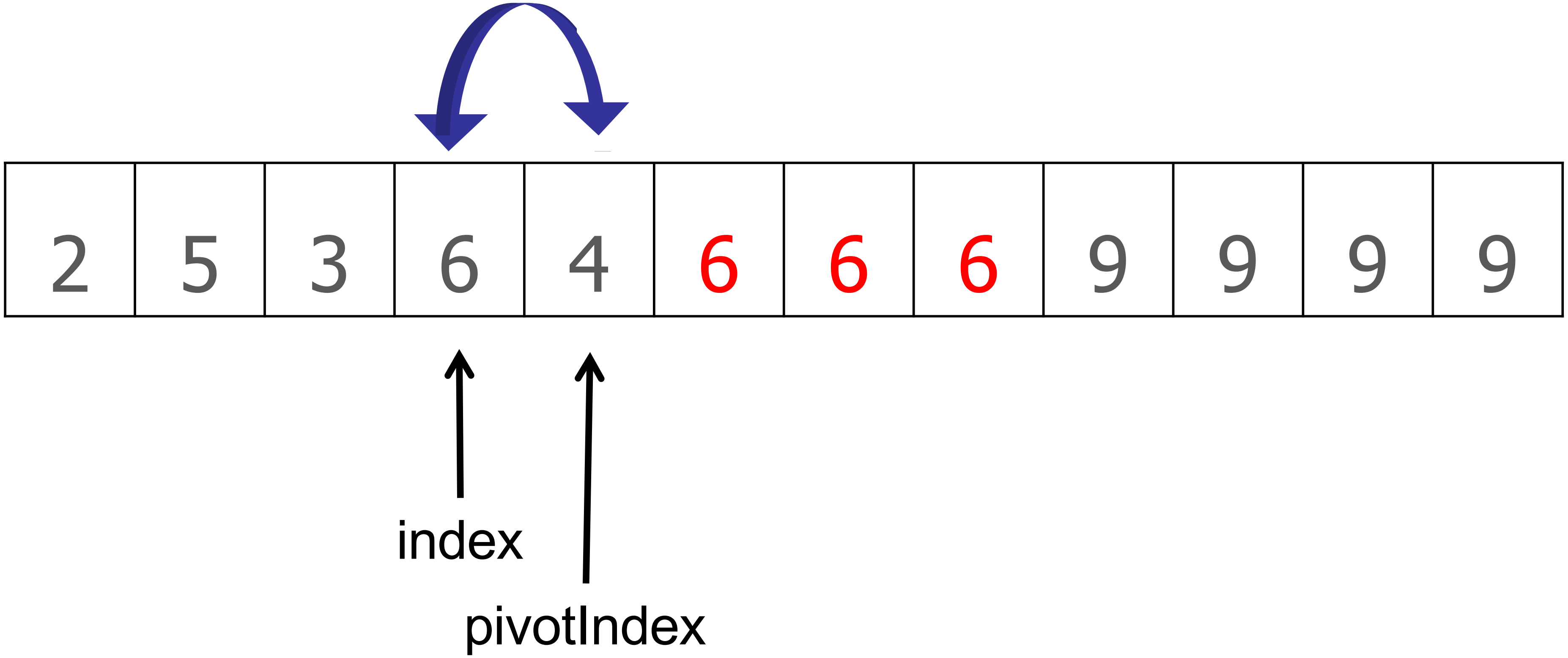
Pack Duplicates

Example:



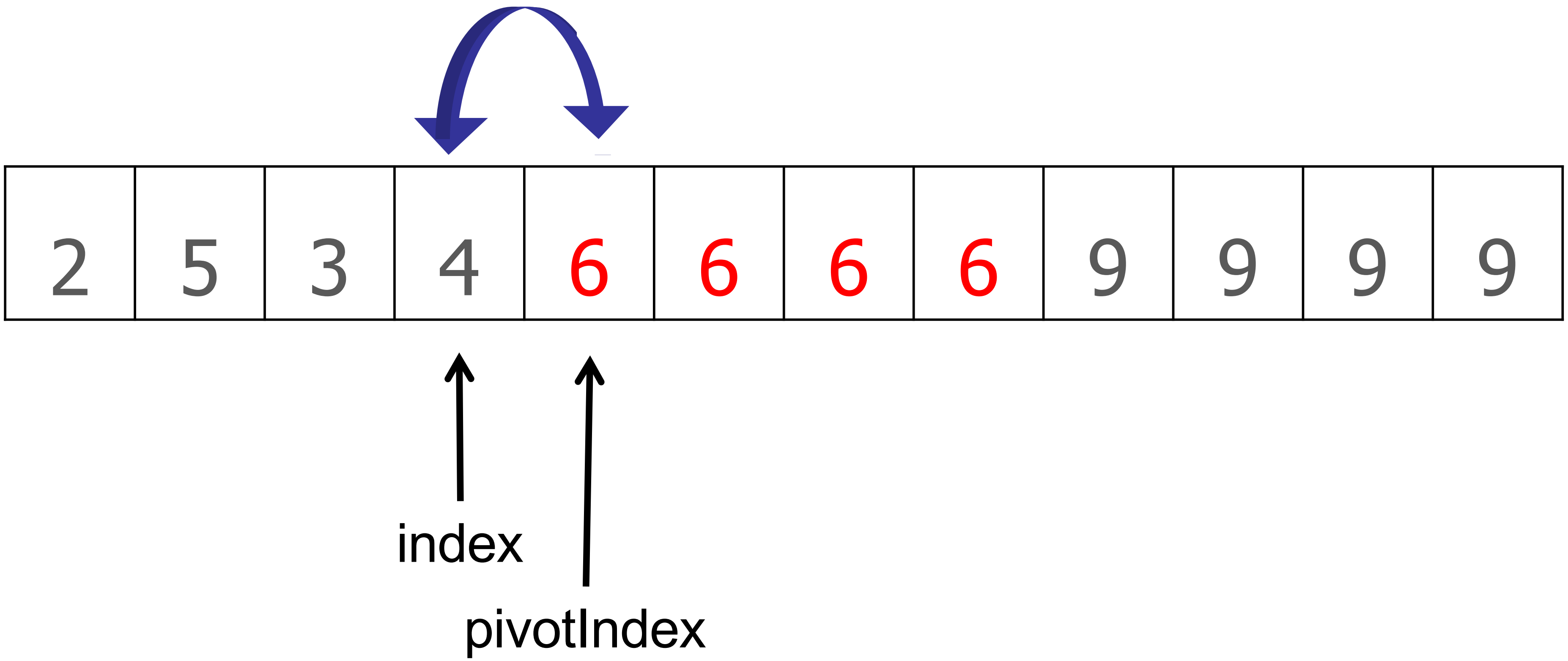
Pack Duplicates

Example:



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;

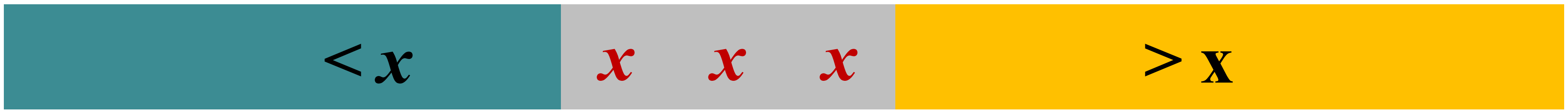
    else

        Choose pivot index *pIndex*.

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

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

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



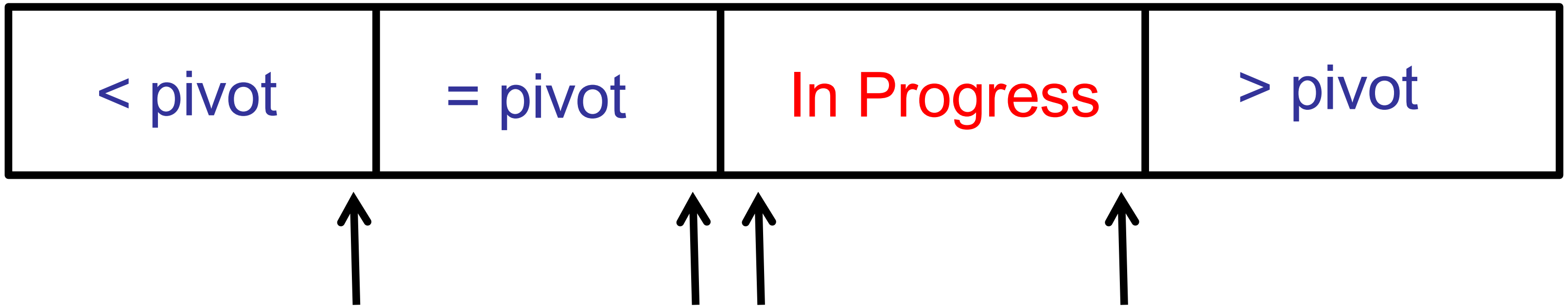
Duplicates

3-Way Partitioning

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

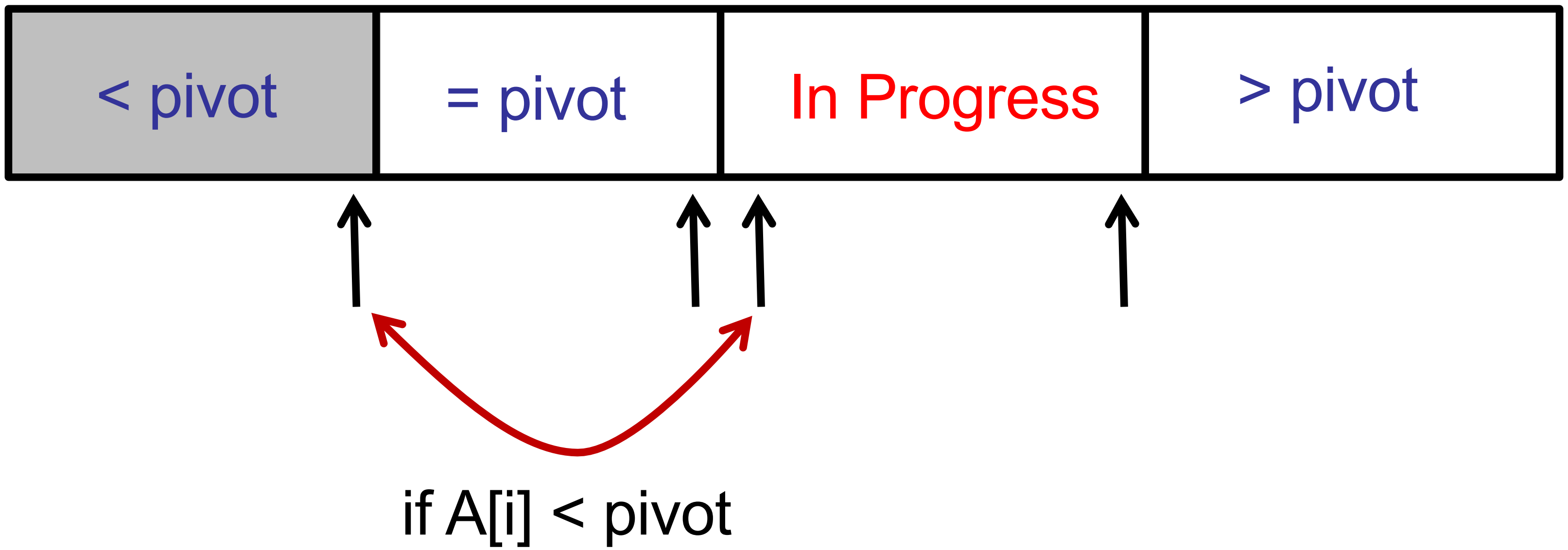
- Option 2: one pass partitioning
  - More complicated.
  - Maintain four regions of the array

3-Way Partitioning

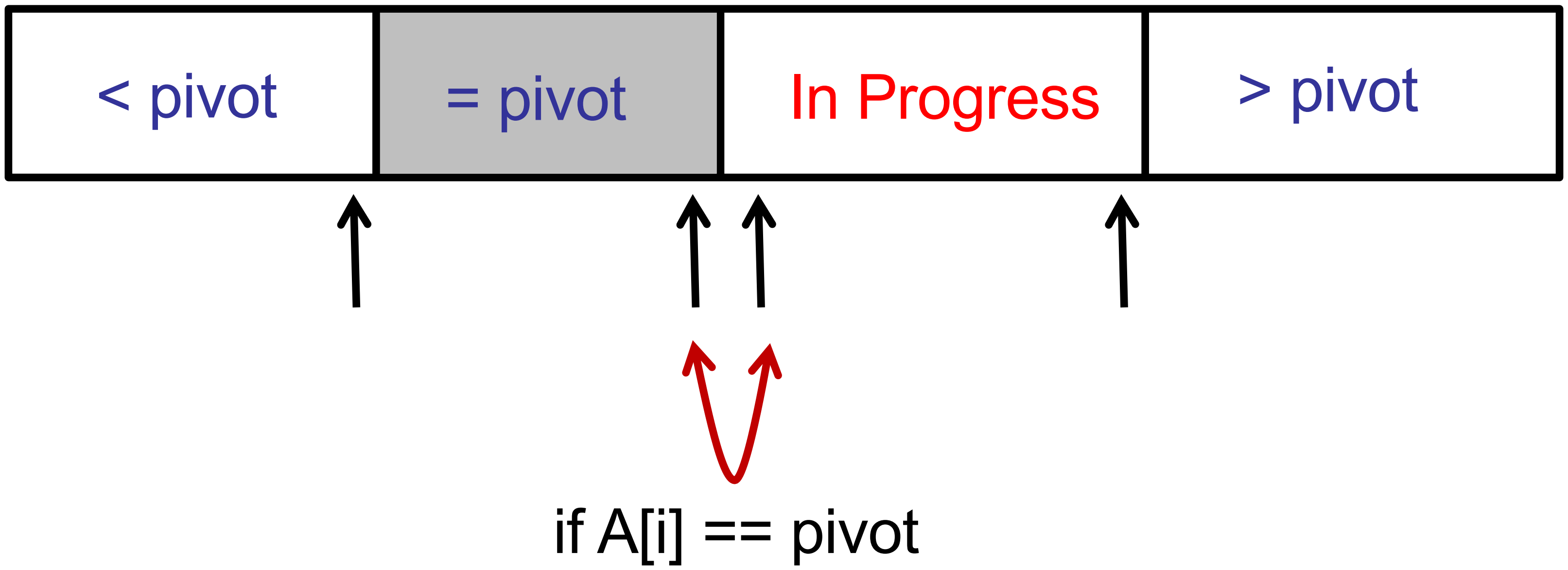


(Note: think about array in terms of invariants!)

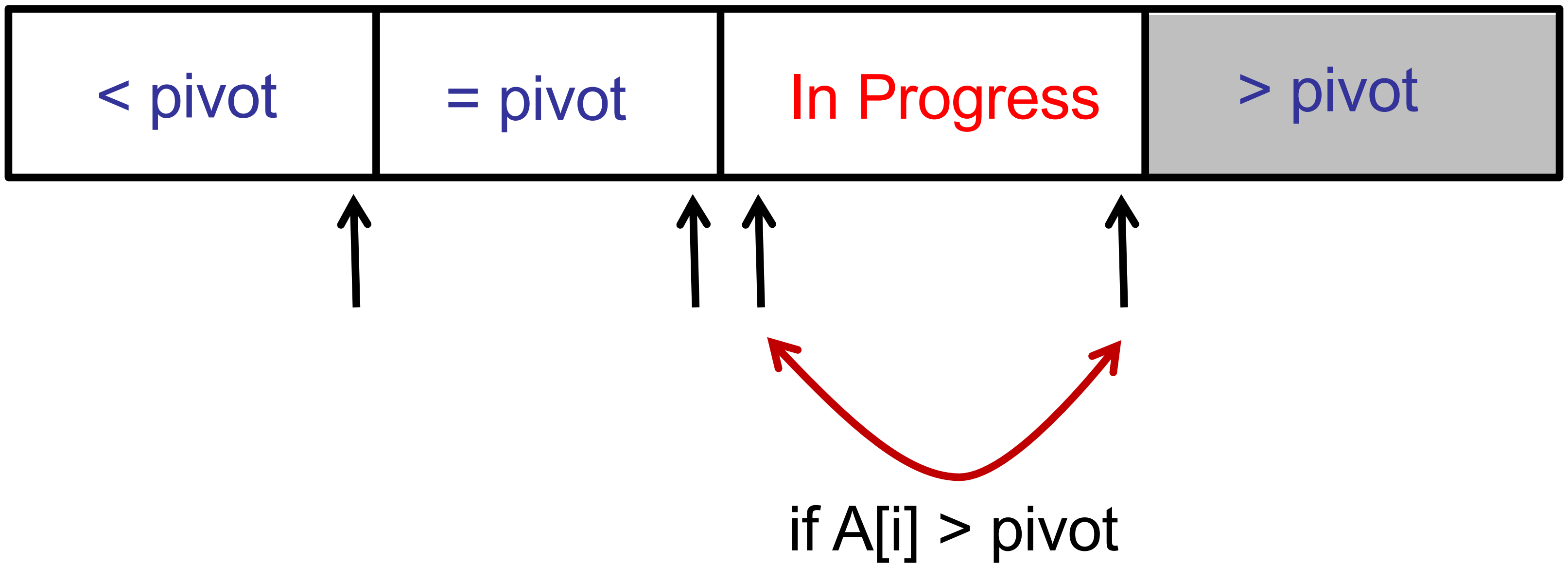
3-Way Partitioning



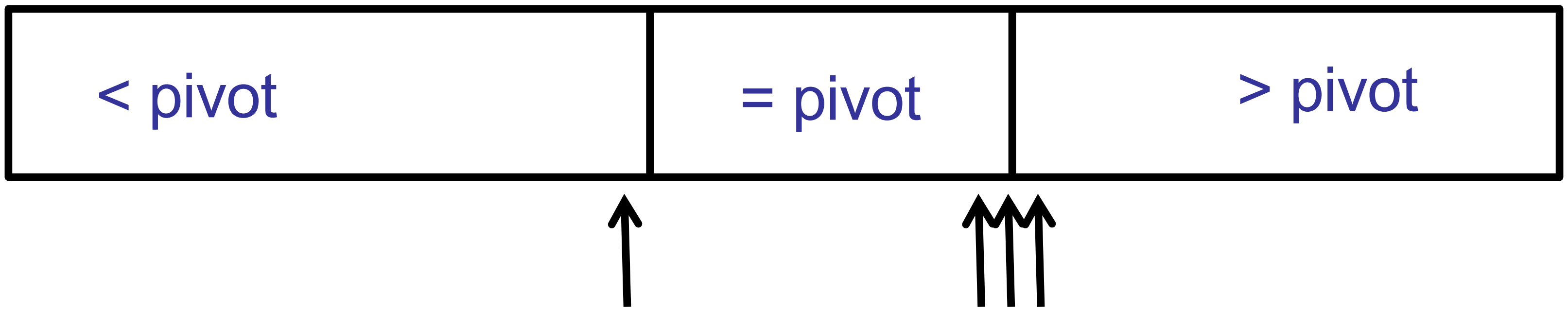
3-Way Partitioning



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, $pIndex$)

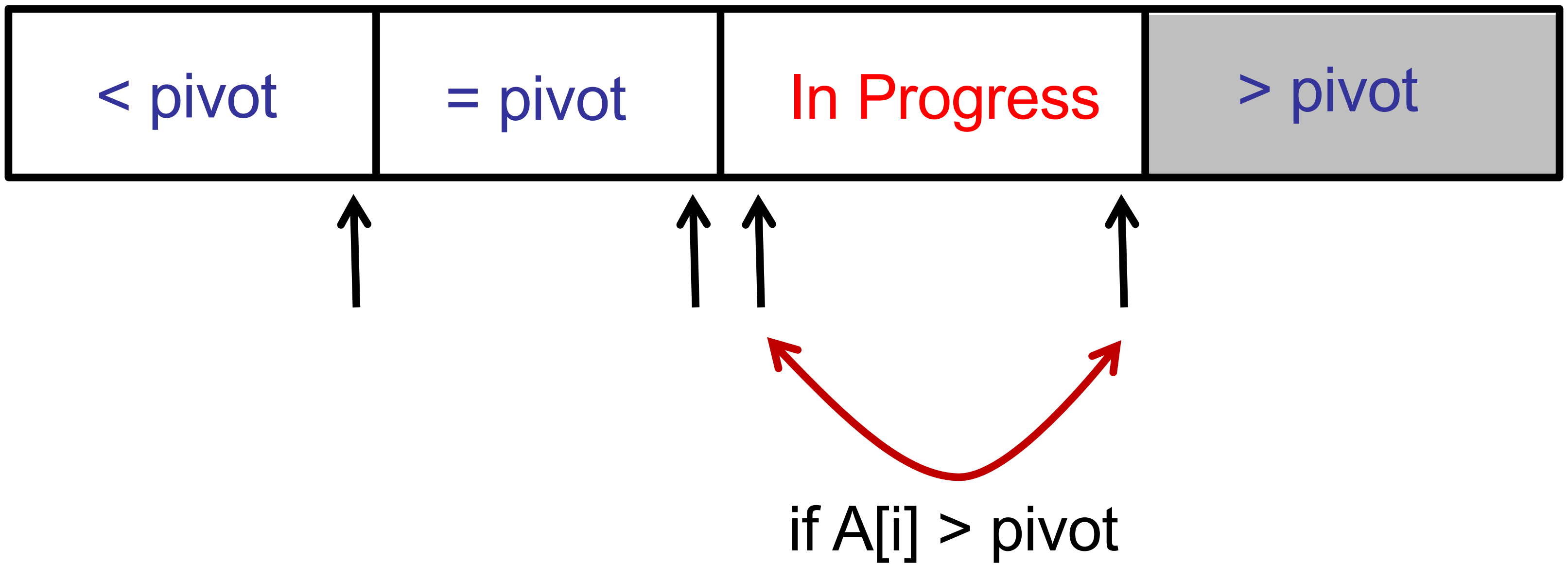
x = QuickSort(A[1..p–1], p–1)

y = QuickSort(A[p+1..n], n–p)

$< x$	x	x	x	$> x$

Is QuickSort stable?

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])

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

$\Downarrow$

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

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

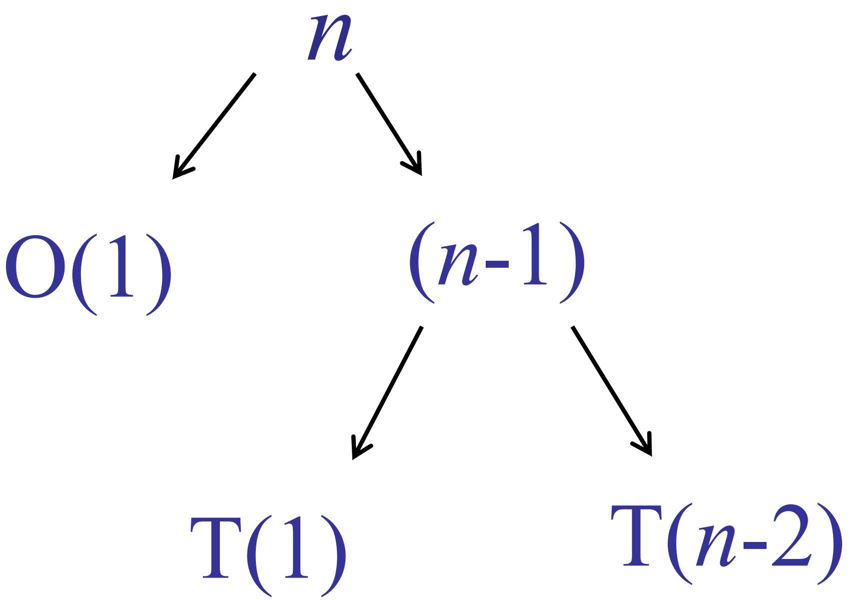
Cost of QuickSort on n – 1 elements

Cost of QuickSort on n elements

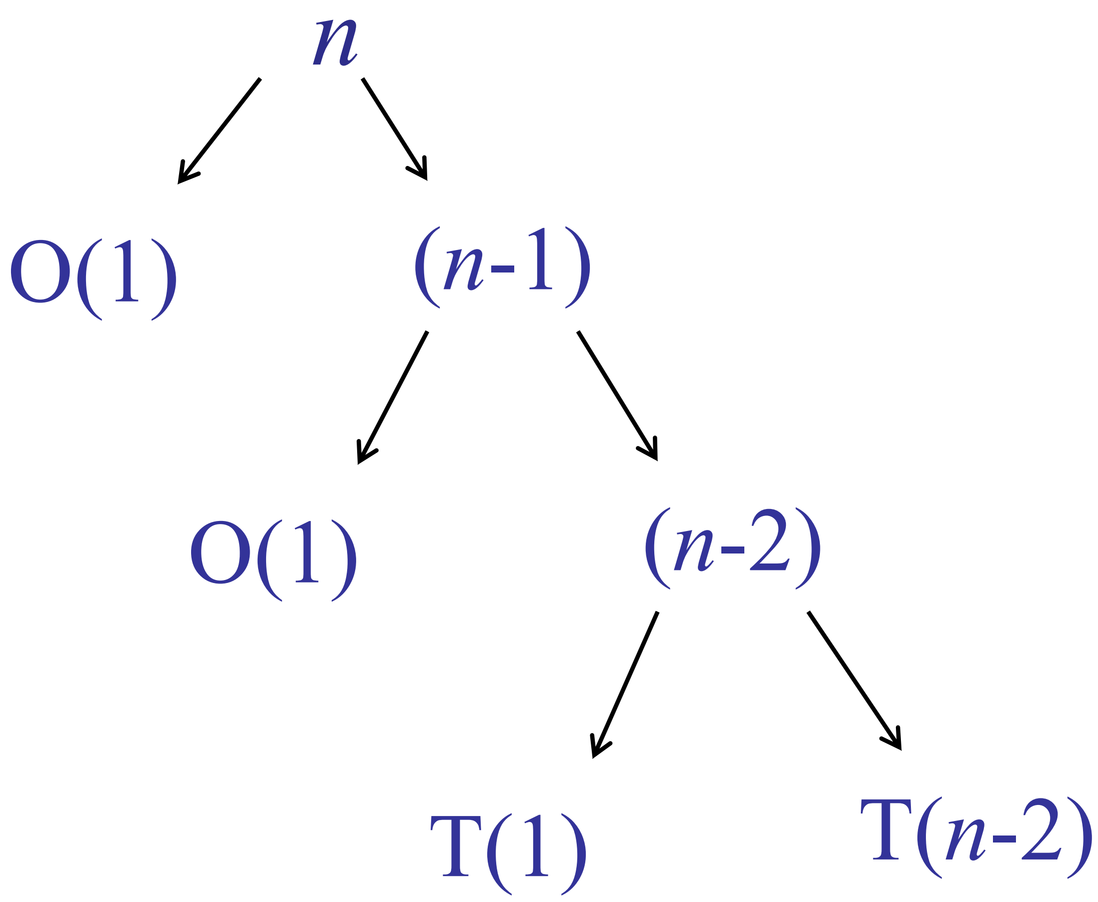
Deterministic QuickSort

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

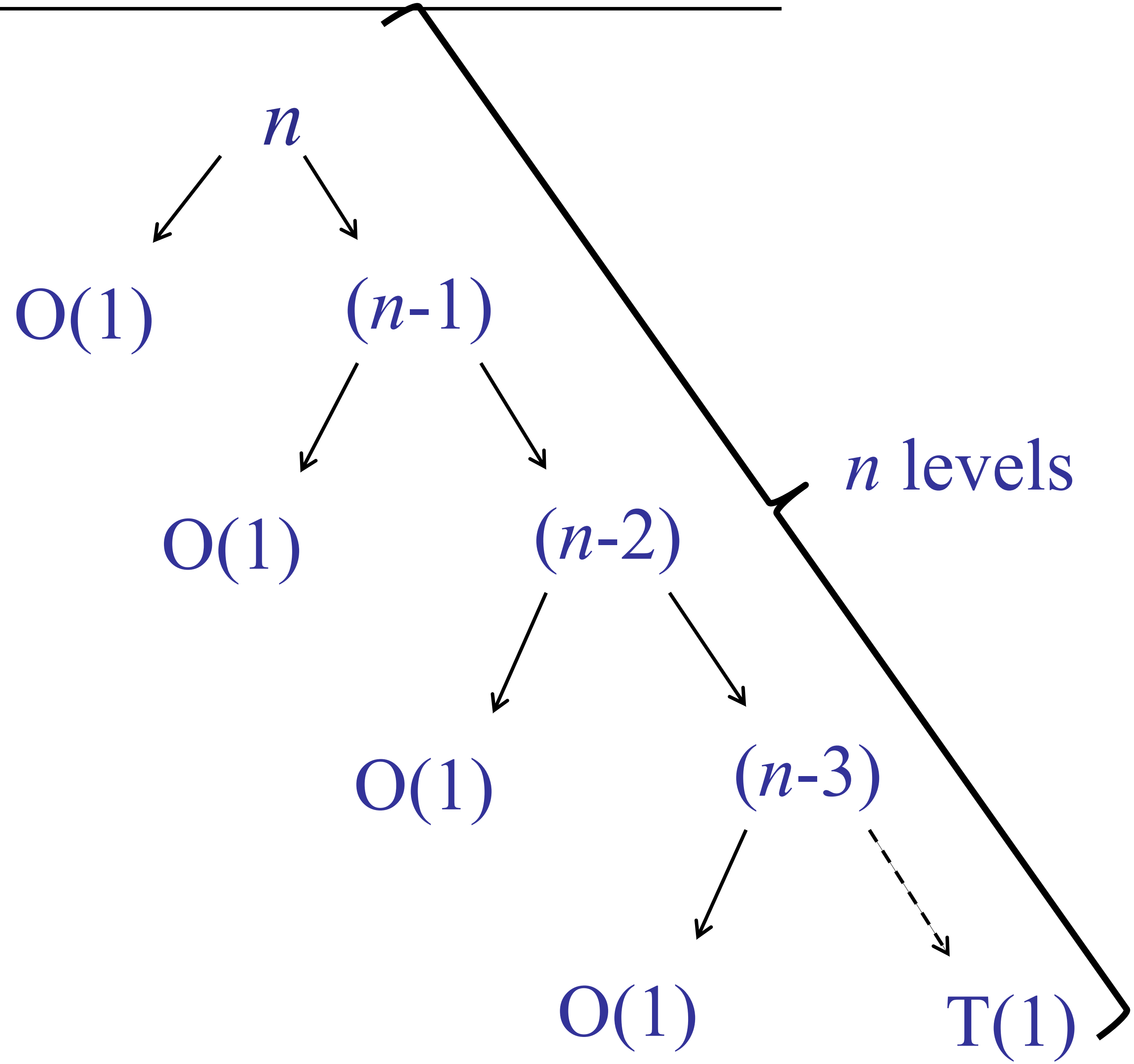
Deterministic QuickSort



Deterministic QuickSort

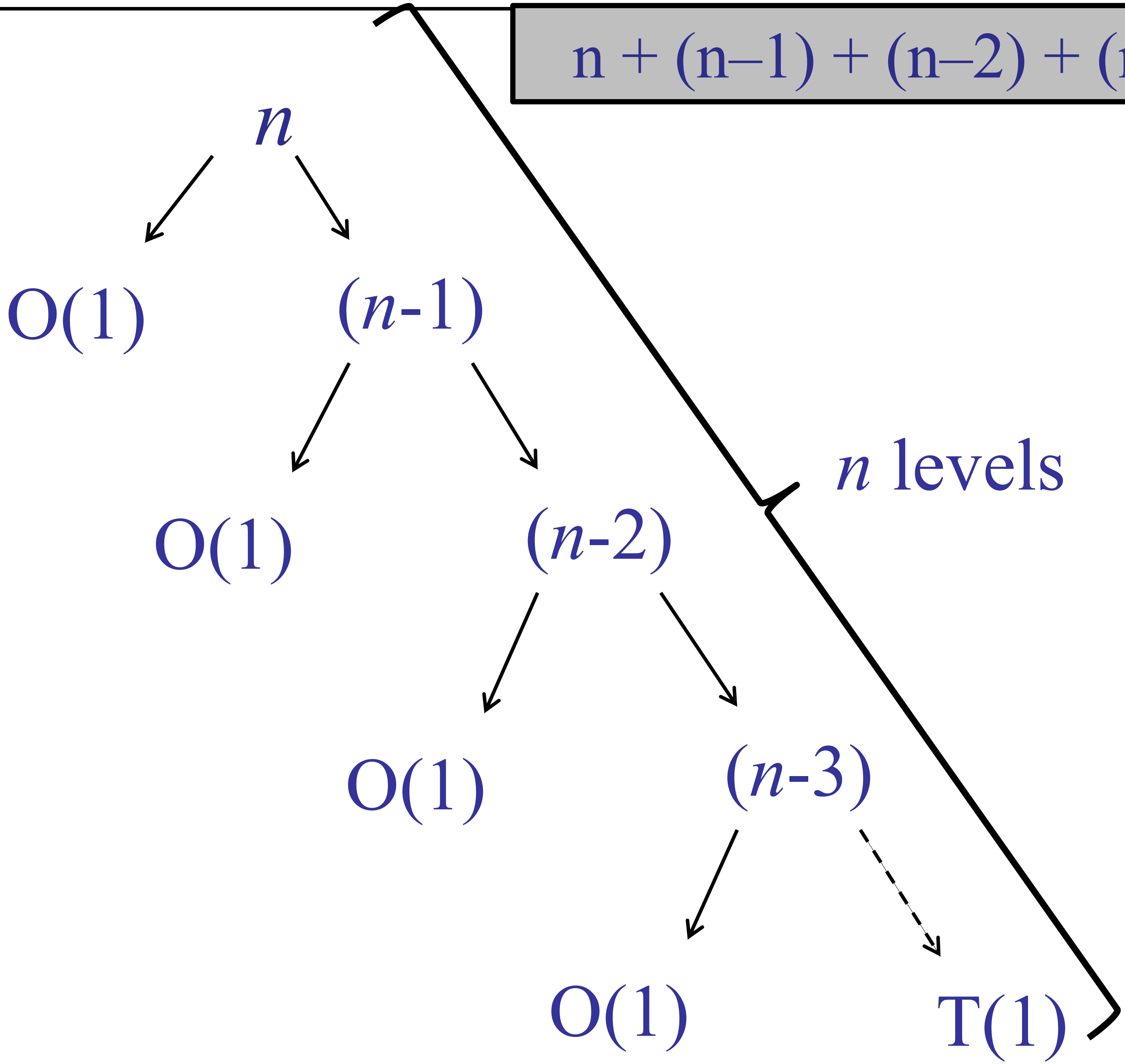


Deterministic QuickSort



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)

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

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:

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

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:

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

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

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

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

What is wrong?

QuickSort

Tempting solution:

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

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

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

$$\cancel{< O(n \log n)}$$

$$= O(n^{6.58})$$

Too loose an estimate.

QuickSort Pivot Choice

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

– $L = \{A[i] : A[i] < pivot\}$

– $H = \{A[i] : A[i] > pivot\}$

L H

What if the *pivot* is chosen so that:

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

QuickSort Analysis

$k = n/10$

$$\begin{tikzcd}[column sep=small] & n \arrow[dl] \arrow[dr] & \\ \mathrm{T}(n/10) & & \mathrm{T}(9n/10) \end{tikzcd}$$

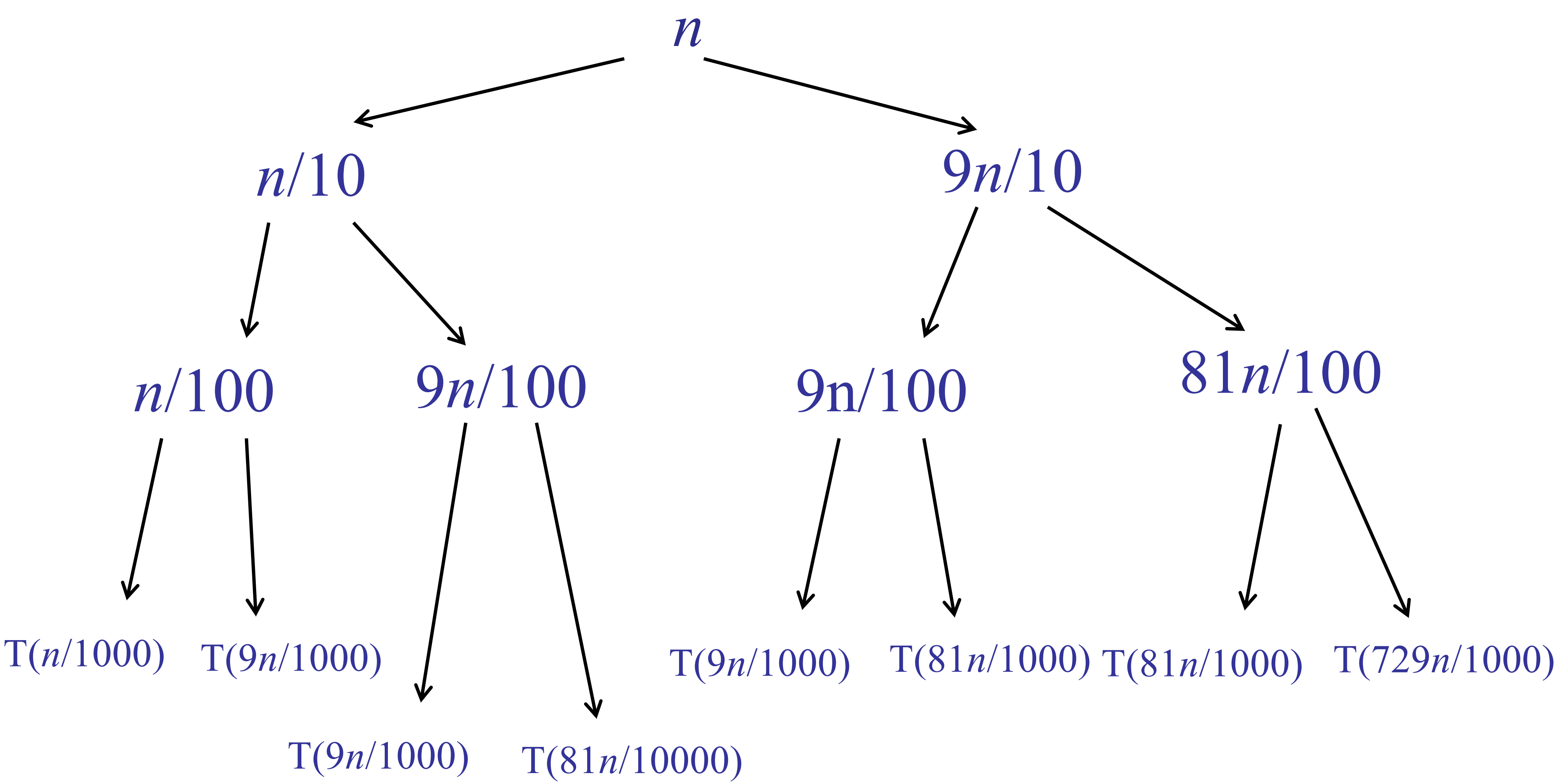
QuickSort Analysis

$k = n/10$

$$\begin{tikzcd}[column sep=small]
 & & & n \arrow[dll] \arrow[drr] & & & \\
 & n/10 \arrow[dl] \arrow[dr] & & & & 9n/10 \arrow[dl] \arrow[dr] & \\
T(n/100) & & T(9n/100) & & T(9n/100) & & T(81n/100)
\end{tikzcd}$$

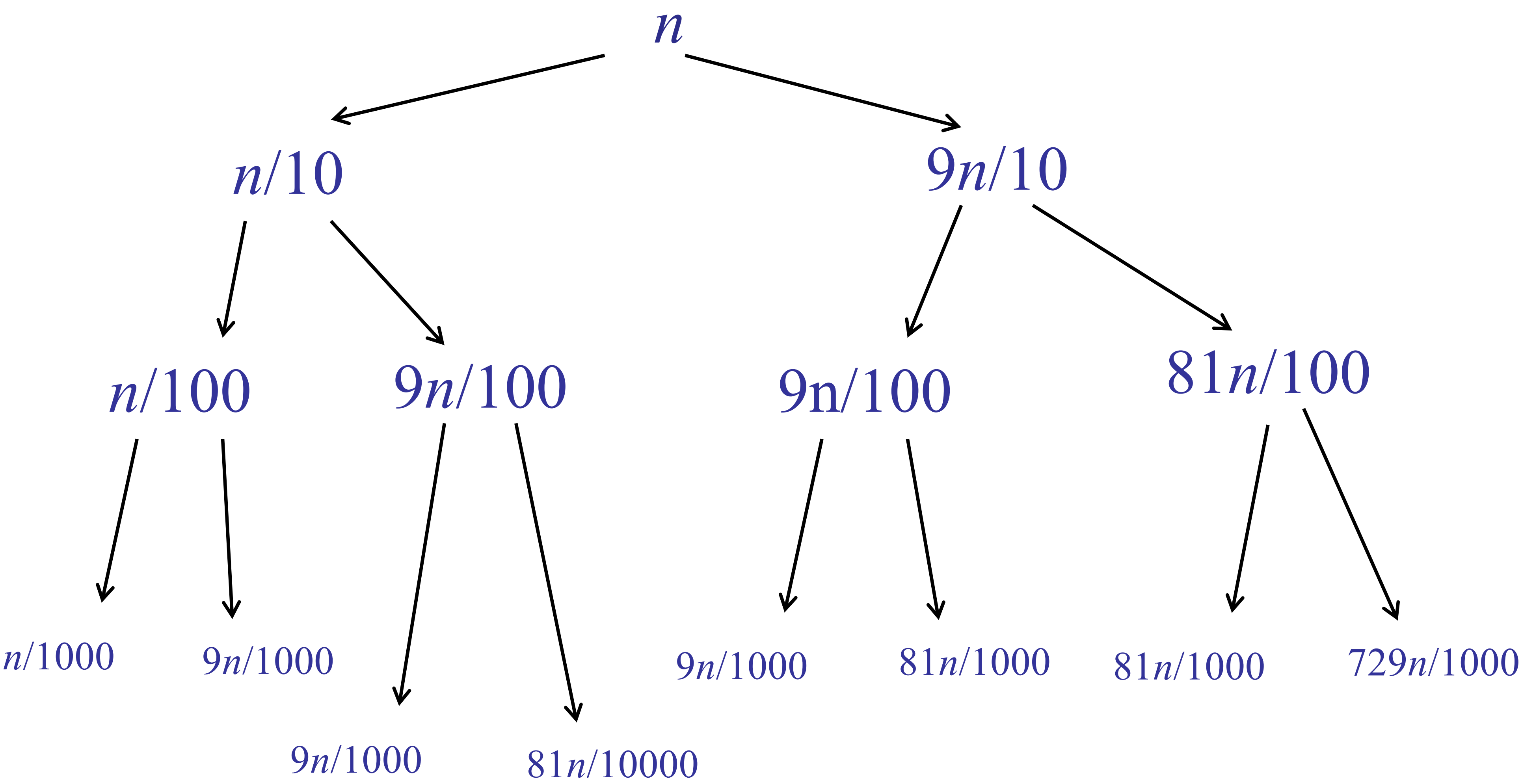
QuickSort Analysis

$k = n/10$

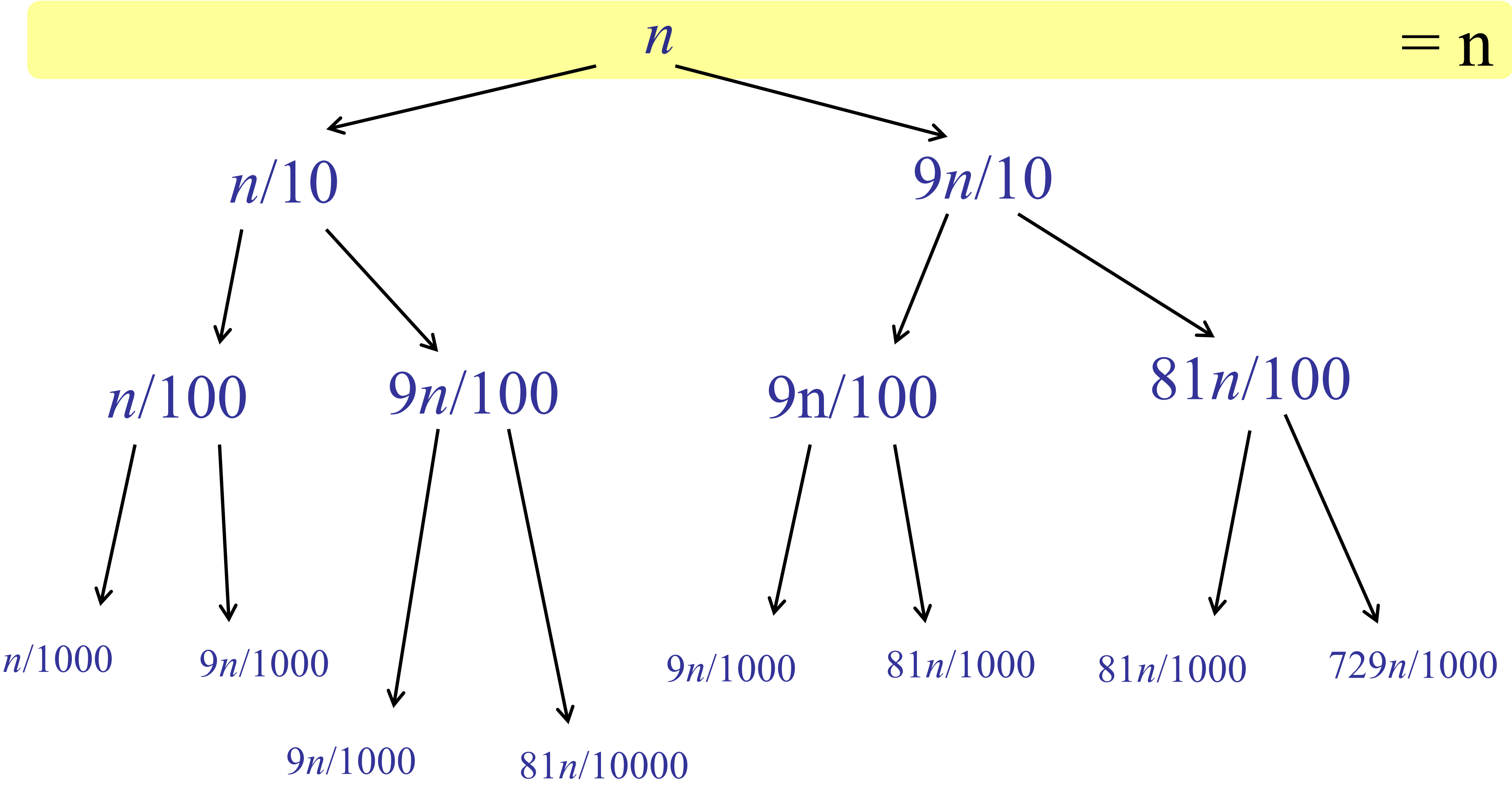


QuickSort Analysis

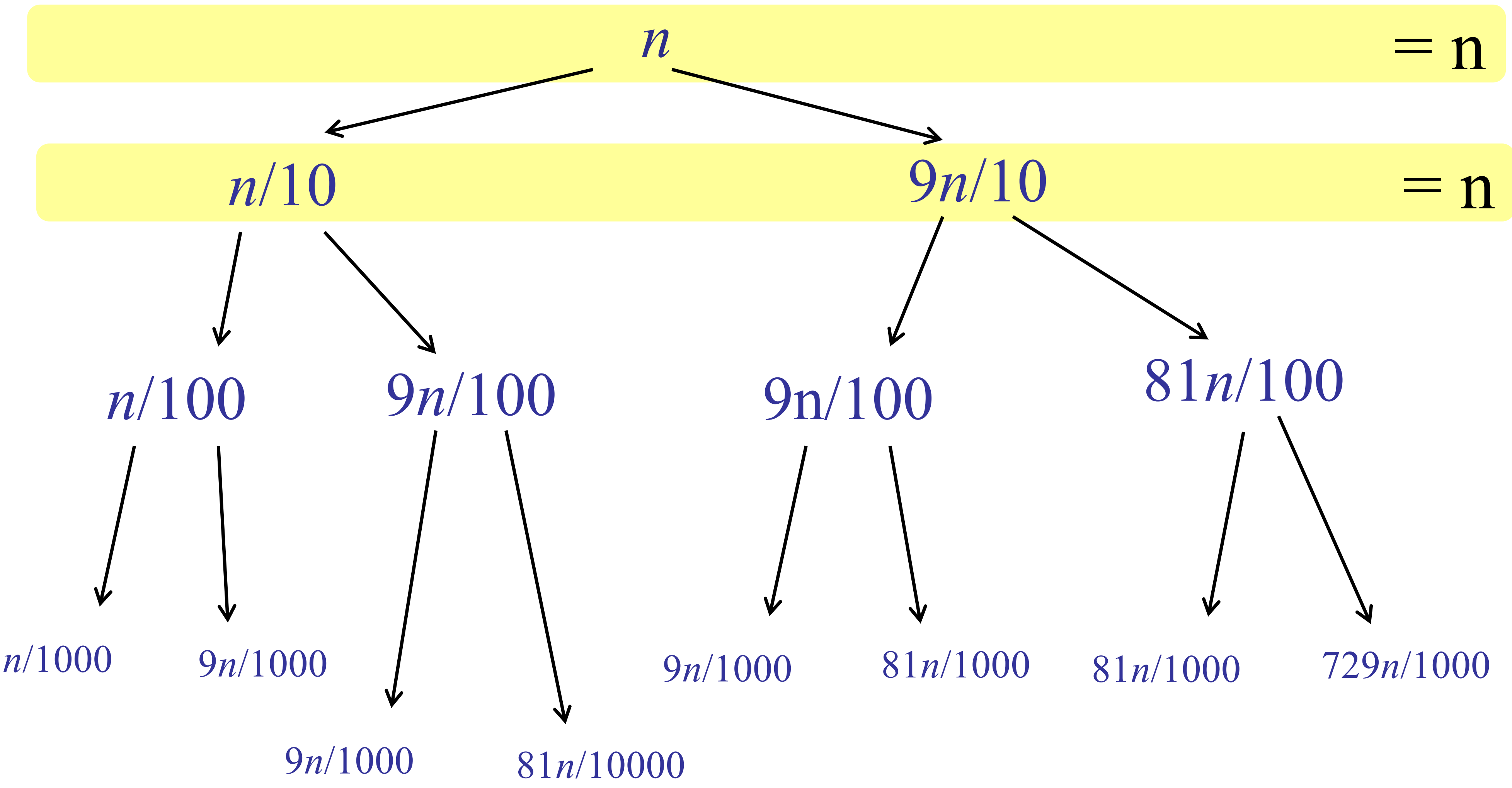
$k = n/10$



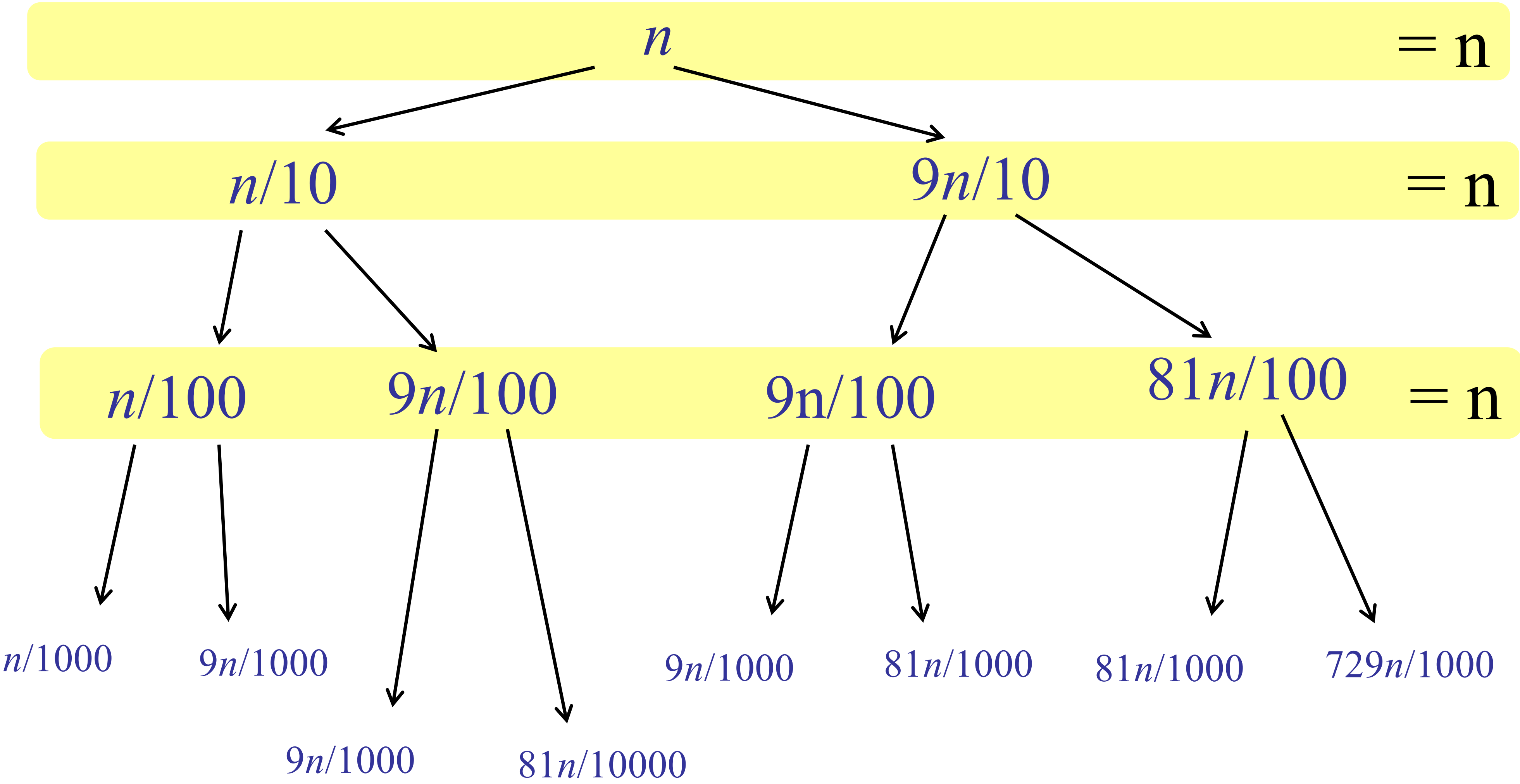
QuickSort Analysis



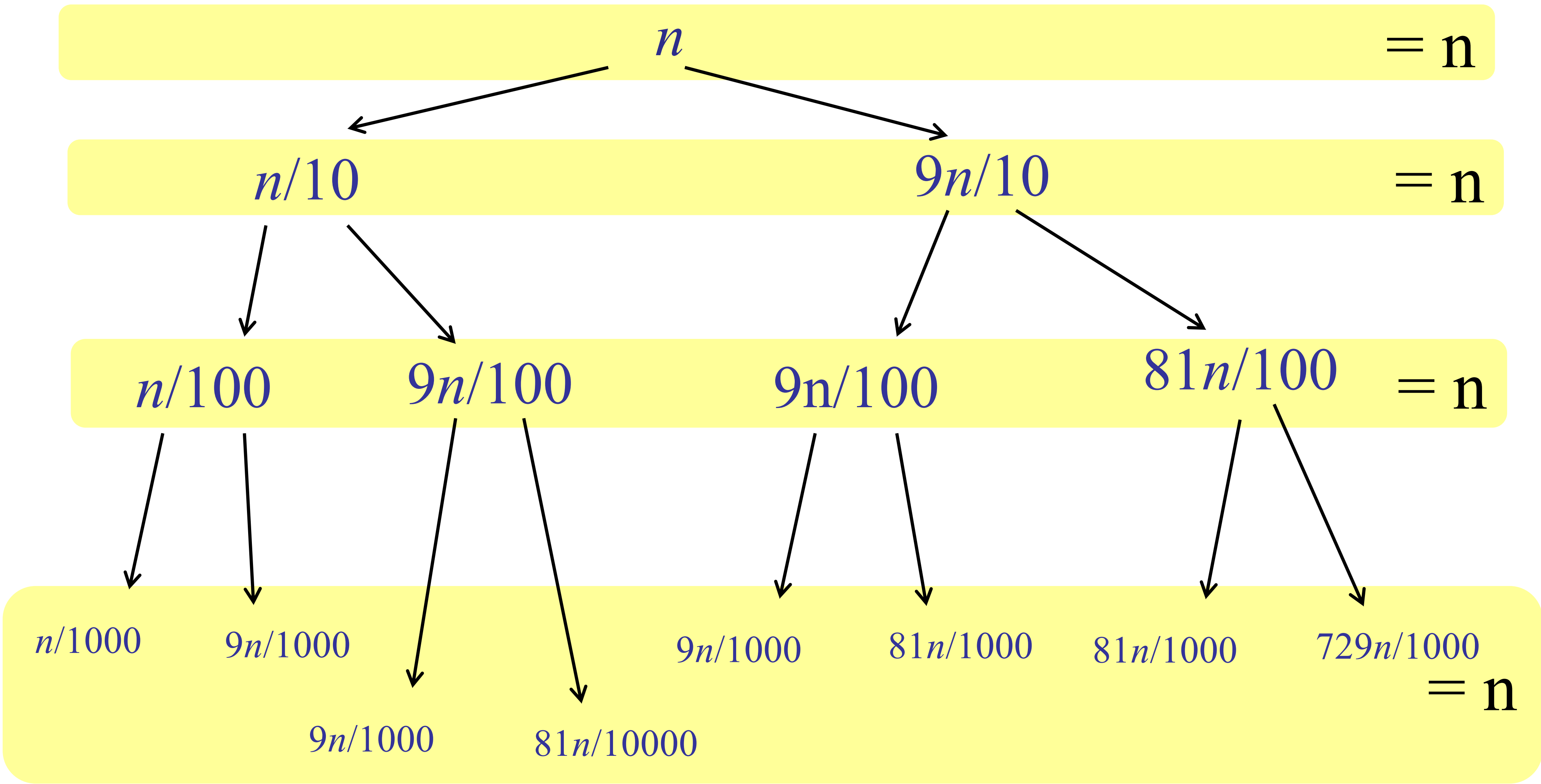
QuickSort Analysis



QuickSort Analysis

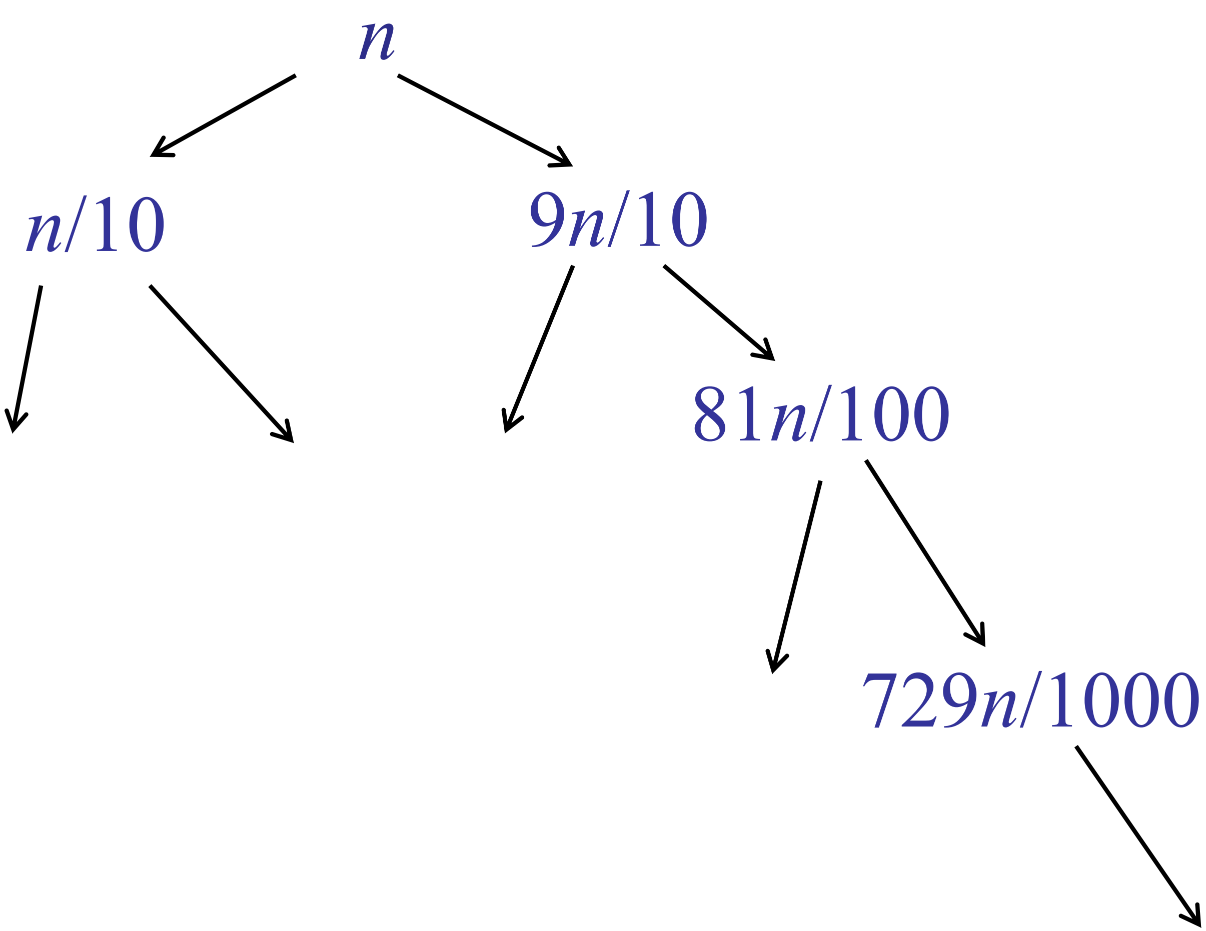


QuickSort Analysis



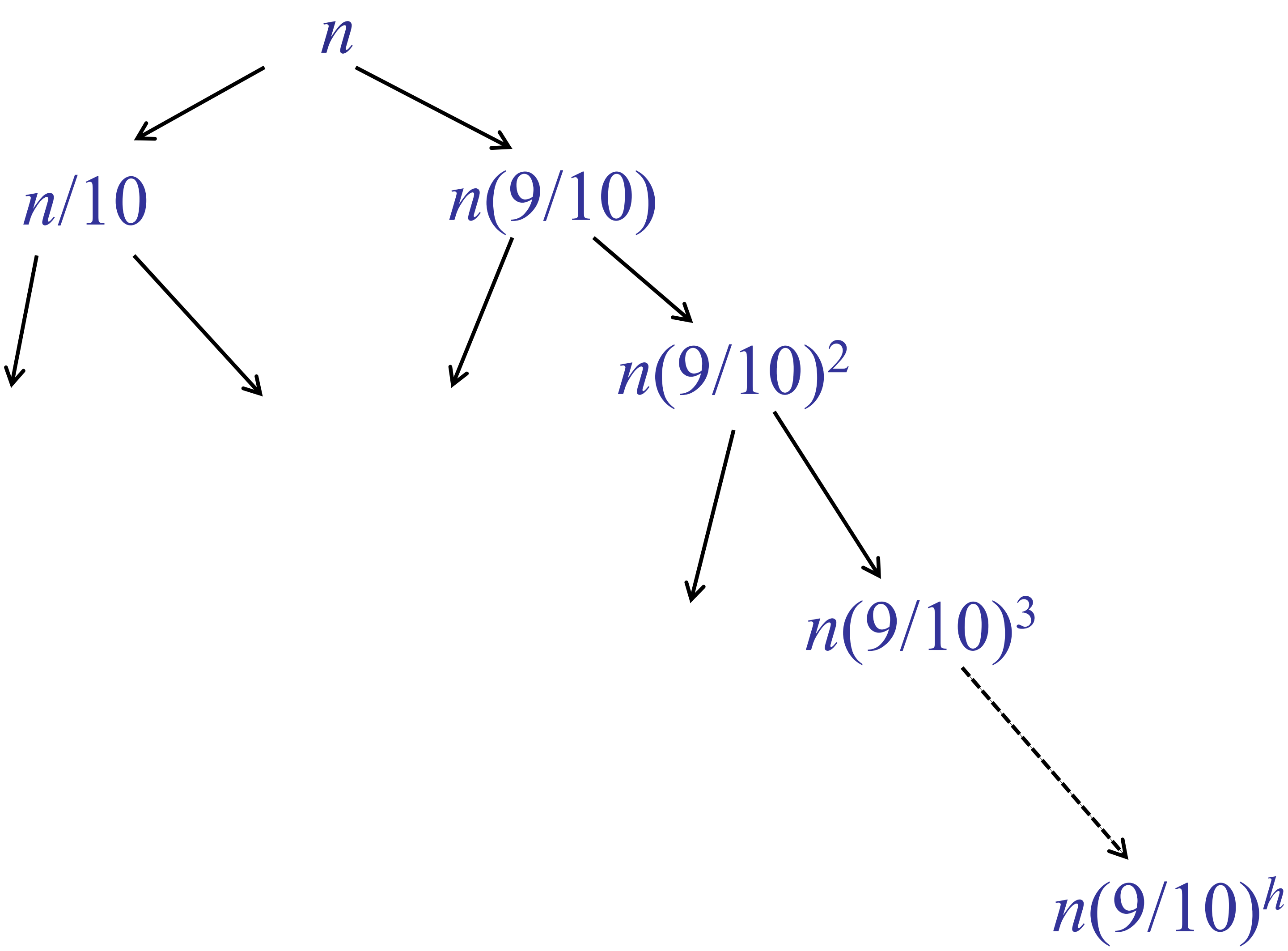
How many levels??

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:

$< n(9/10)$

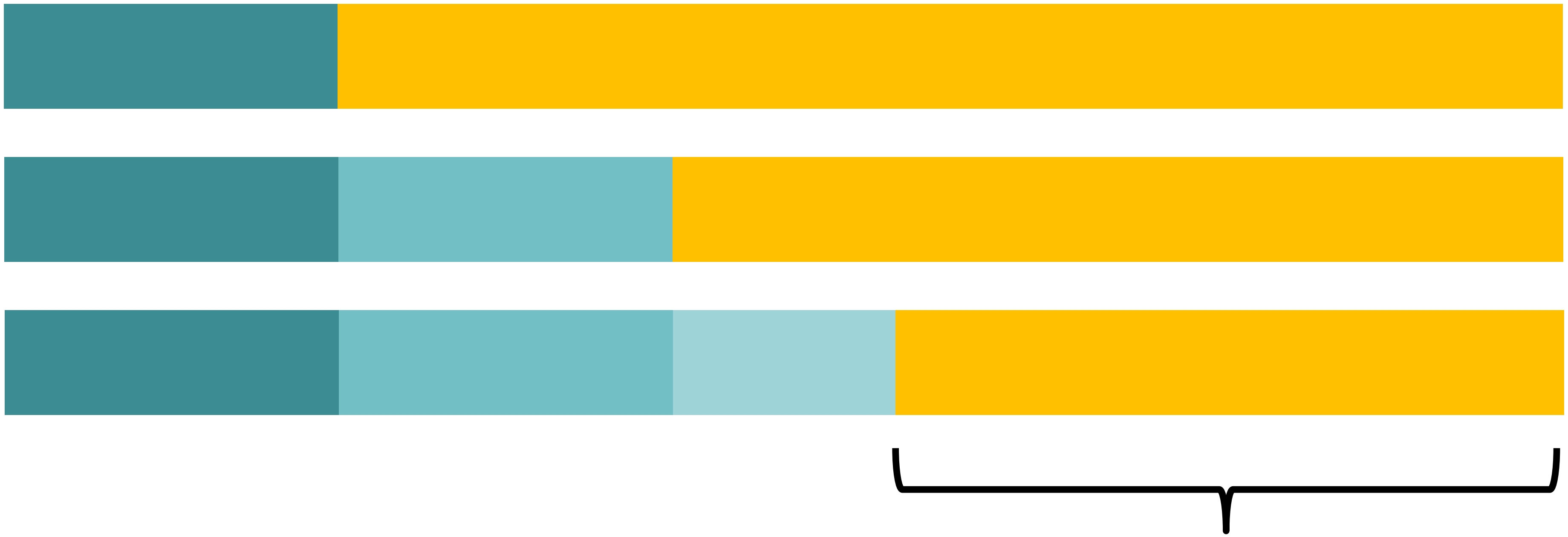
QuickSort Analysis

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

$< n(9/10)^2$

QuickSort Analysis

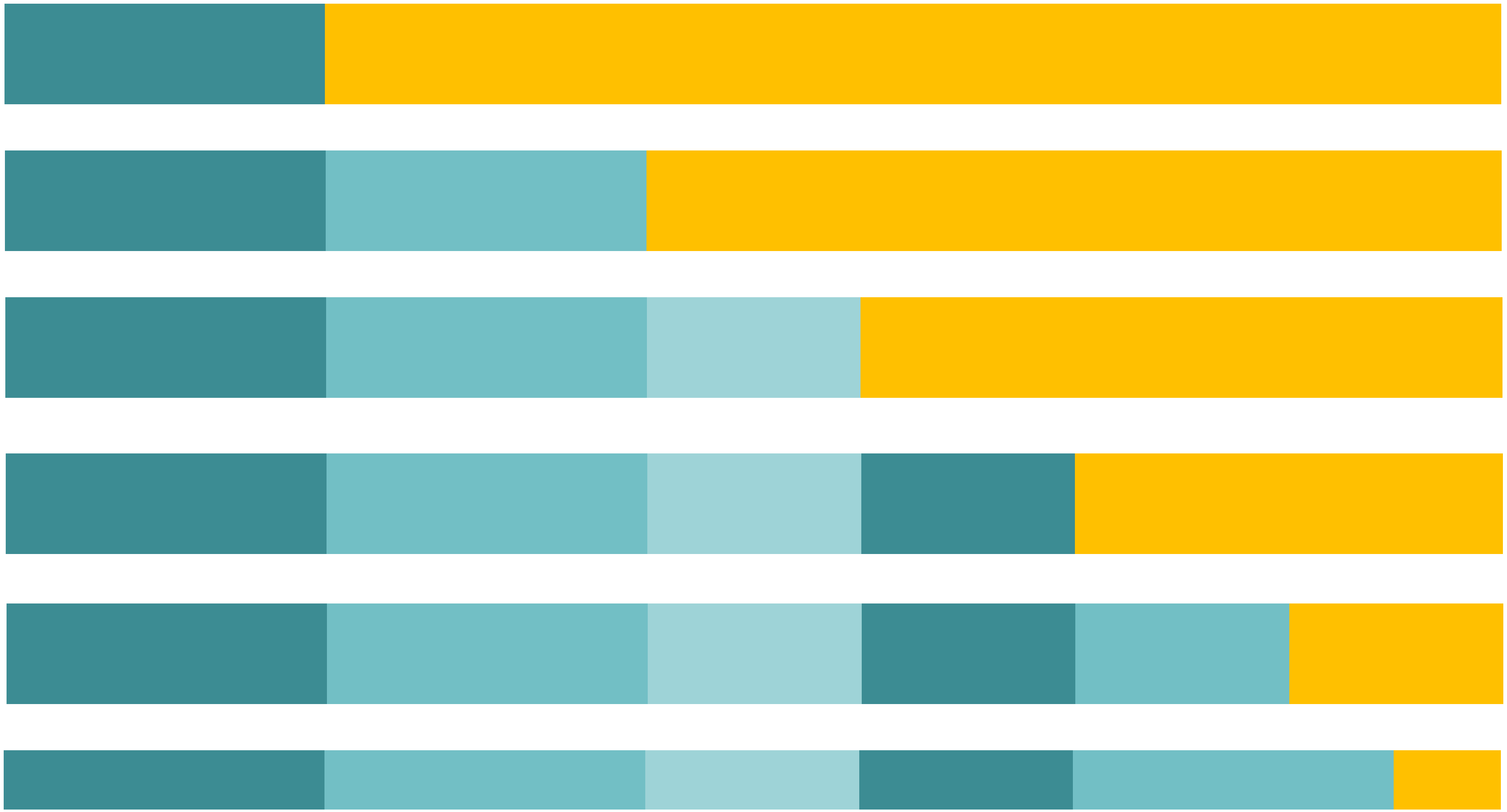
Assume larger part shrinks by at least 9/10 every iteration:



$< n(9/10)^3$

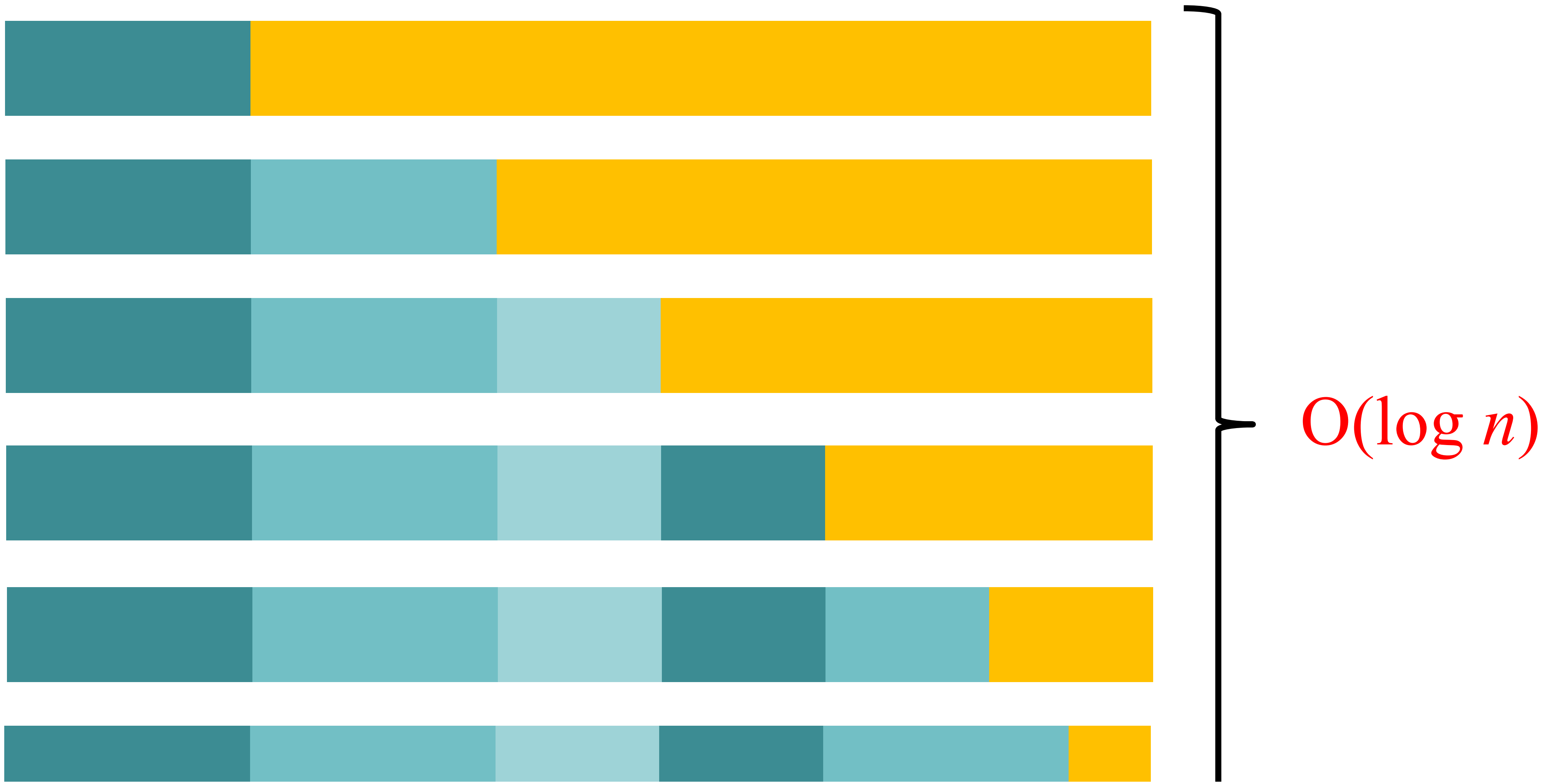
QuickSort Analysis

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)
```

$< x$	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

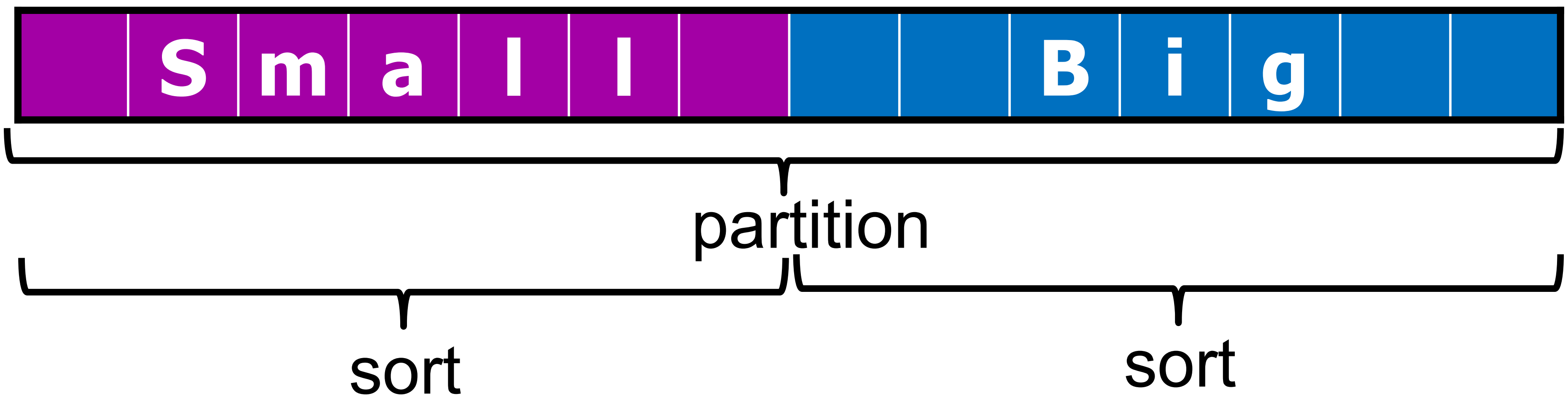
Randomized algorithm:

- 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 = 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) → ParanoidQuickSort(A[1..n], n)

1 → **if** (n == 1) **then** return;

else

?? → **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

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

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

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

Coin flips are independent:

- Pr(heads → heads) = ½ * ½ = ¼
- Pr(heads → tails → 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

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:

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

Expected value of A:

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

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
- Pr(heads, tails) = ¼
- Pr(tails, heads) = ¼
- Pr(tails, tails) = ¼

$$2 * \tfrac{1}{4} = \tfrac{1}{2}$$
$$1 * \tfrac{1}{4} = \tfrac{1}{4}$$
$$1 * \tfrac{1}{4} = \tfrac{1}{4}$$
$$0 * \tfrac{1}{4} = 0$$

1

Probability Theory

Flipping a coin:

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

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

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:

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

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(heads) = p
- Pr(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)
```

QuickSort Partition

Remember:

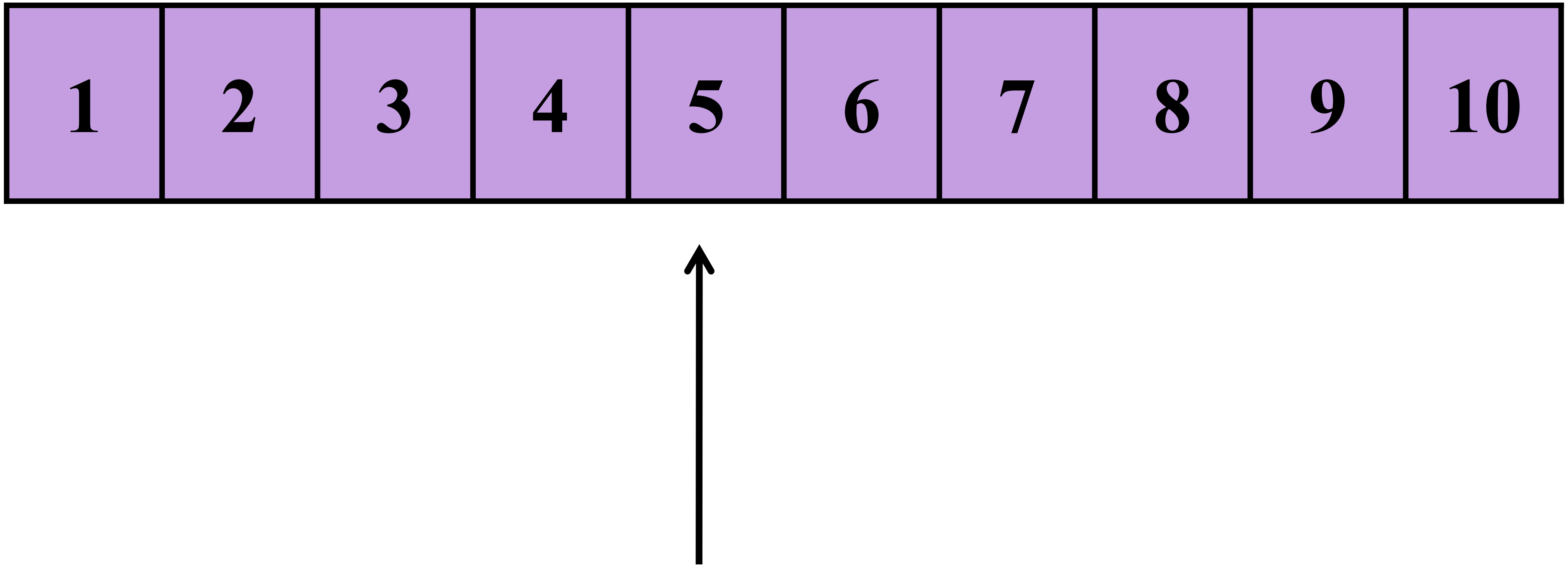
A *pivot* is **good** if it divides the array into two pieces, each of which is size at least $n/10$.

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

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

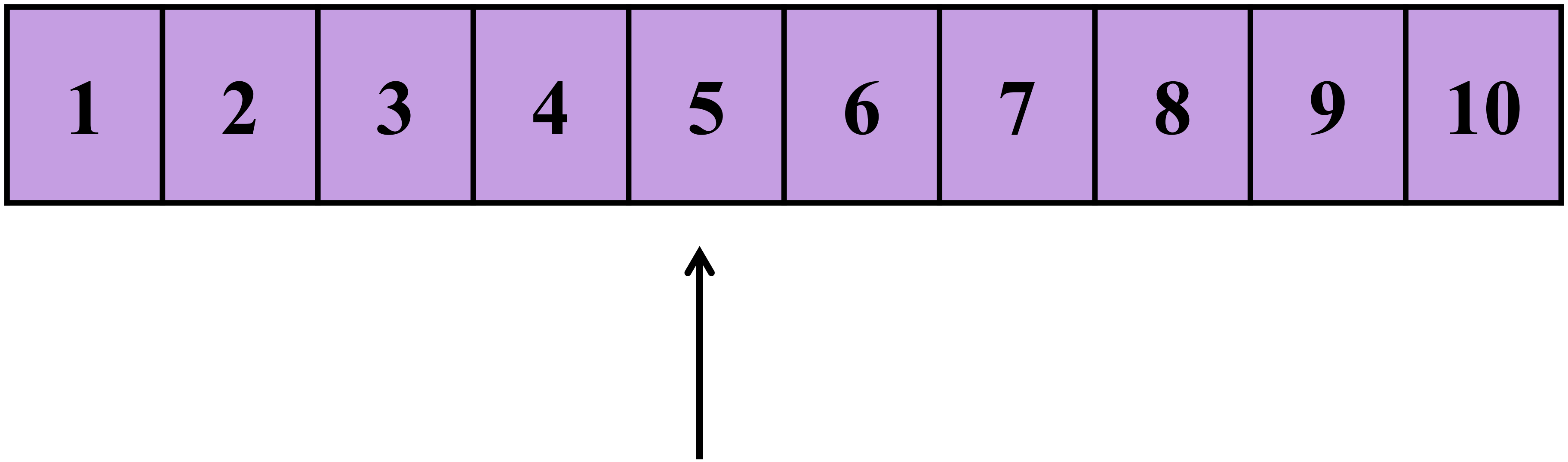
Imagine the array divided into 10 pieces:



Choose a random point at which to partition.

Choosing a Good Pivot

Imagine the array divided into 10 pieces:

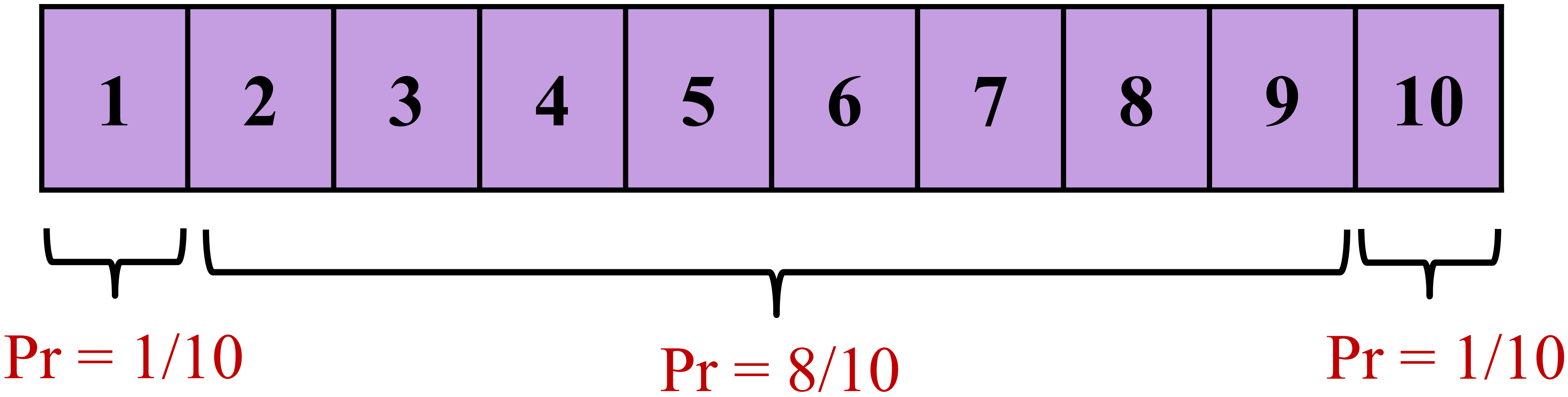


Choose a random point at which to partition.

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

Choosing a Good Pivot

Imagine the array divided into 10 pieces:

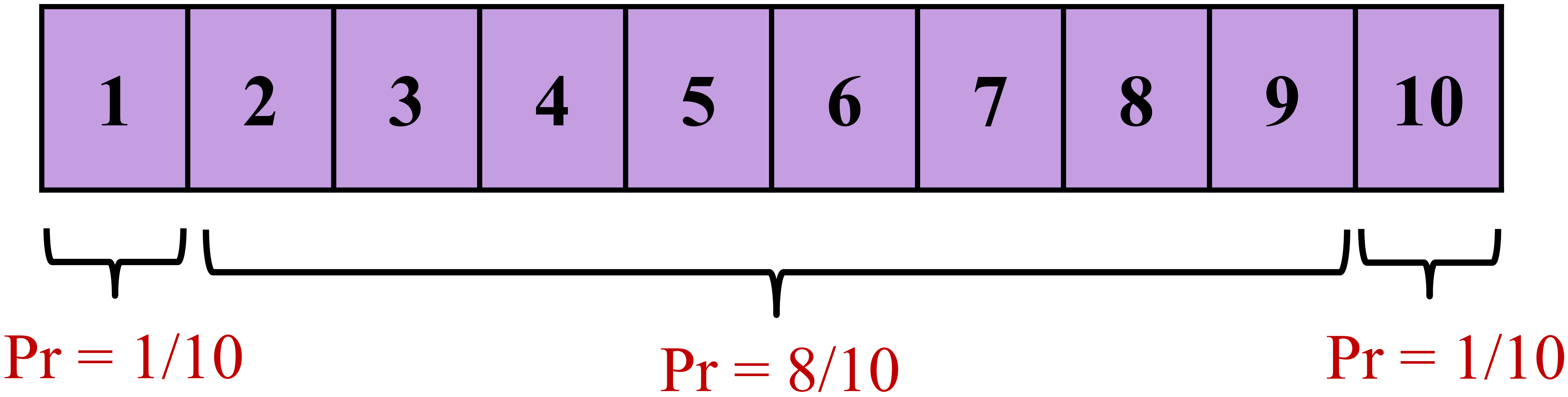


Choose a random point at which to partition.

- 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

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

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

else

repeat

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

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

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

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

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

Paranoid QuickSort

Key claim:

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

Then we know:

$$\mathbf{E}[\mathrm{T}(n)] = \mathbf{E}[\mathrm{T}(k)] + \mathbf{E}[\mathrm{T}(n-k)] + \mathbf{E}[\#\text{ pivot choices}](n)$$

$$<= \mathbf{E}[\mathrm{T}(k)] + \mathrm{E}[\mathrm{T}(n-k)] + 2n$$

$$= \mathrm{O}(n \log n)$$

Regular QuickSort

Also true:

Expected running time is $O(n \log n)$.

With high probability, running time is $O(n \log n)$.

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:

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