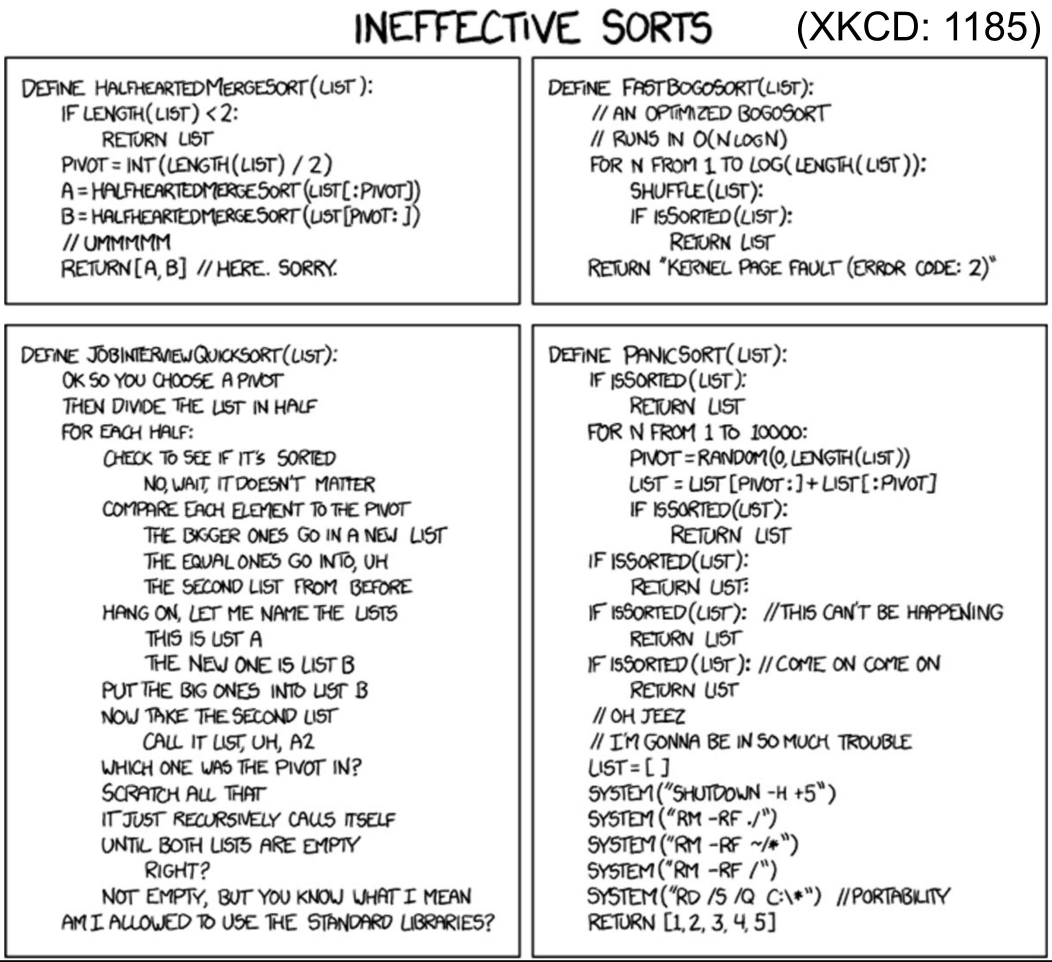
CS2040S
Data Structures and Algorithms



(XKCD: 1185)

Welcome!

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

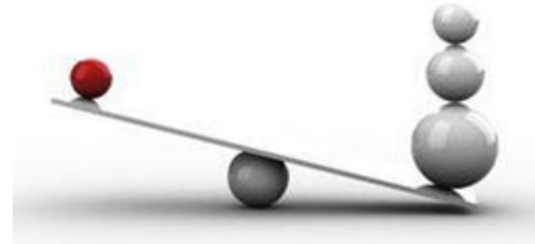
- Running time
- Space usage
- Stability

Sorting, Part II

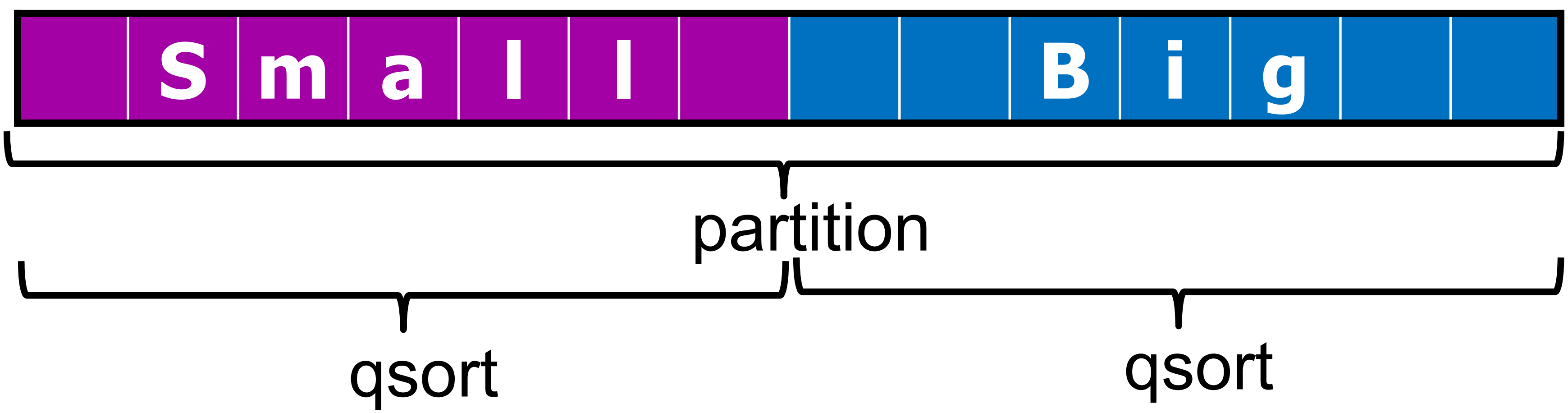
QuickSort

- 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

22	1	6	40	32	10	18	4	50

low
< 22

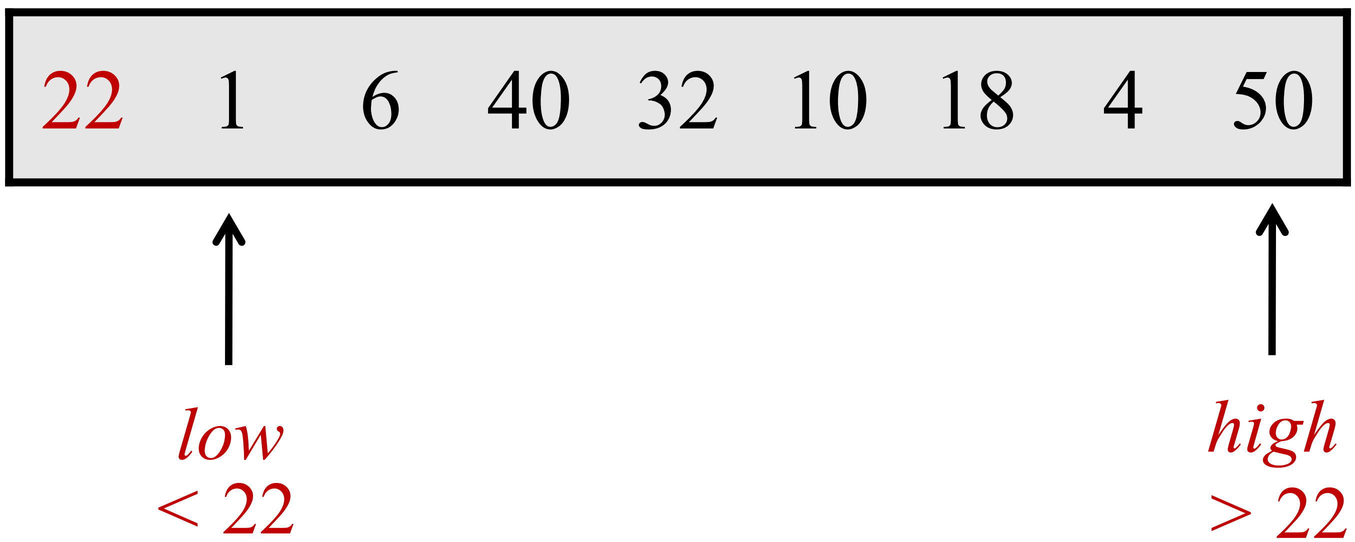
Move until it's bigger than the pivot

high
> 22

Move until it's less than the pivot

Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22

22	1	6	40	32	10	18	4	50

< 22

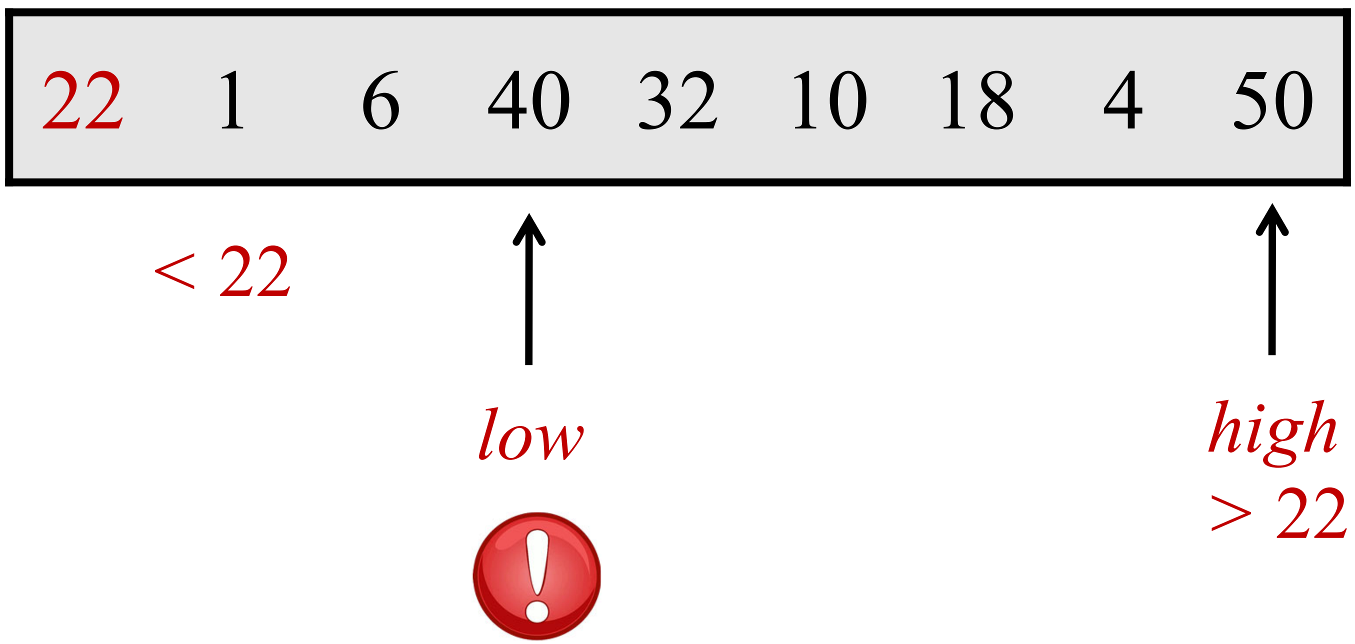
low

high

> 22

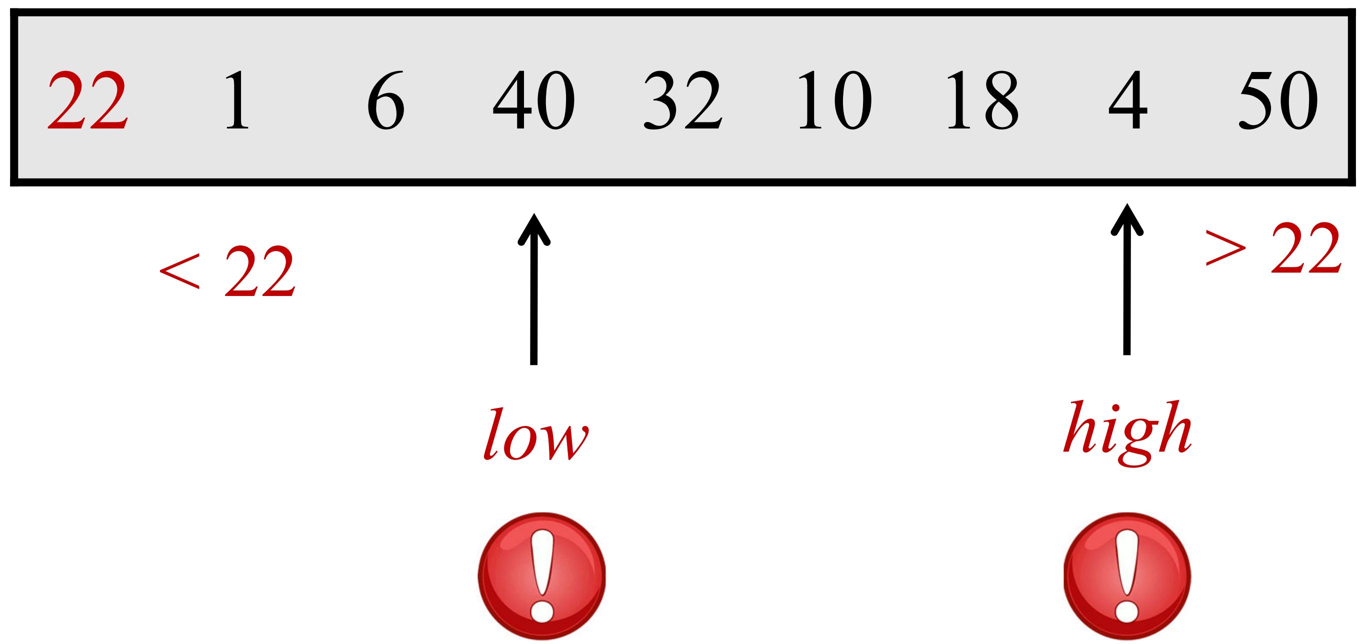
Partitioning an Array

Example: partition around 22



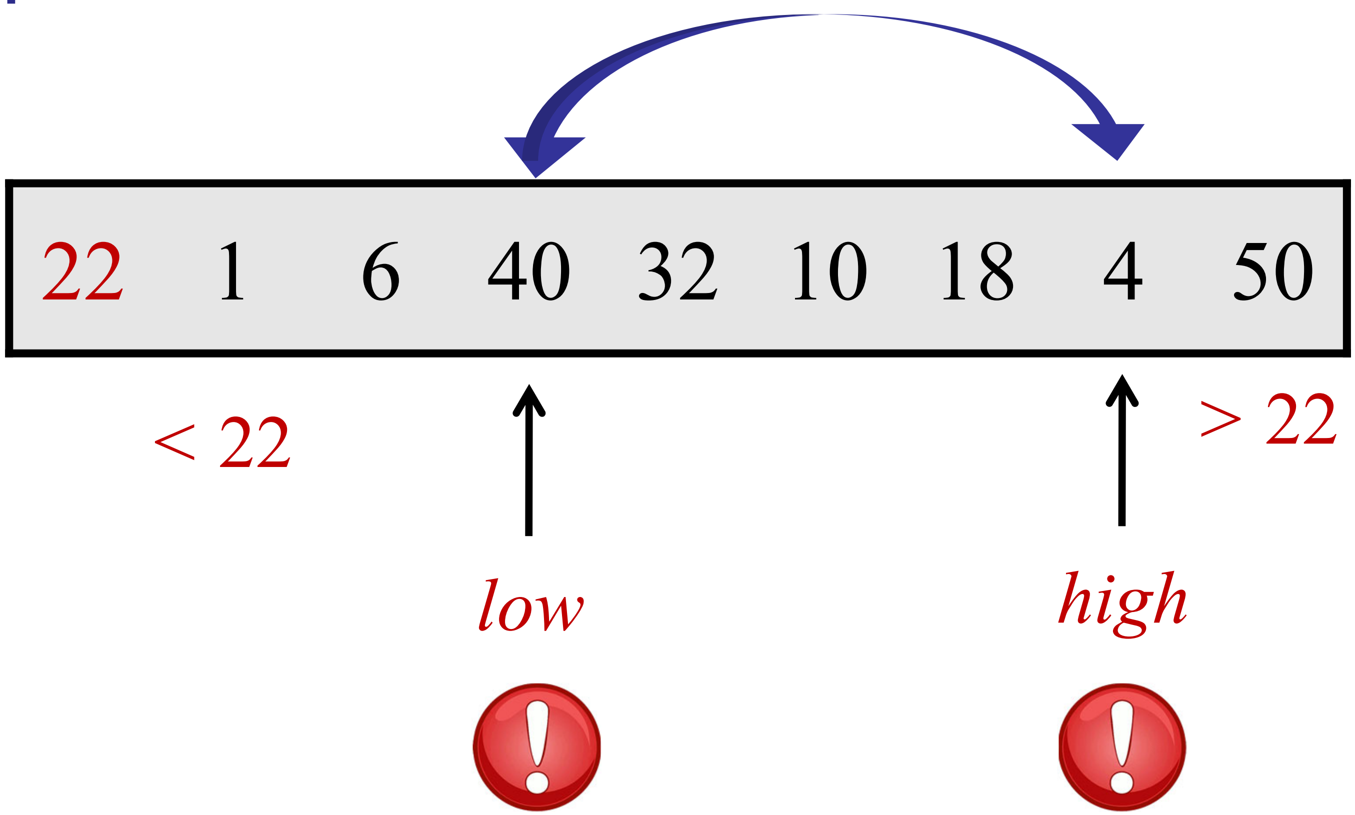
Partitioning an Array

Example: partition around 22



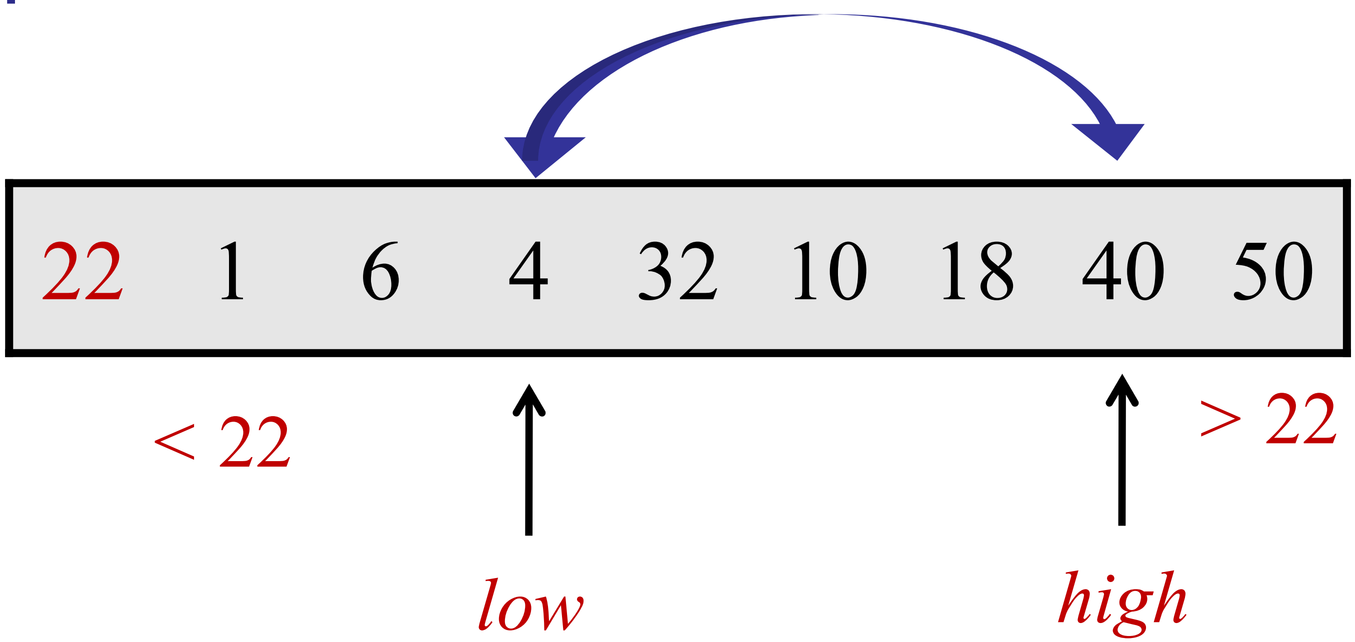
Partitioning an Array

Example: partition around 22



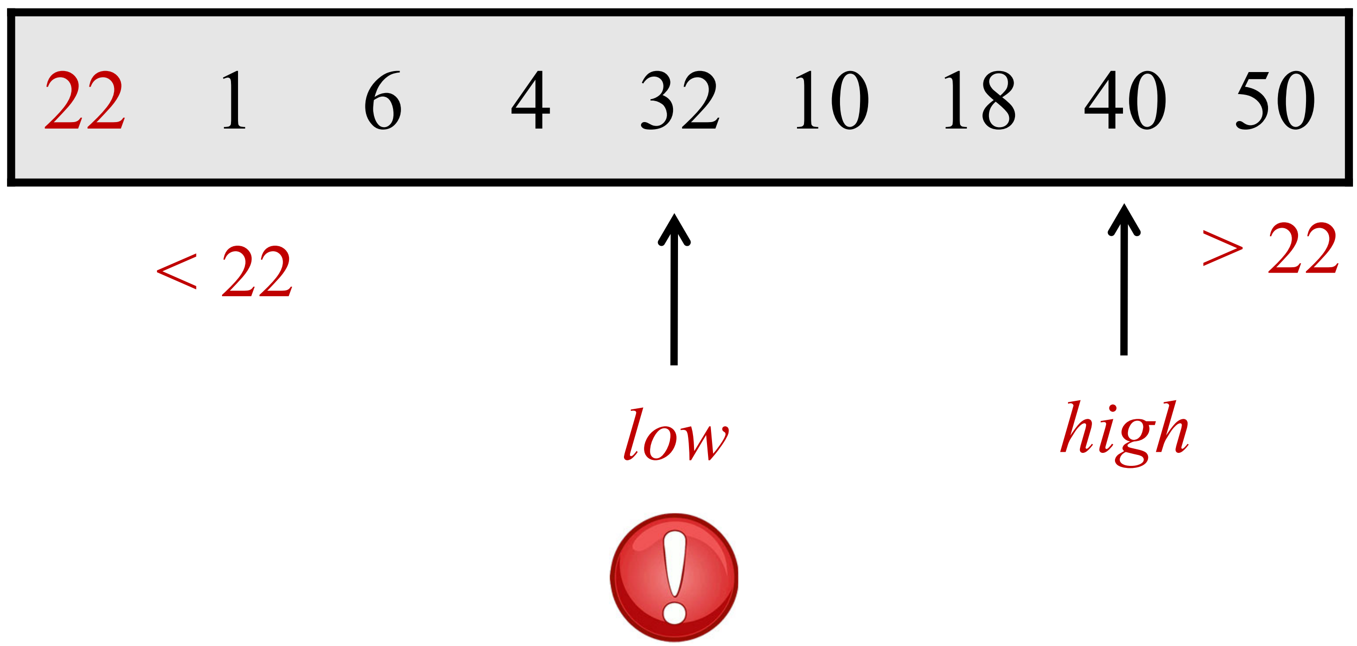
Partitioning an Array

Example: partition around 22



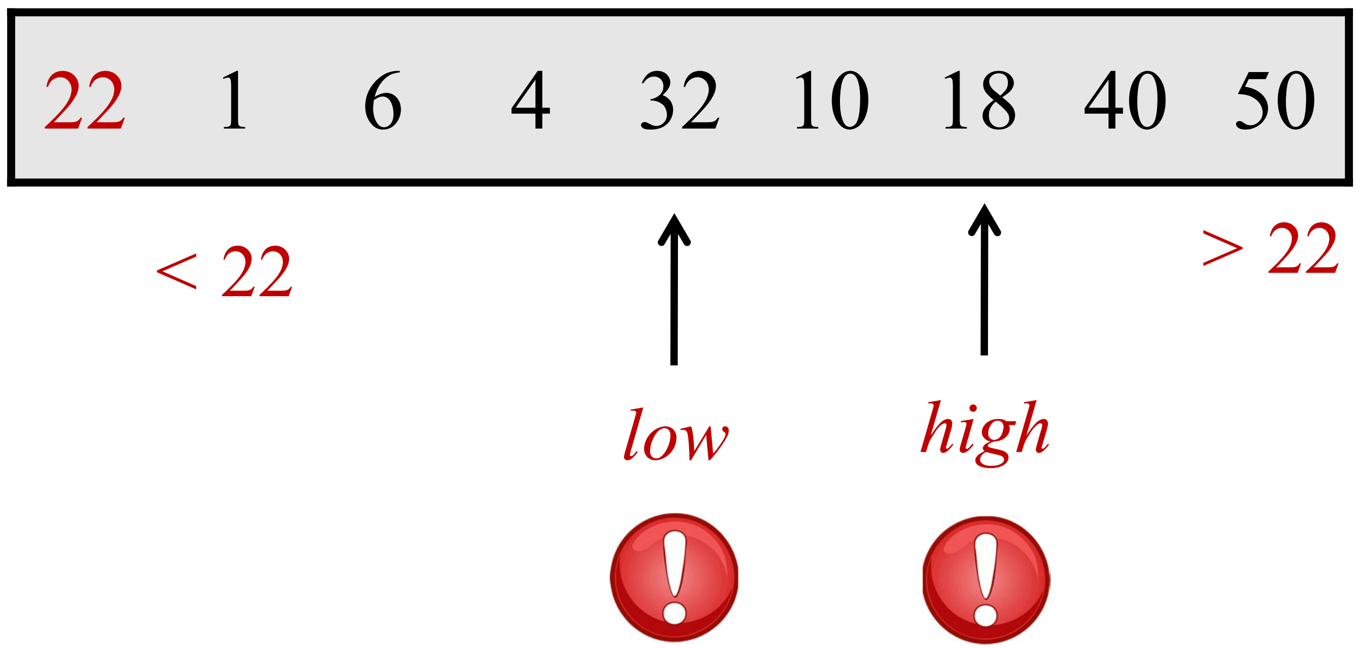
Partitioning an Array

Example: partition around 22



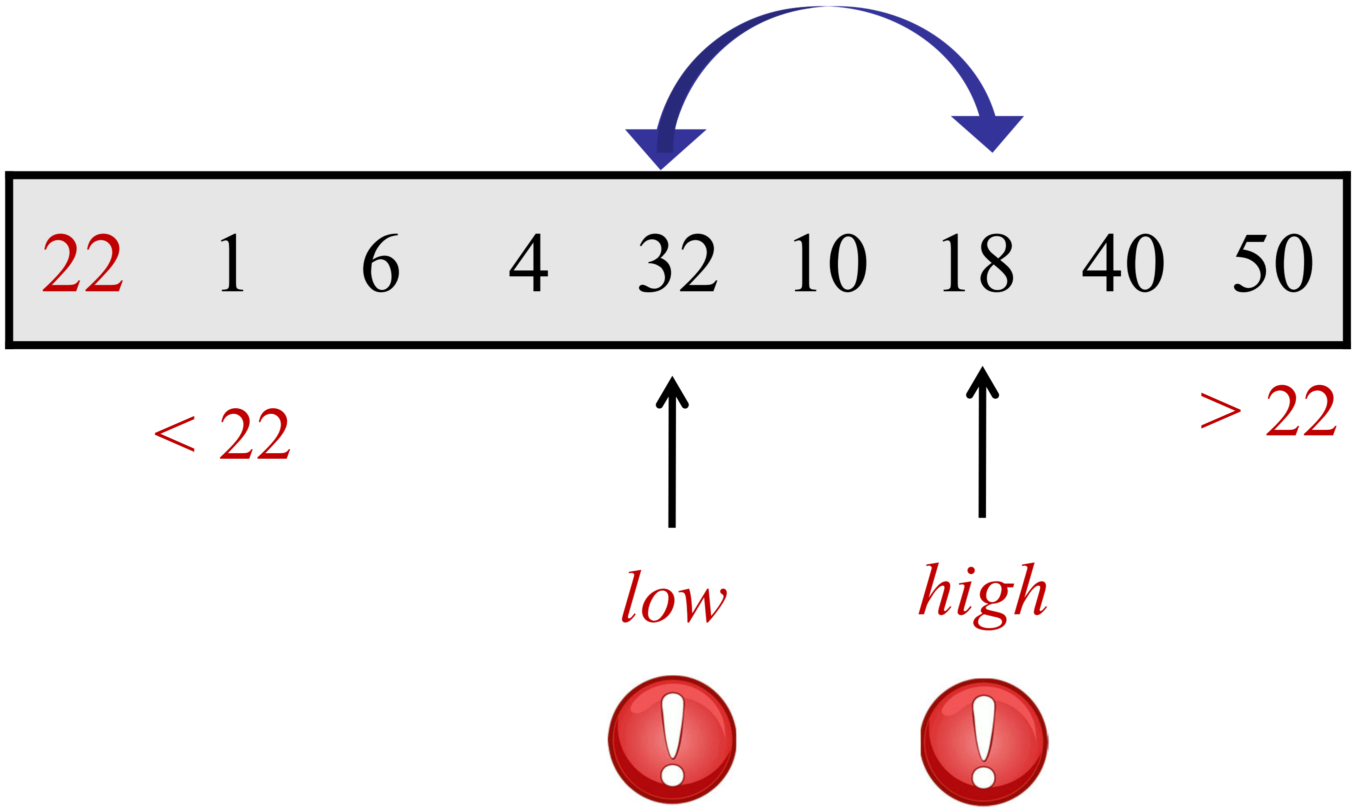
Partitioning an Array

Example: partition around 22



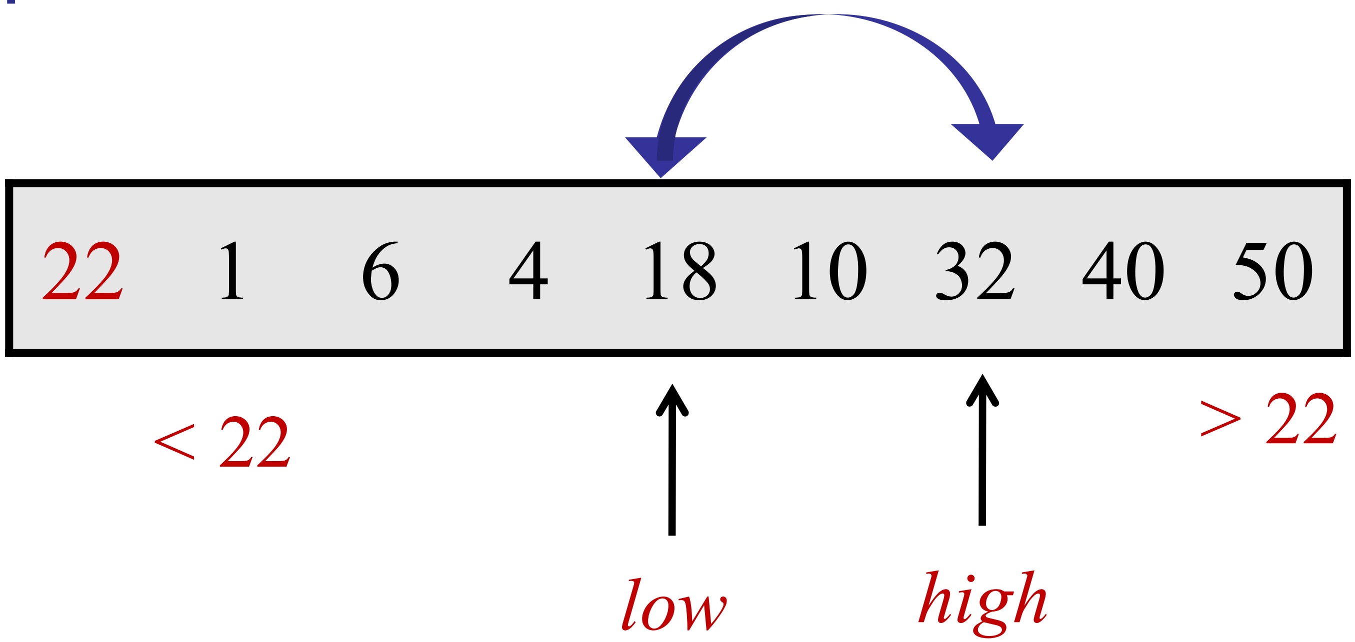
Partitioning an Array

Example: partition around 22



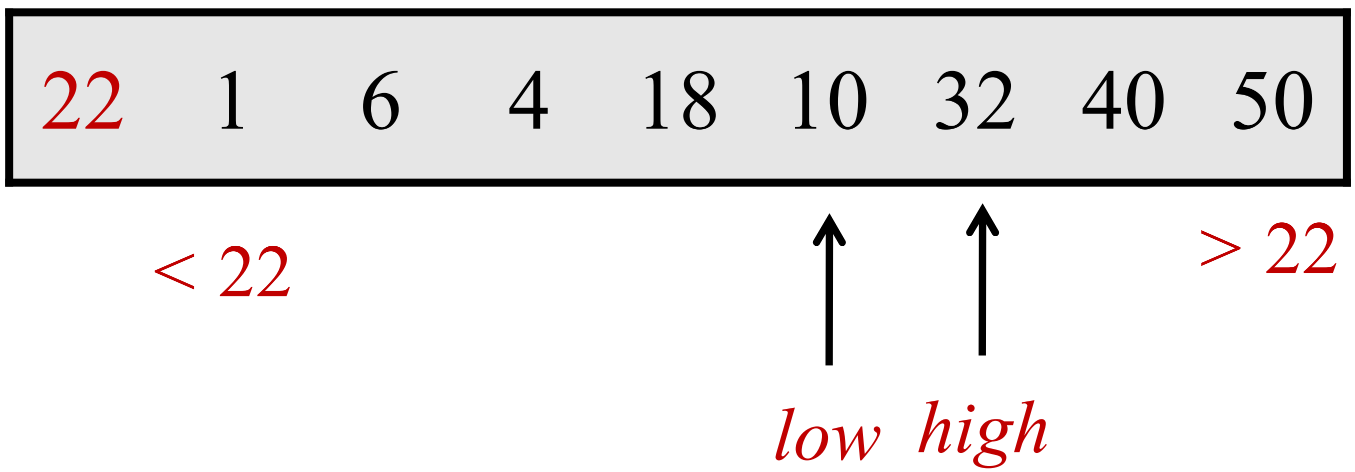
Partitioning an Array

Example: partition around 22



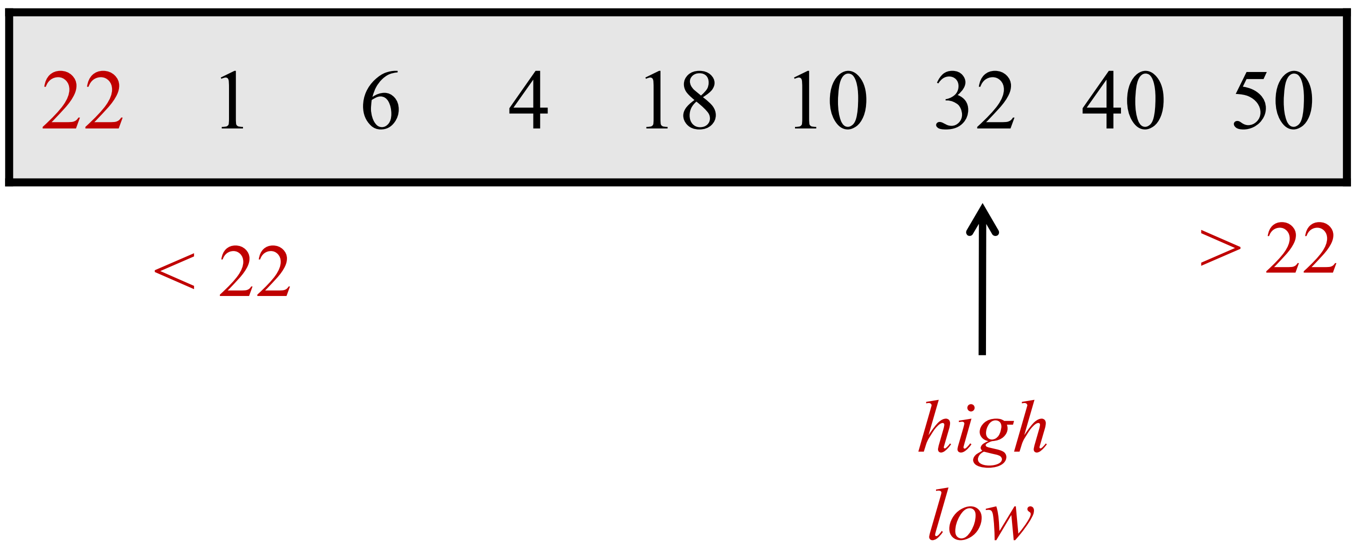
Partitioning an Array

Example: partition around 22



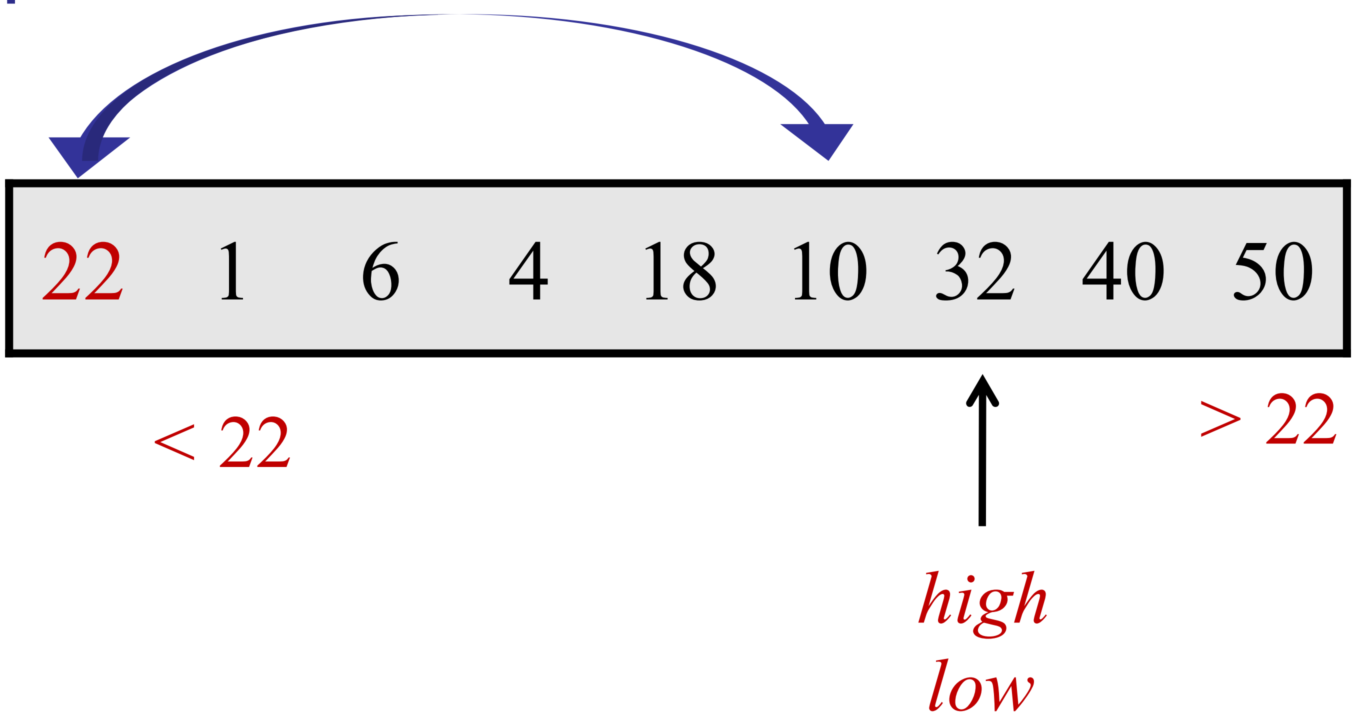
Partitioning an Array

Example: partition around 22



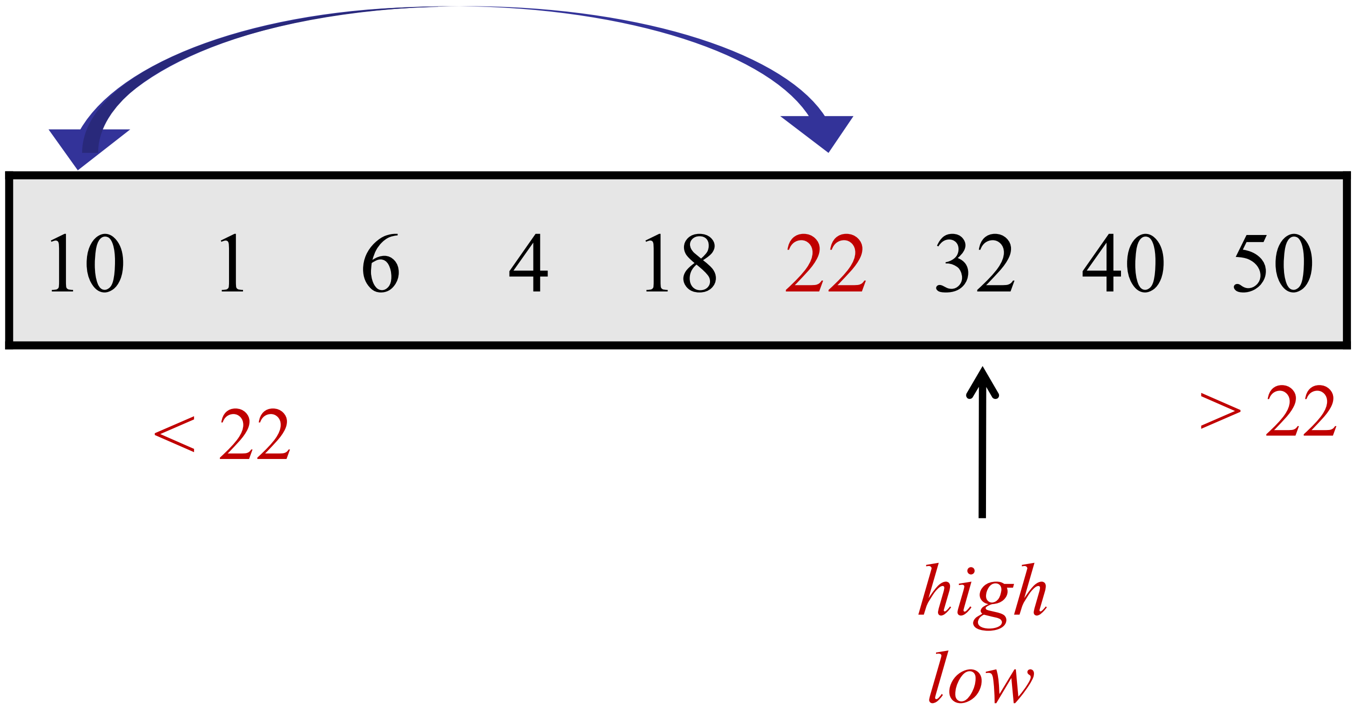
Partitioning an Array

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 = 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 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

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

ARCHIPELAGO

is open

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;

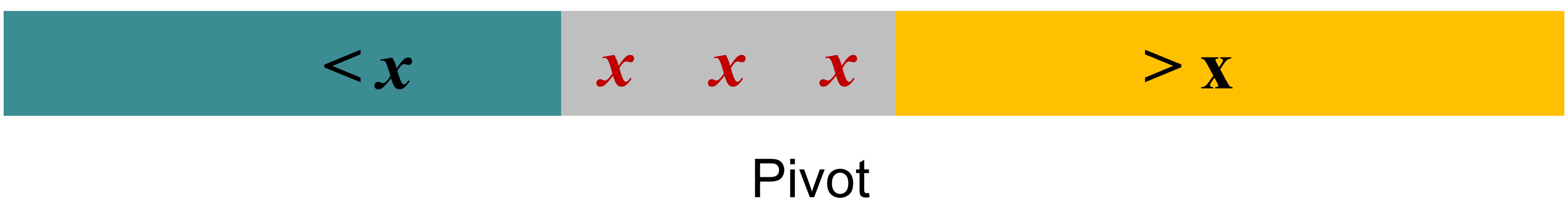
    else

        Choose pivot index *pIndex*.

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

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

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



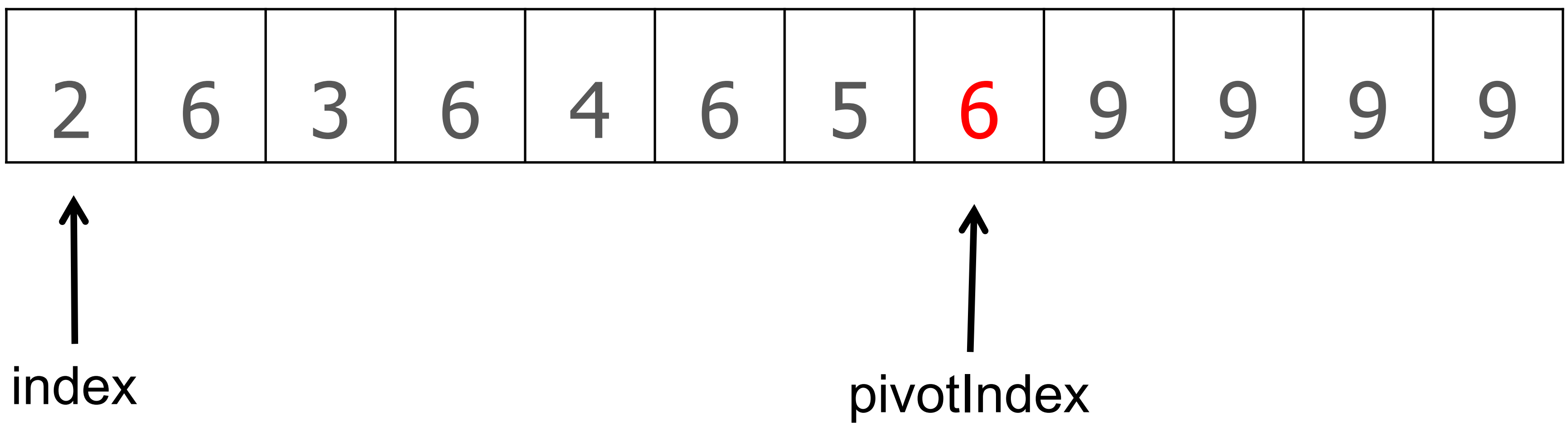
Duplicates

3-Way Partitioning

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

Pack Duplicates

Example:



Pack Duplicates

Example:

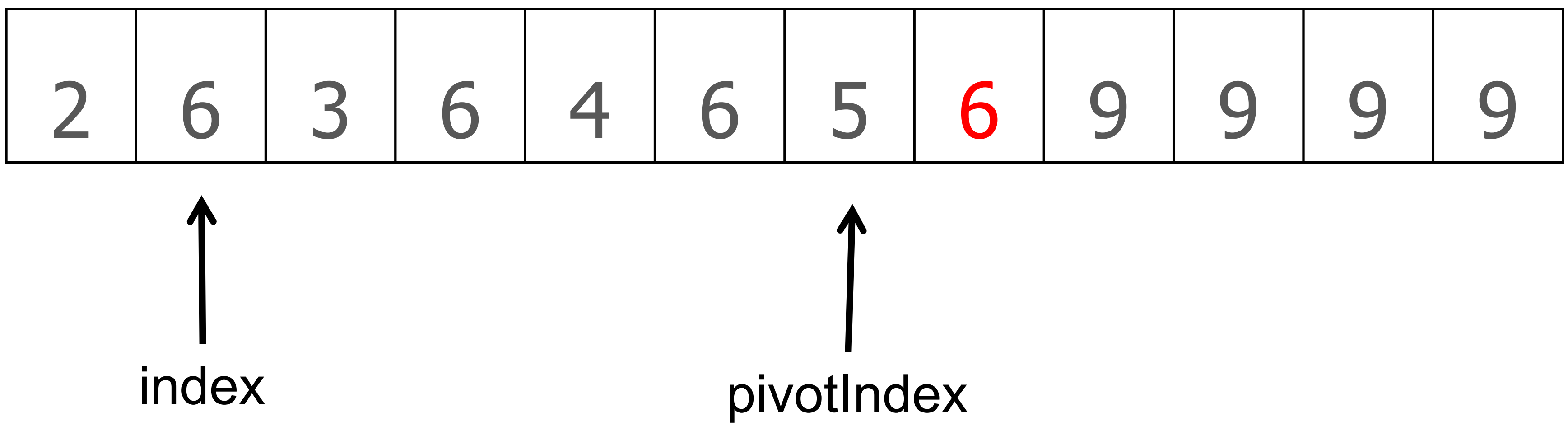
2	6	3	6	4	6	5	6	9	9	9	9

index

pivotIndex

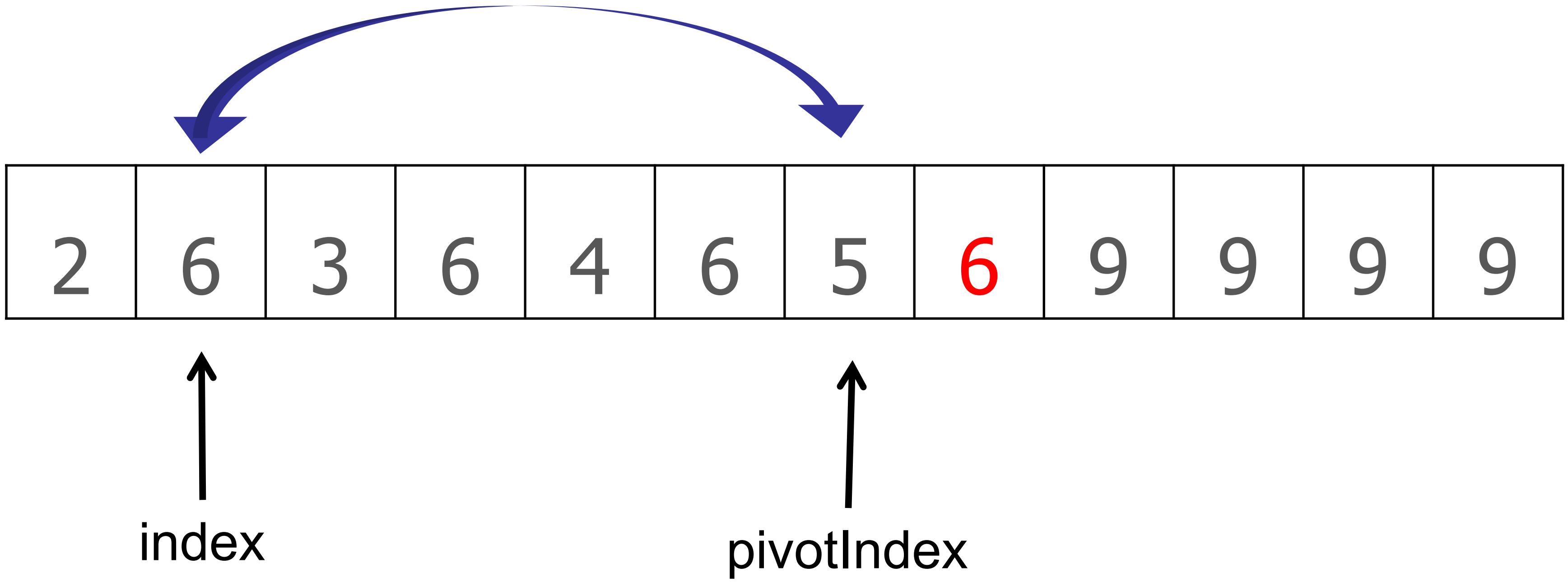
Pack Duplicates

Example:



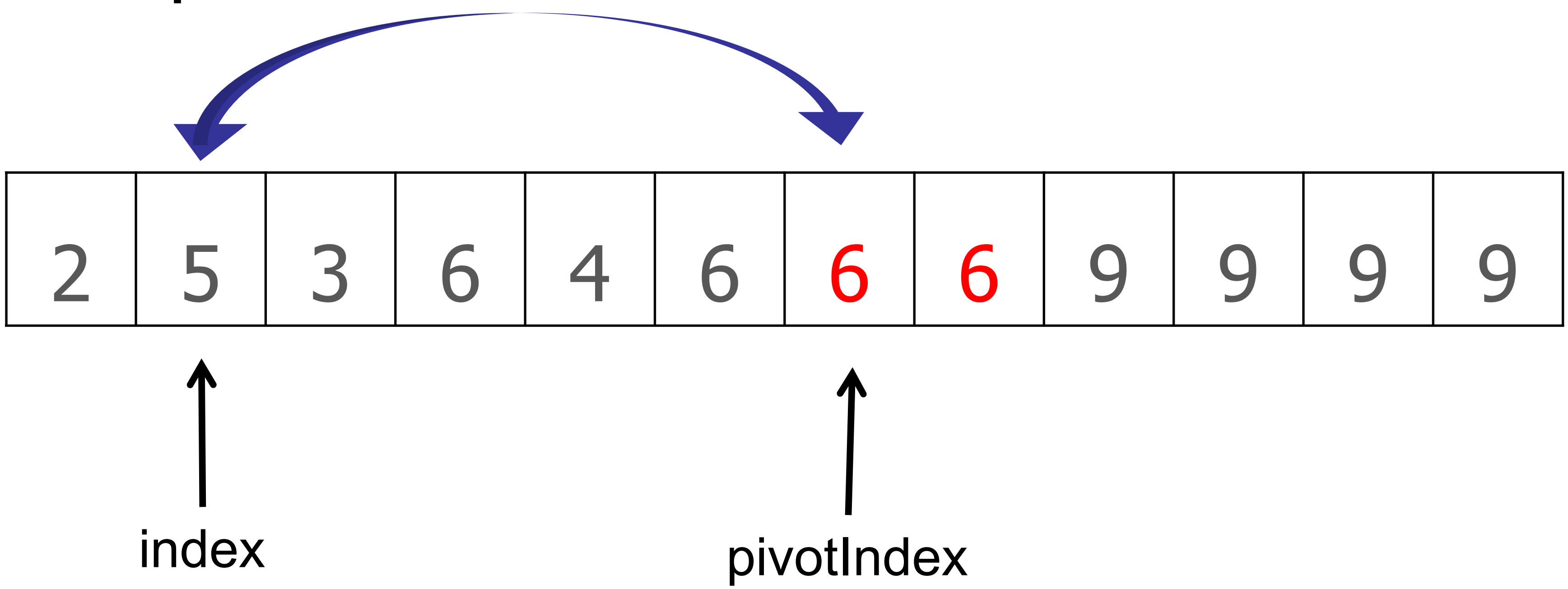
Pack Duplicates

Example:



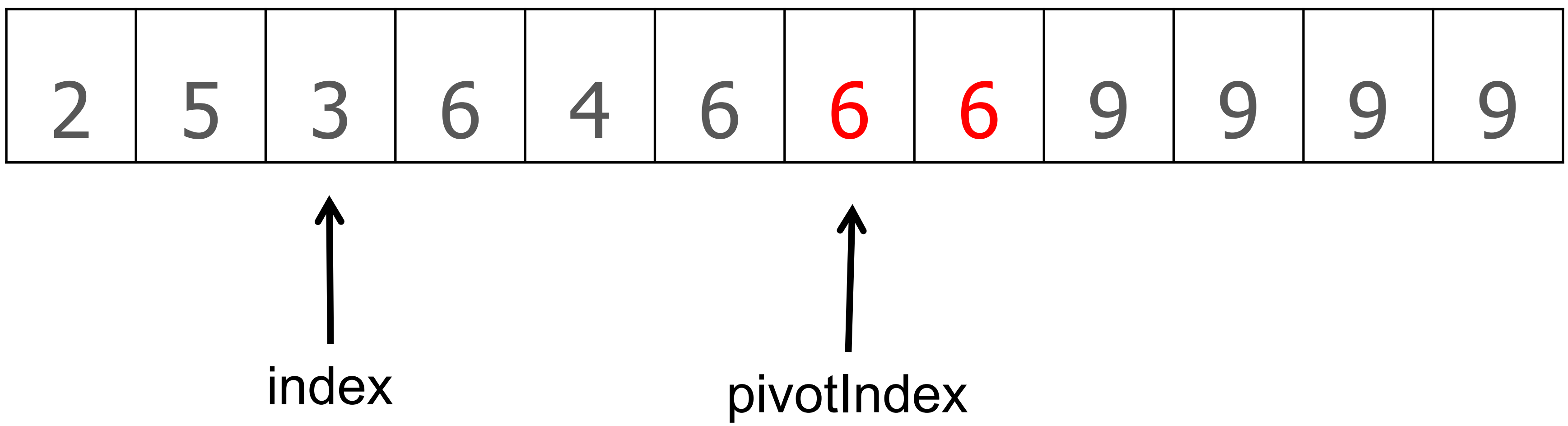
Pack Duplicates

Example:



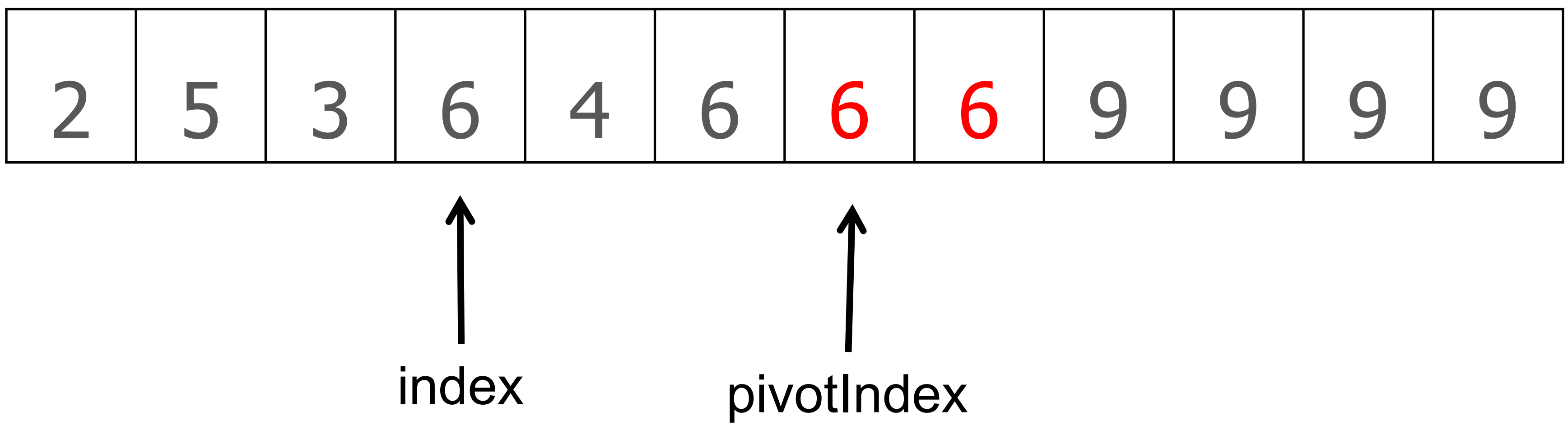
Pack Duplicates

Example:



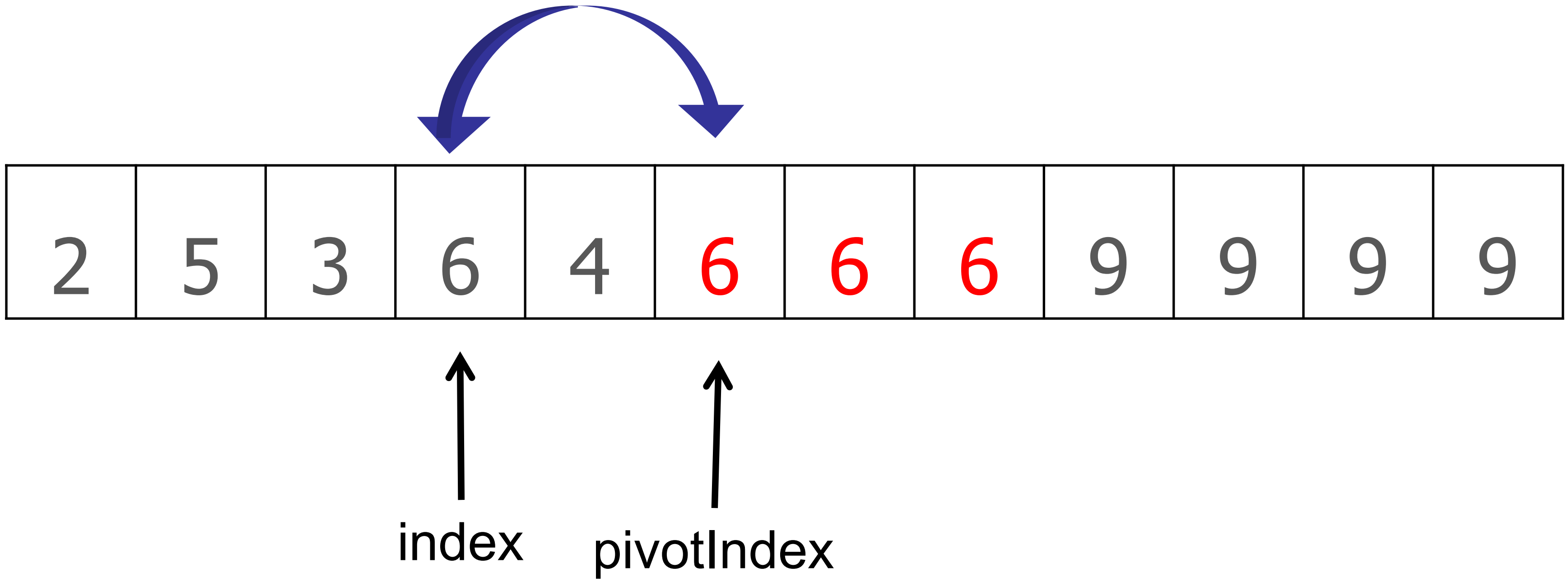
Pack Duplicates

Example:



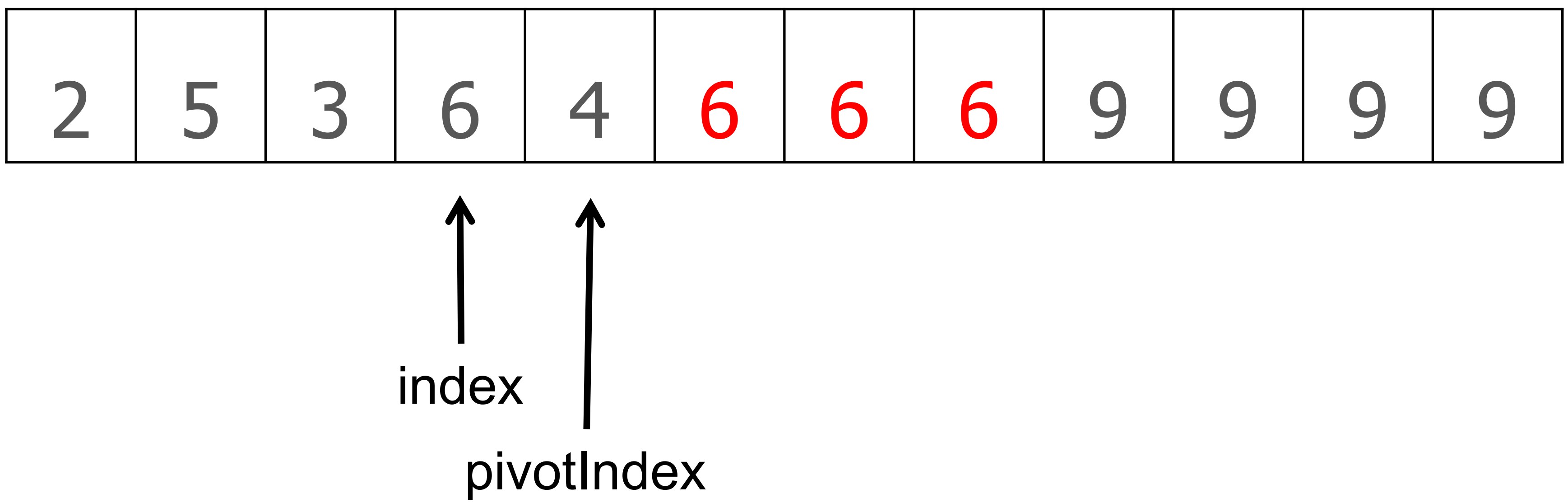
Pack Duplicates

Example:



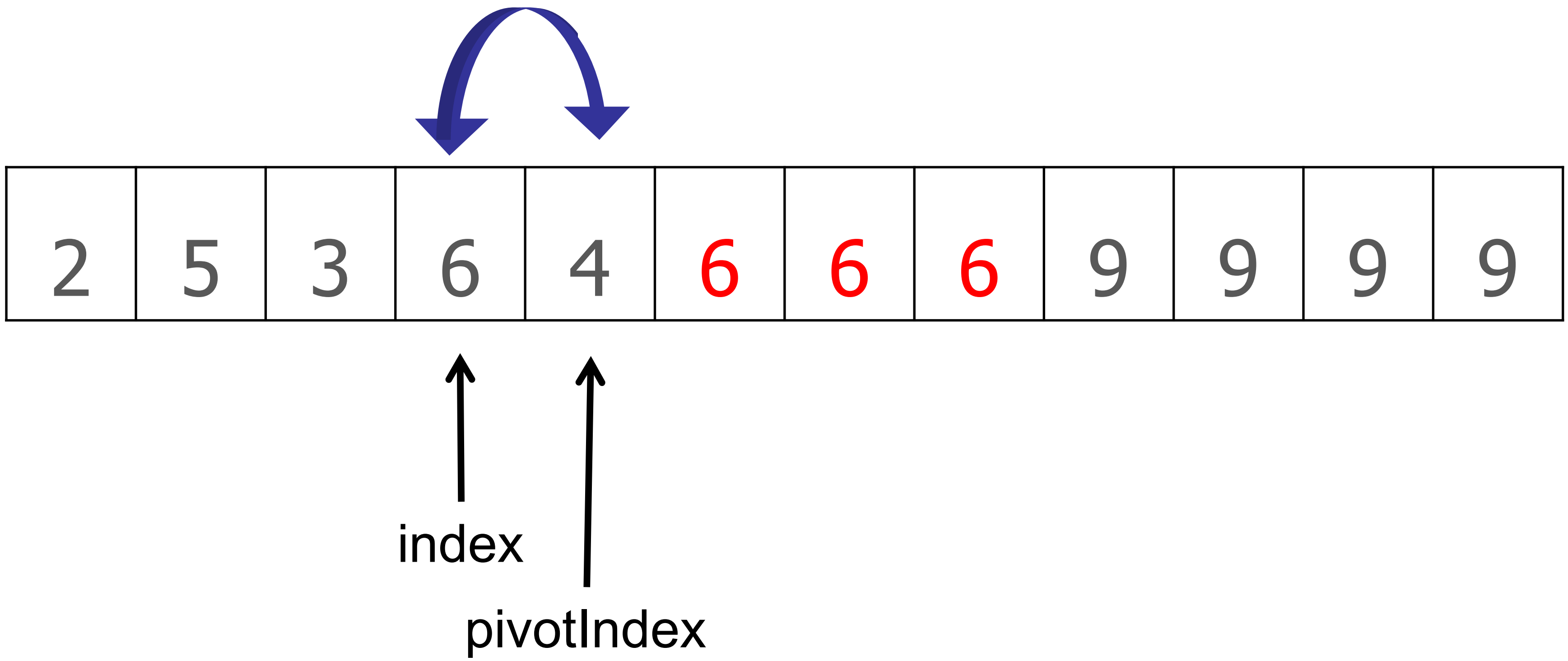
Pack Duplicates

Example:



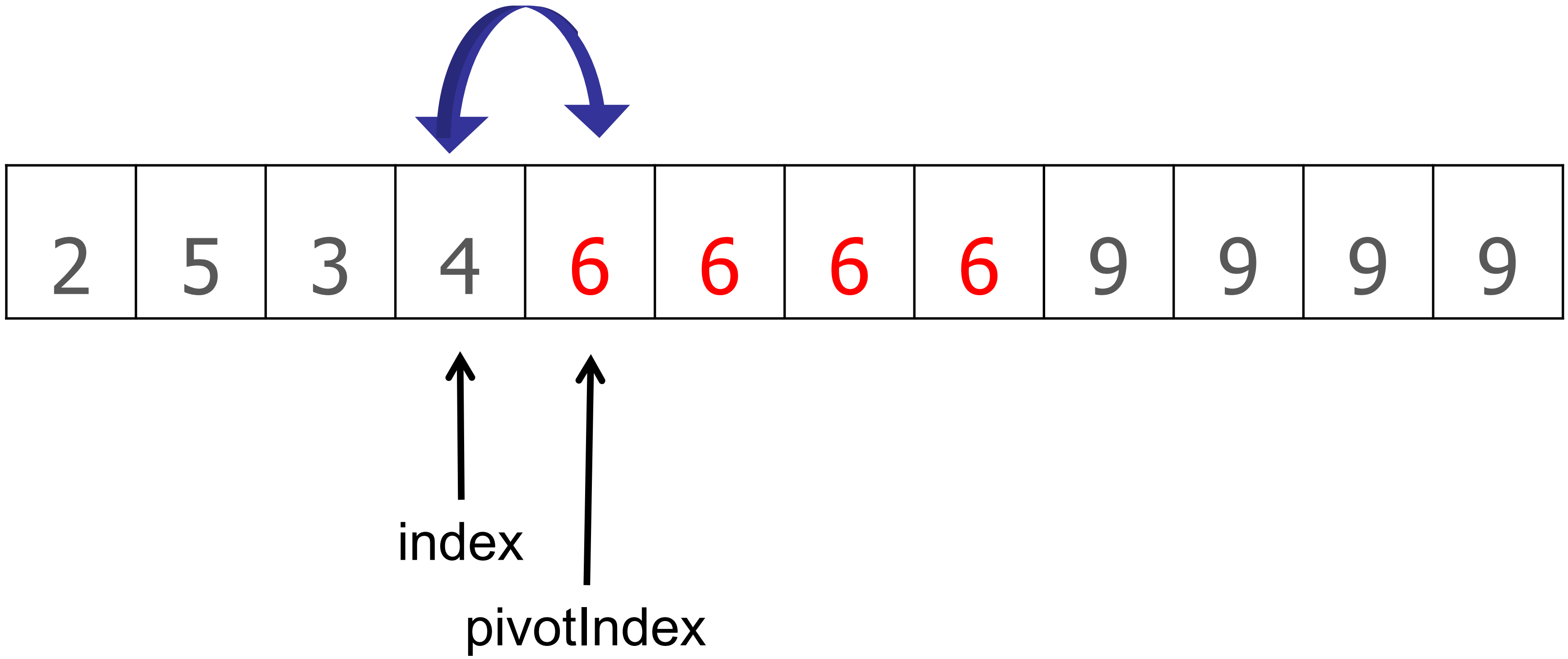
Pack Duplicates

Example:



Pack Duplicates

Example:



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	> x

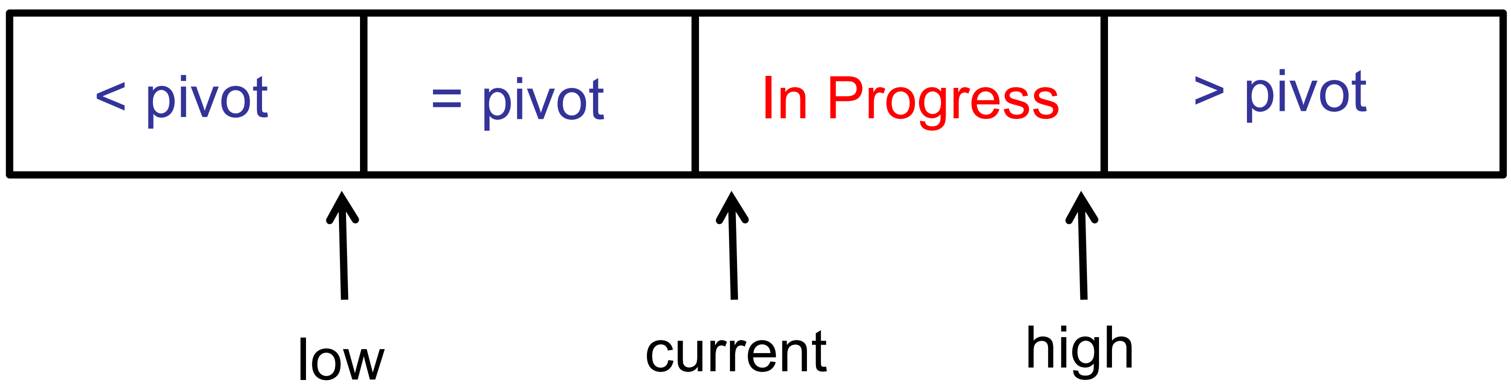
Duplicates

3-Way Partitioning

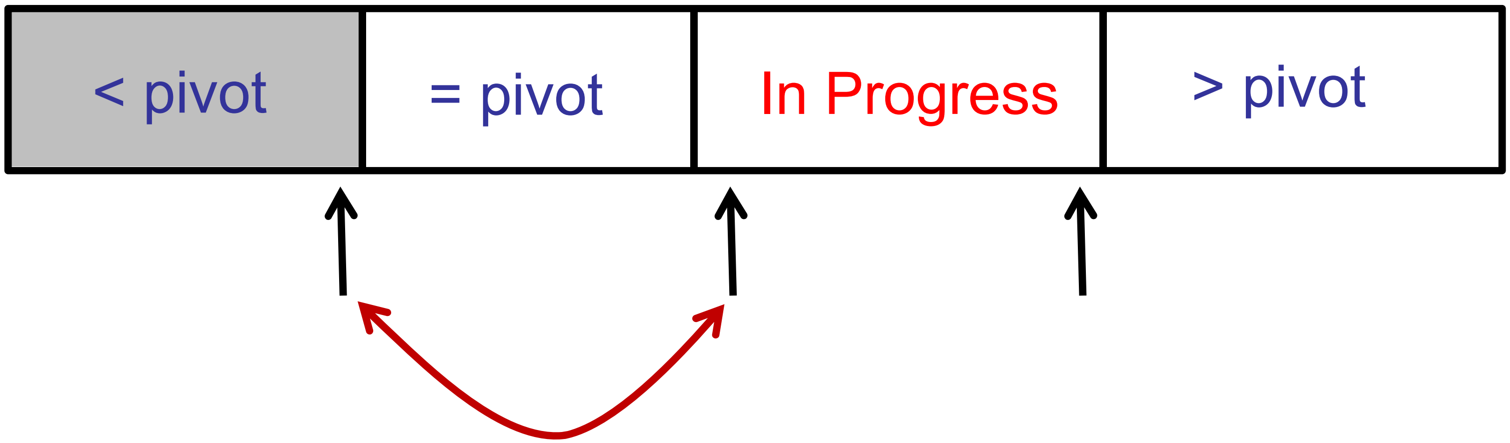
- Option 1: two pass partitioning
  1. Regular partition.
  2. Pack duplicates.

- Option 2: one pass partitioning
  - Standard solution.
  - Maintain four regions of the array

3-Way Partitioning



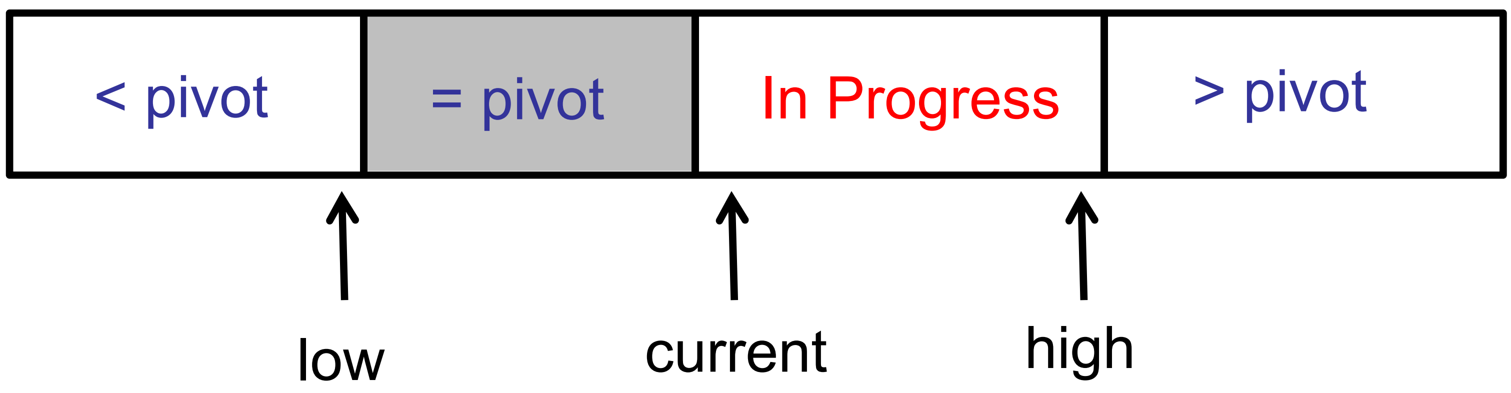
3-Way Partitioning



if A[current] < pivot:

- Increment low
- Swap A[current], A[low]
- Increment current

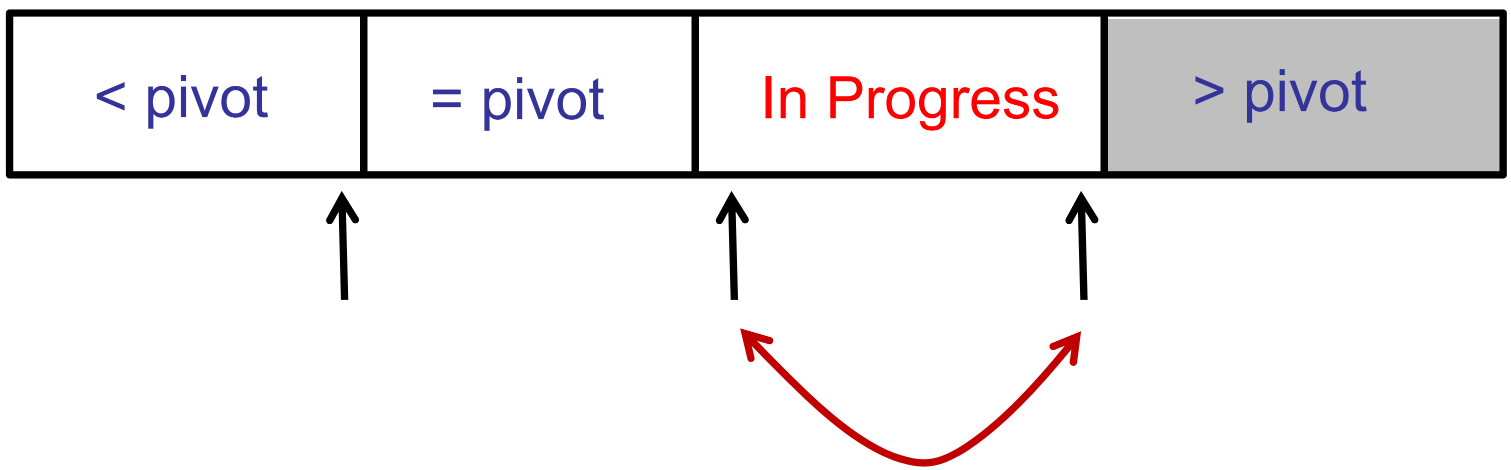
3-Way Partitioning



if A[current] == pivot:

- Increment current

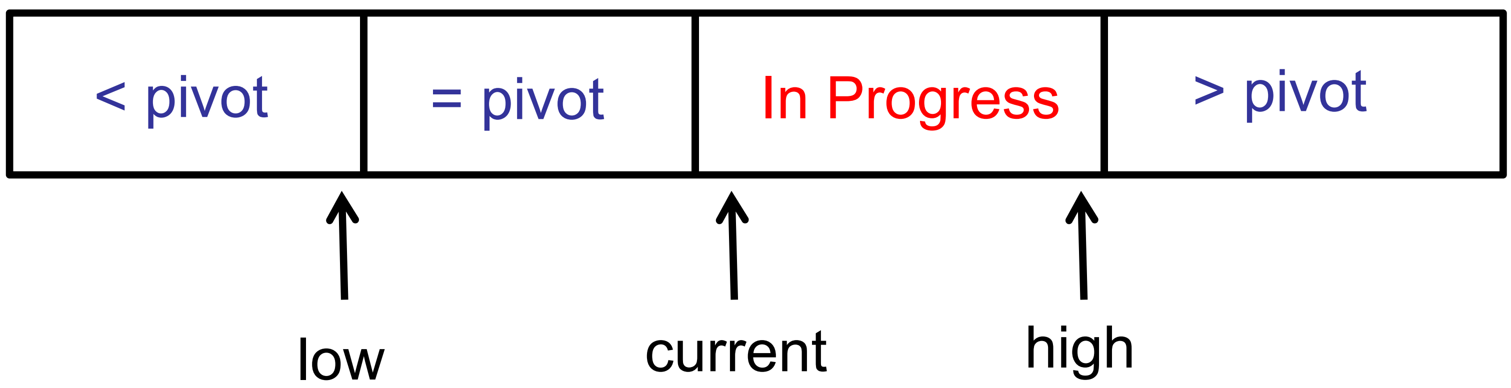
3-Way Partitioning



if A[current] > pivot:

- Swap A[current], A[high]
- Decrement high

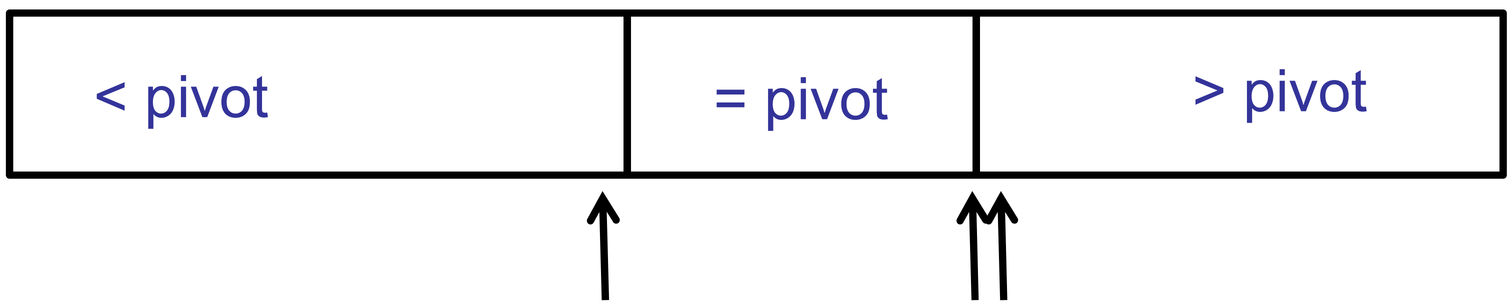
3-Way Partitioning



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;

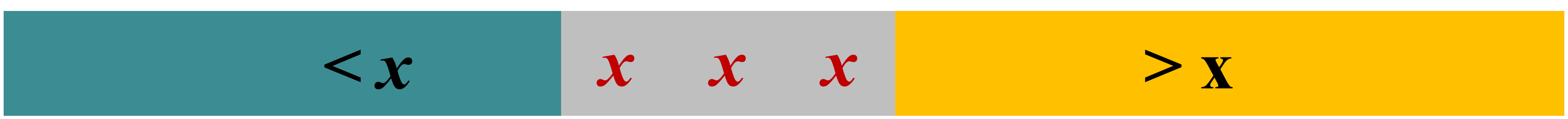
    else

        Choose pivot index *pIndex*.

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

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

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

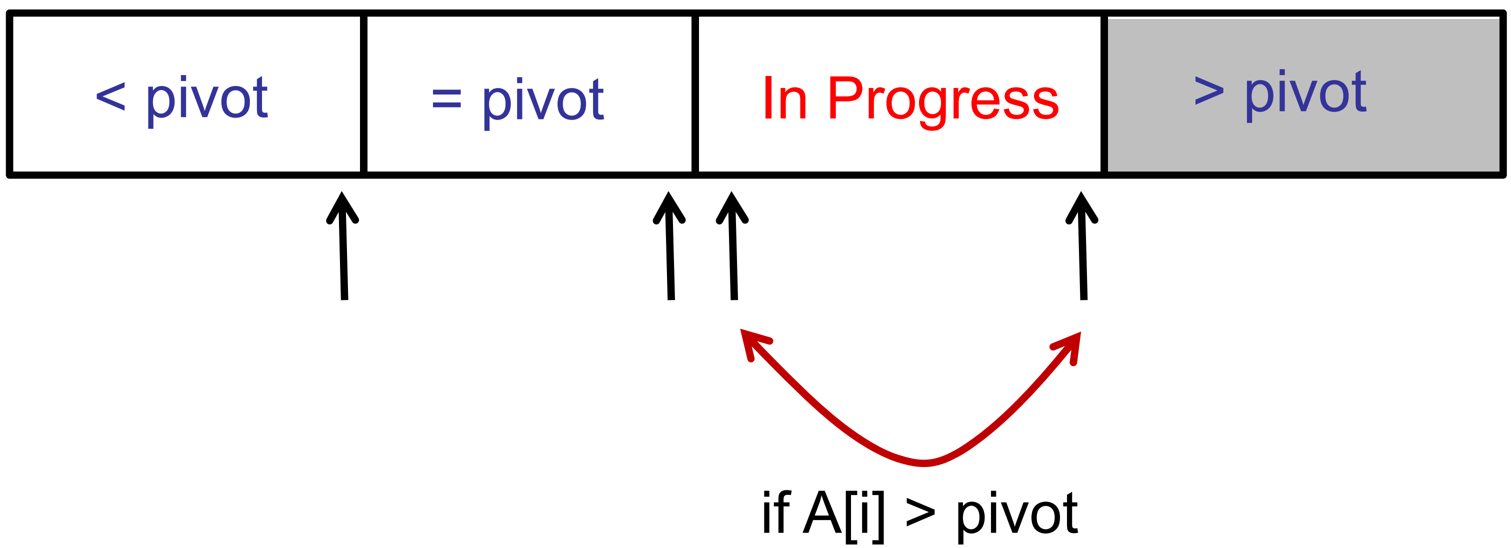


Is QuickSort stable?

ARCHIPELAGO

is open

QuickSort is not stable



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

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

⇩

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

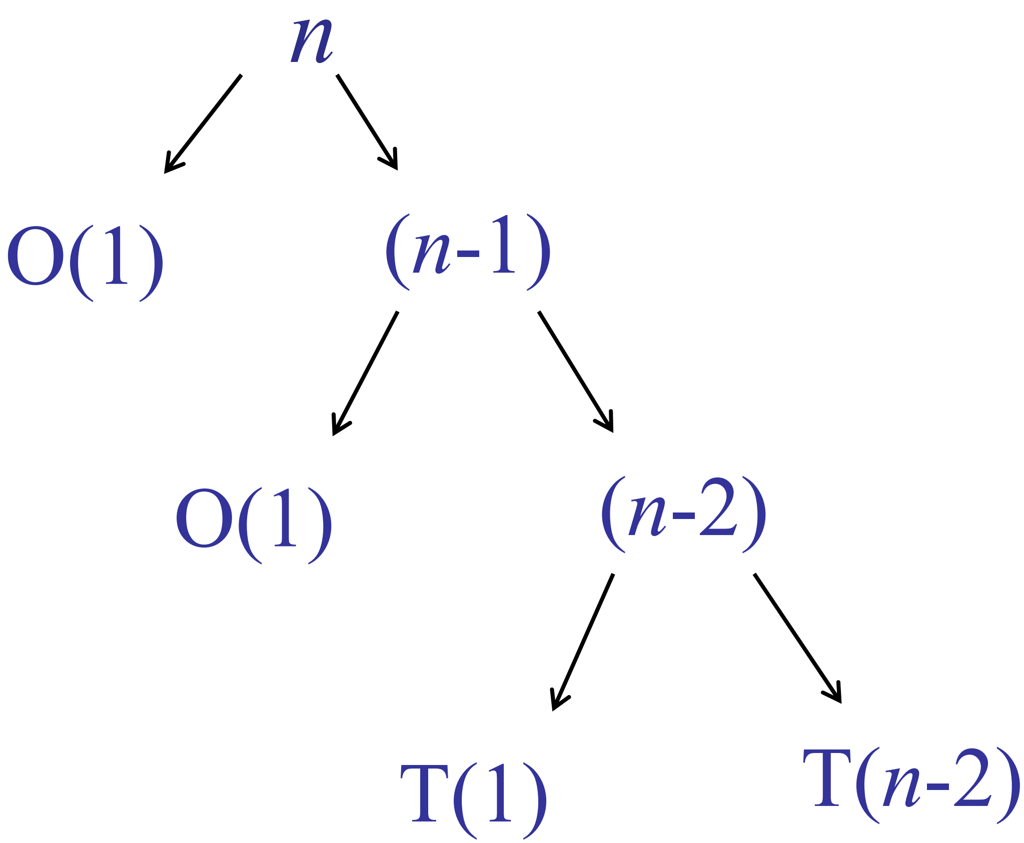
Deterministic QuickSort

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

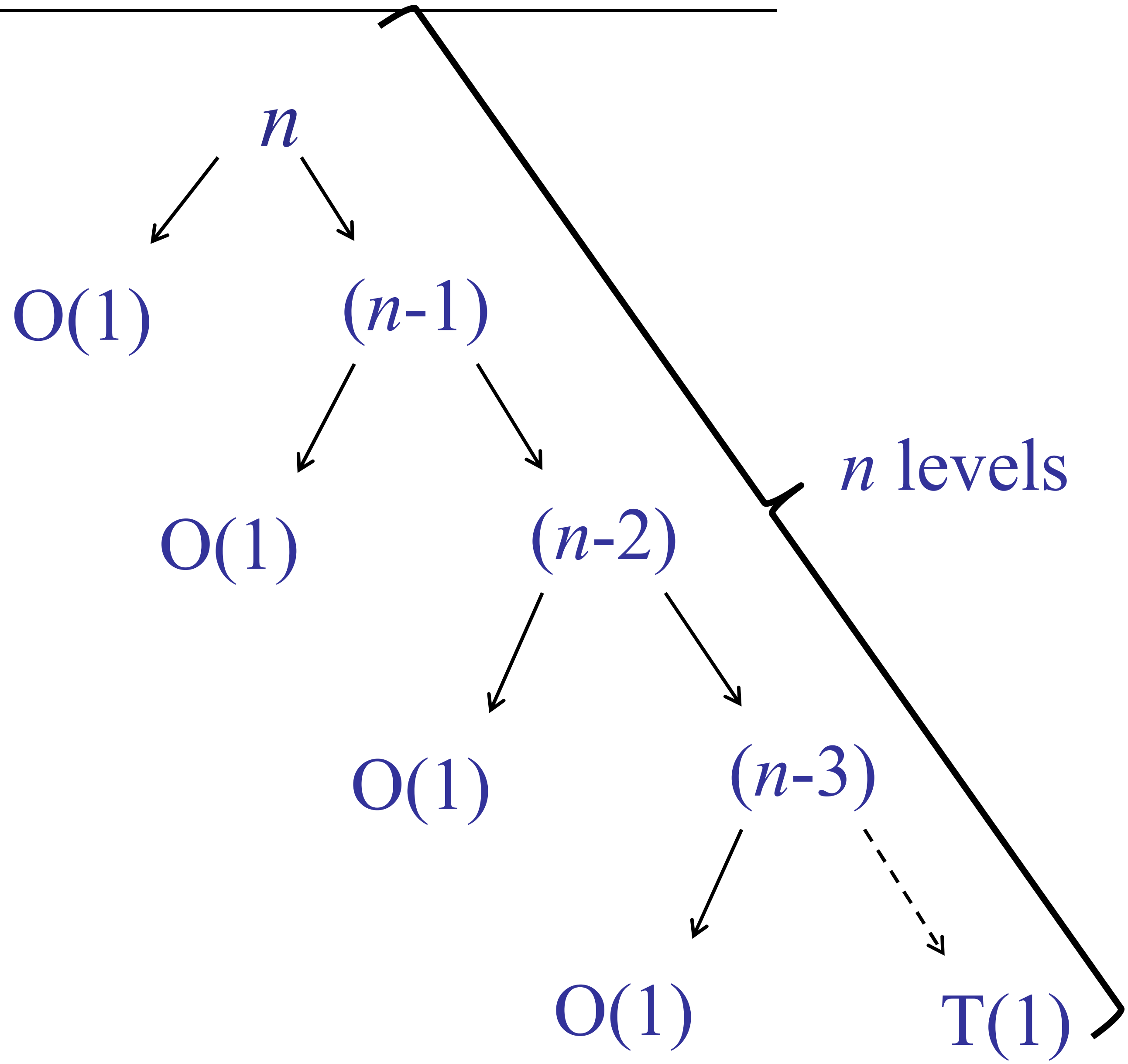
Deterministic QuickSort

$$\begin{tikzcd}
& n \arrow[dl] \arrow[dr] & & \\
O(1) & & (n-1) \arrow[dl] \arrow[dr] & \\
& T(1) & & T(n-2)
\end{tikzcd}$$

Deterministic QuickSort

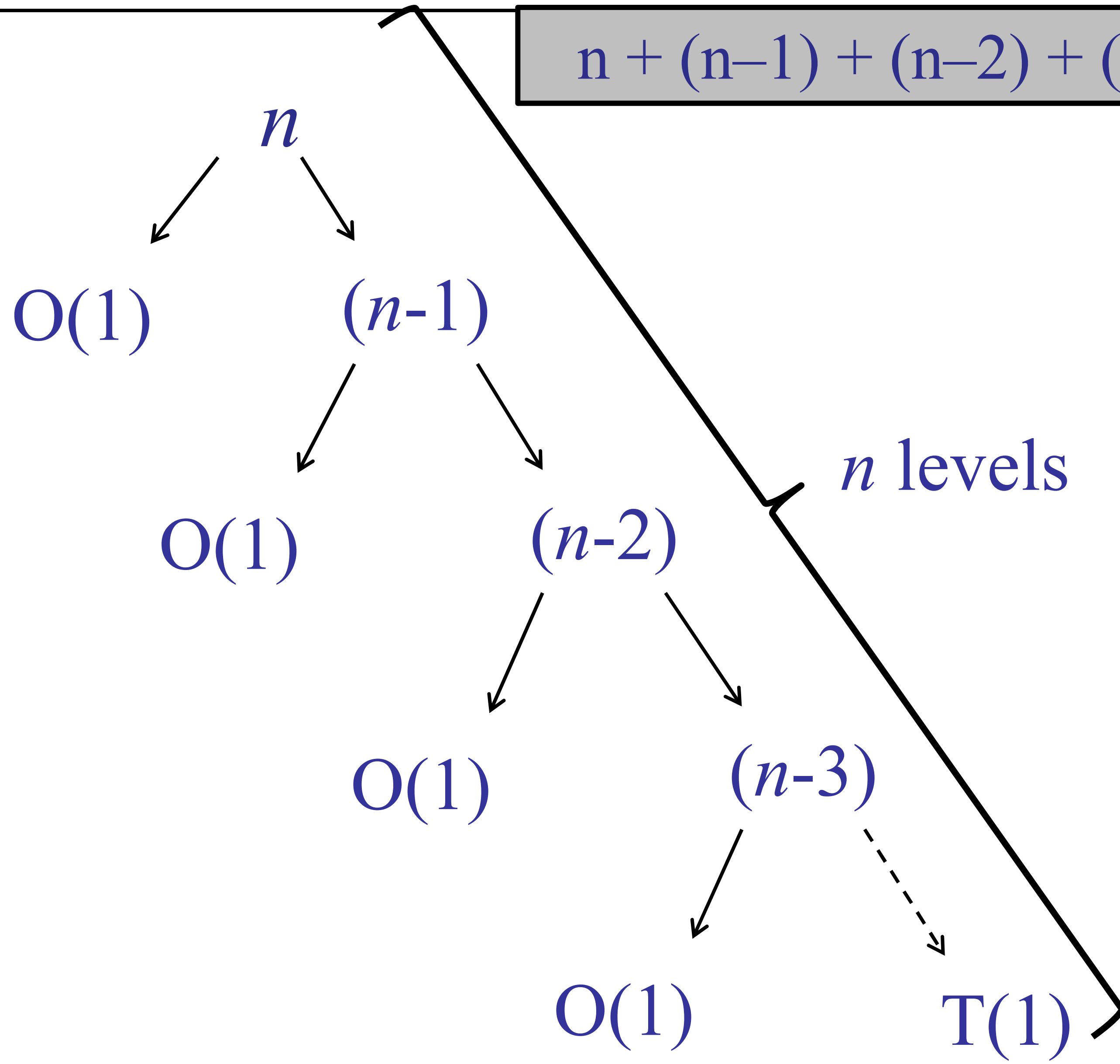


Deterministic QuickSort



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

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:

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

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:

$$T(n) = T(n\text{–}k) + T(k) + n$$

Cost of partition on n elements

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

Assume: $9n/10 > (n - \mathrm{k}) > n/10$

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:

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

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*)

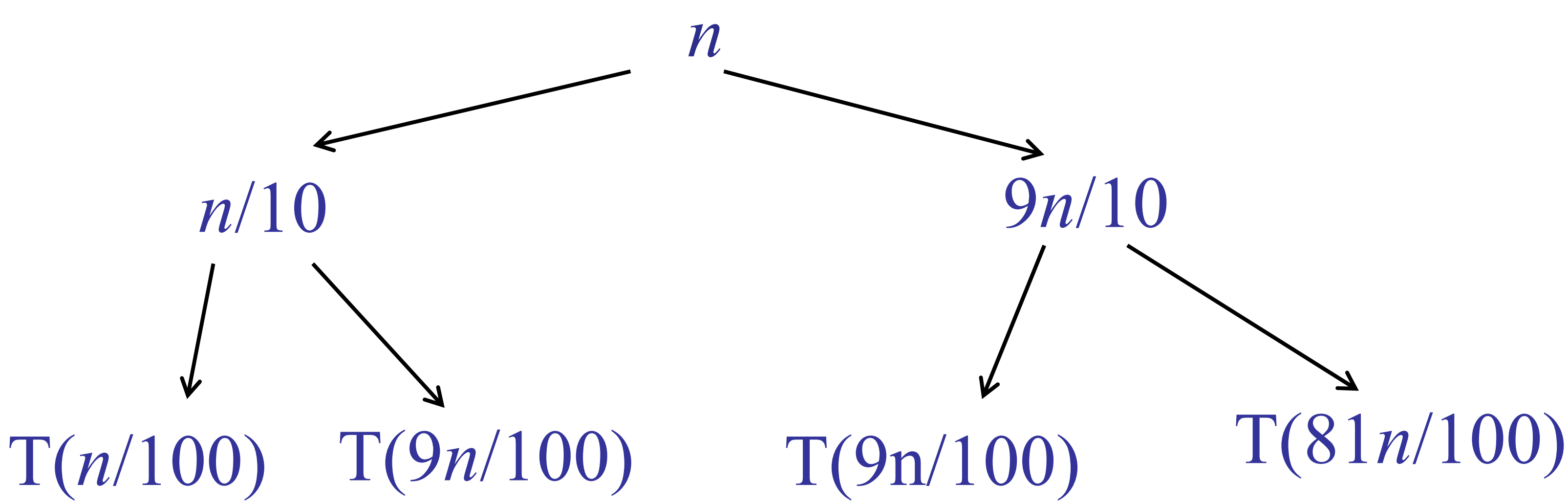
QuickSort Analysis

$k = n/10$

$$
\begin{array}{ccc}
 & n & \\
\swarrow & & \searrow \\
T(n/10) & & T(9n/10)
\end{array}
$$

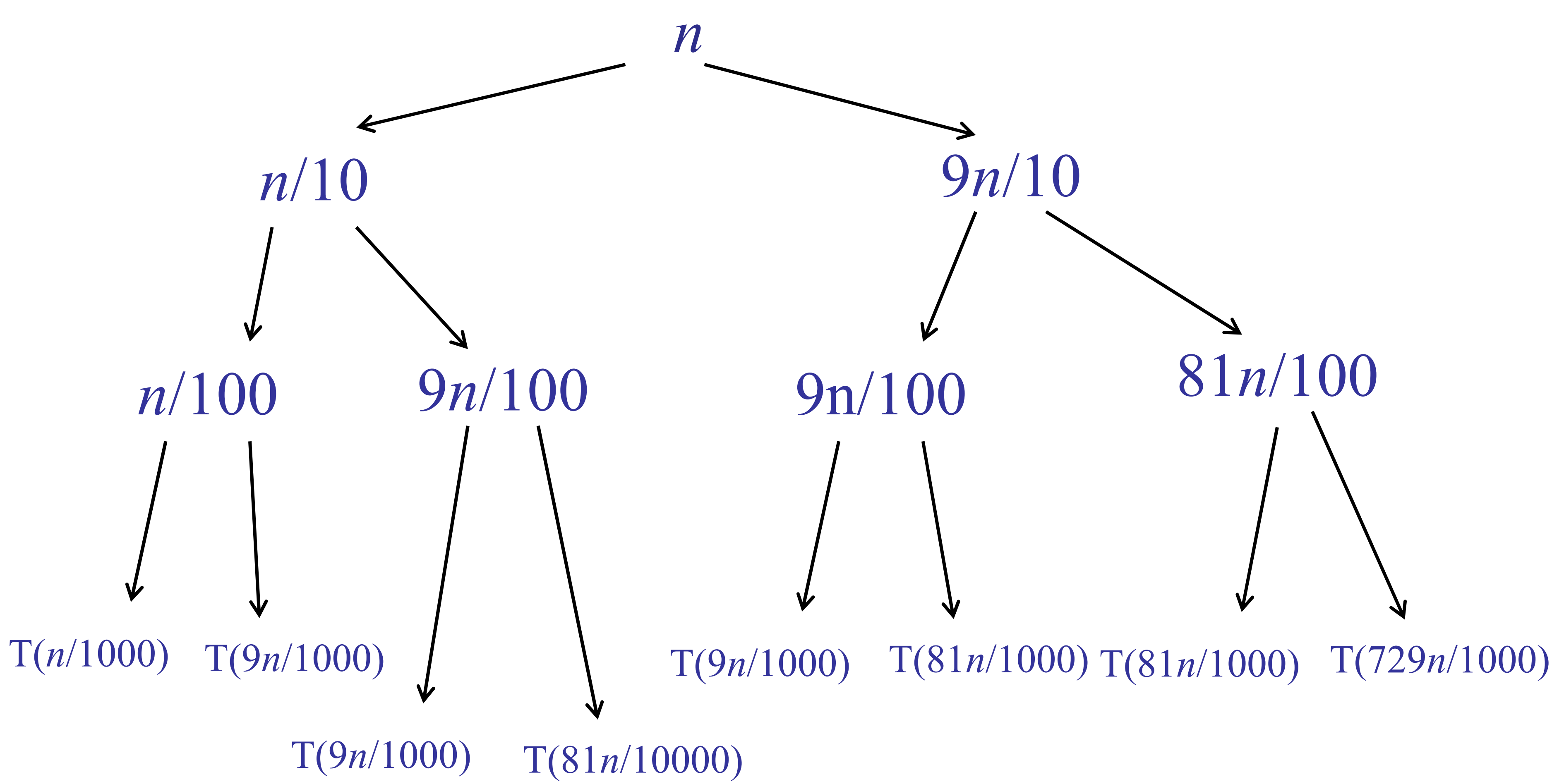
QuickSort Analysis

$k = n/10$



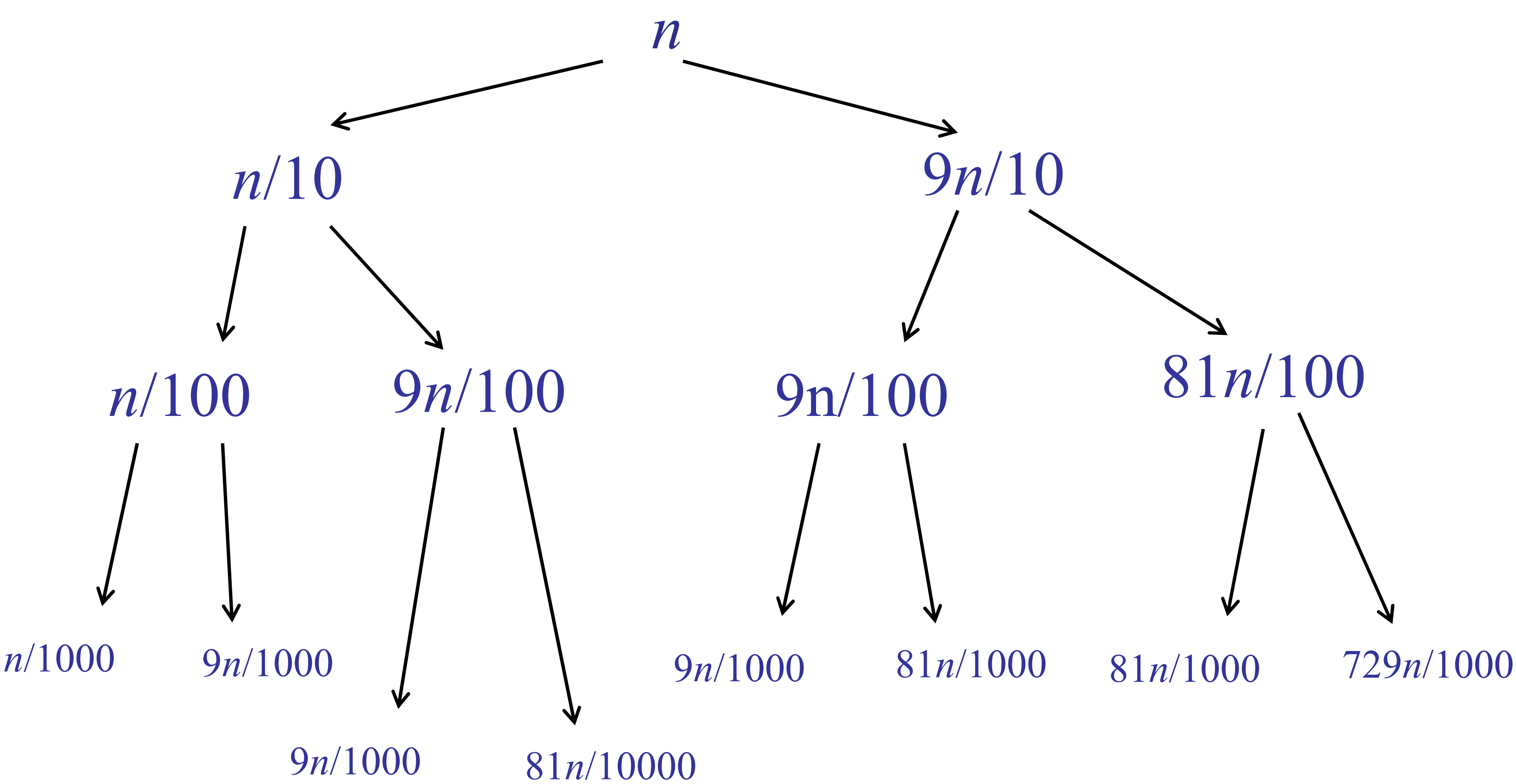
QuickSort Analysis

$k = n/10$

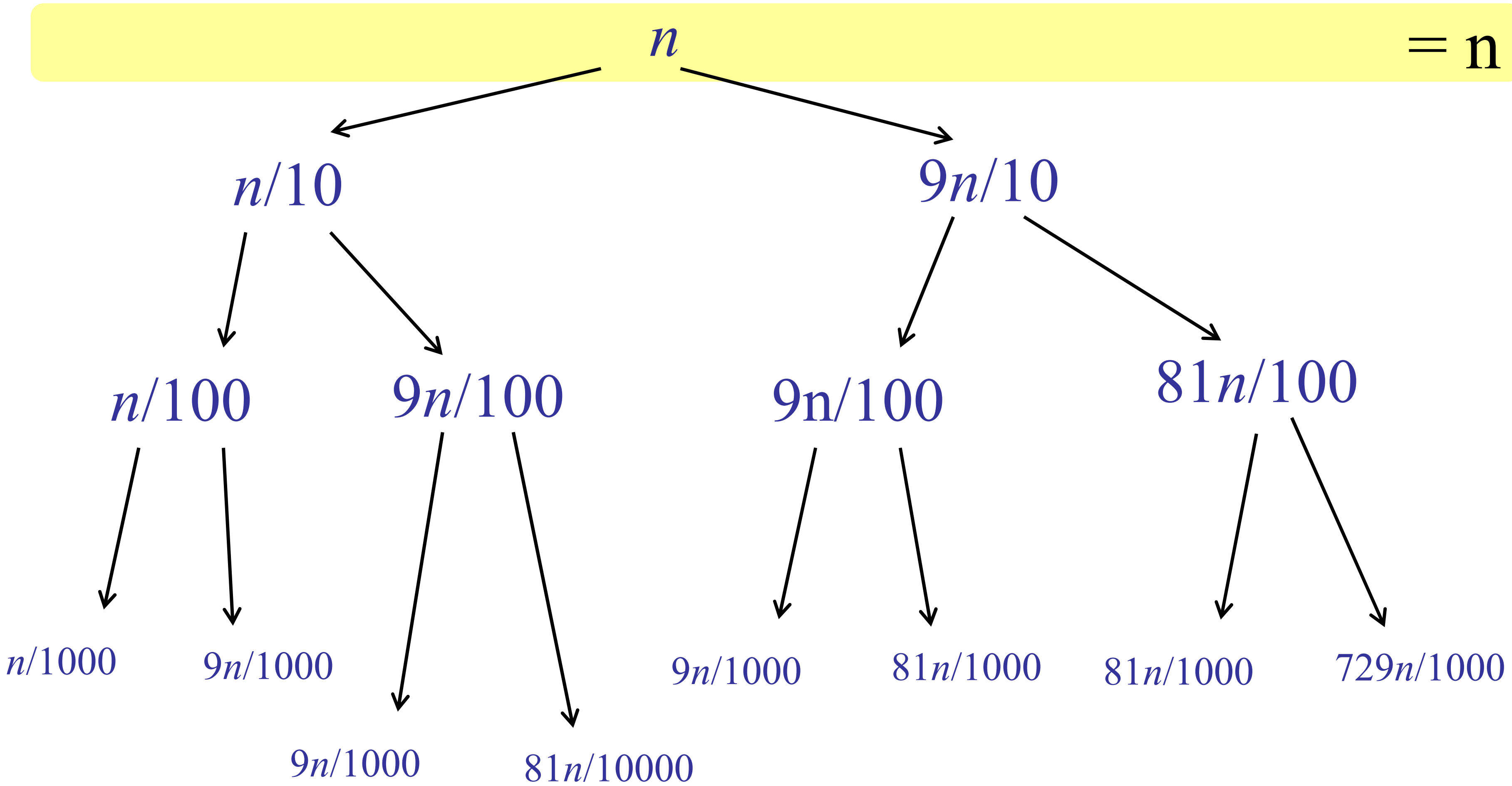


QuickSort Analysis

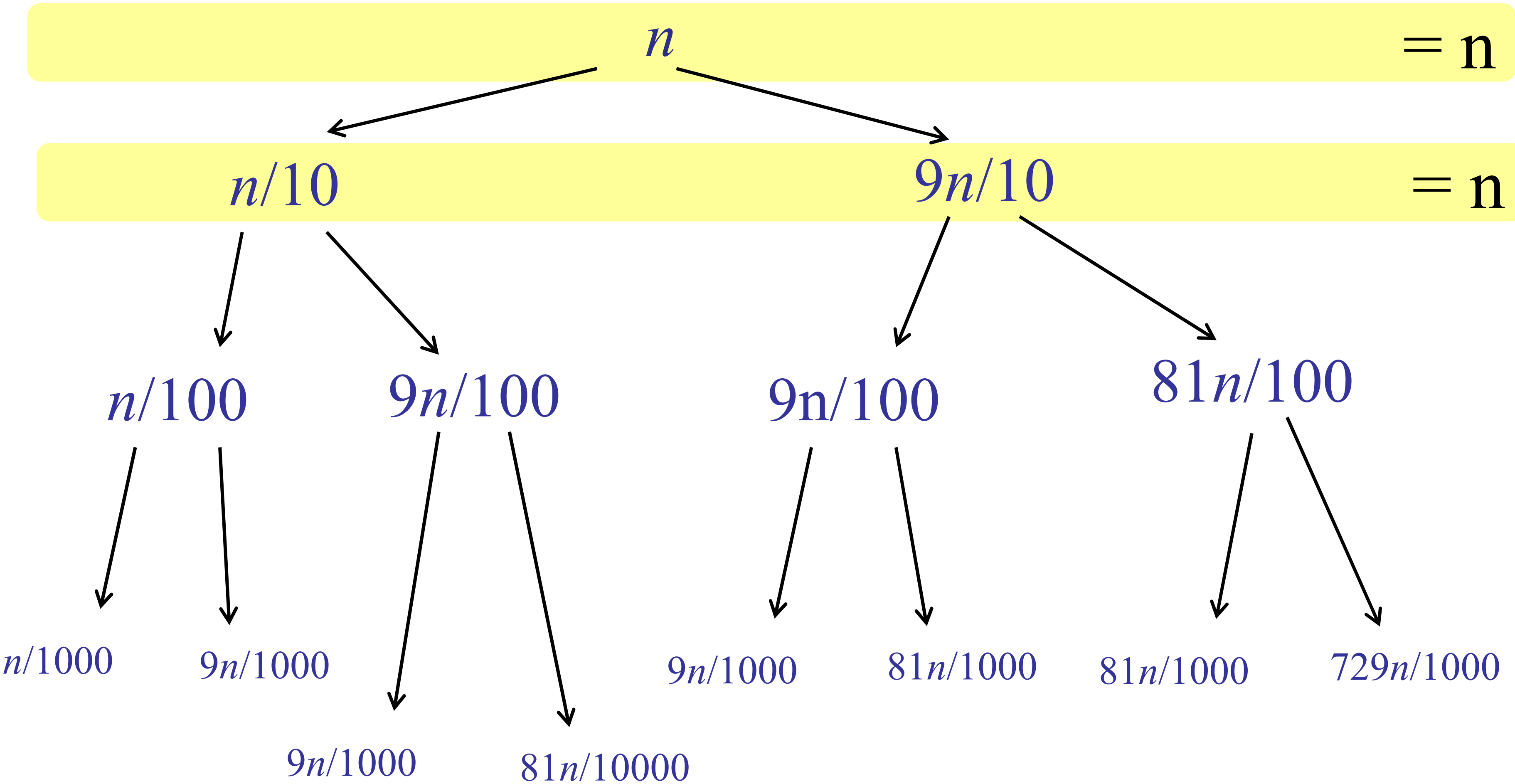
$k = n/10$



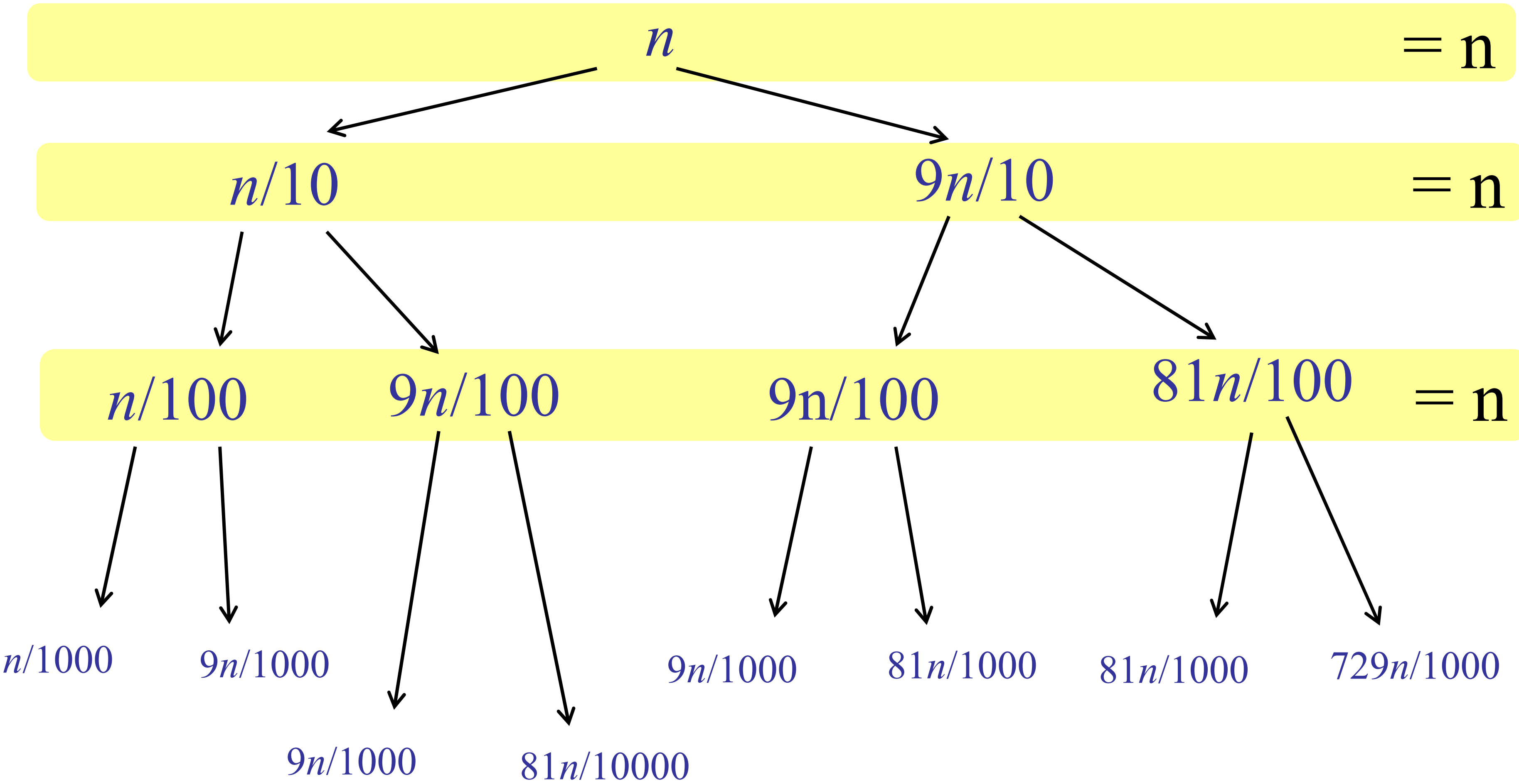
QuickSort Analysis



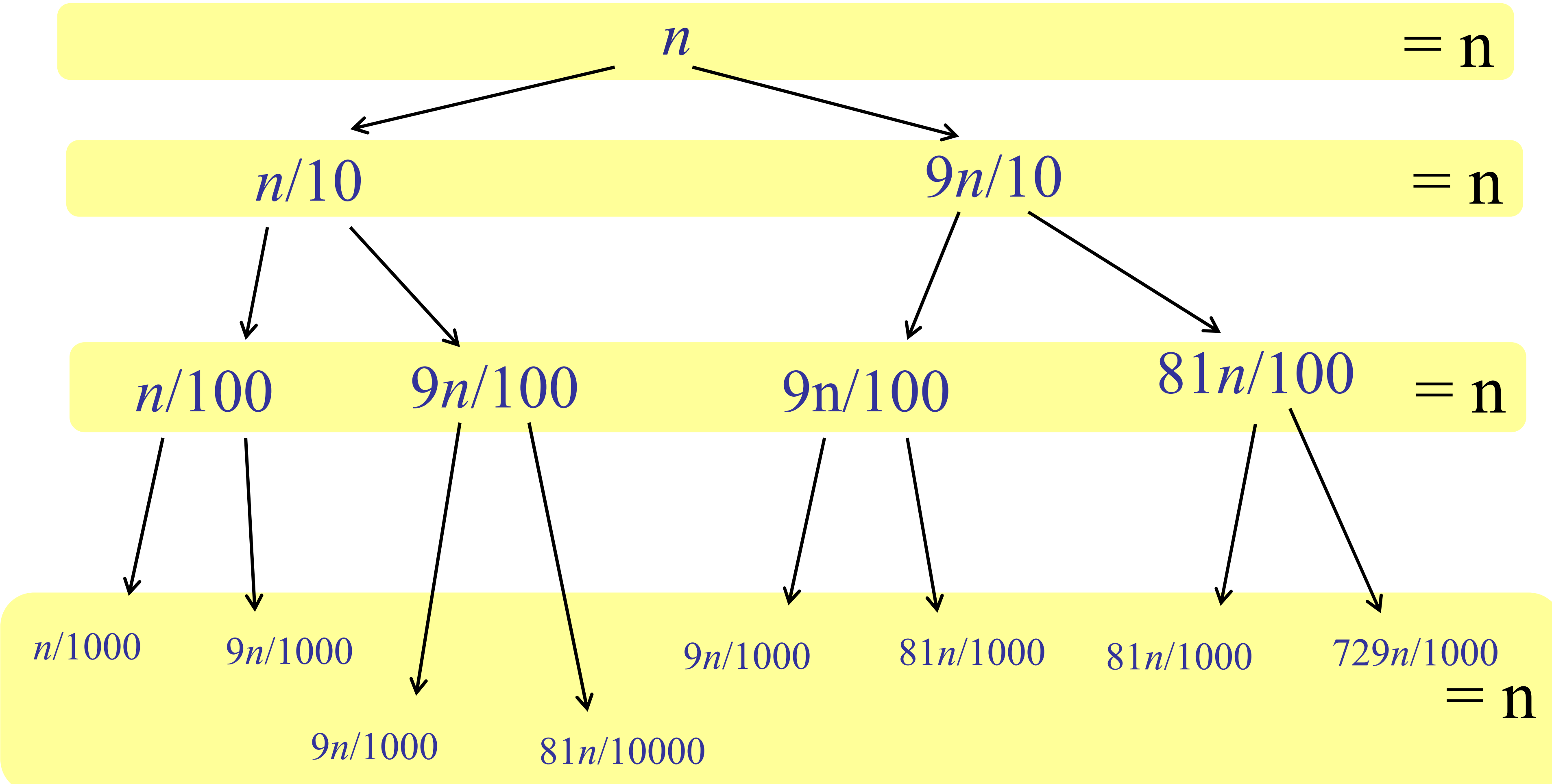
QuickSort Analysis



QuickSort Analysis

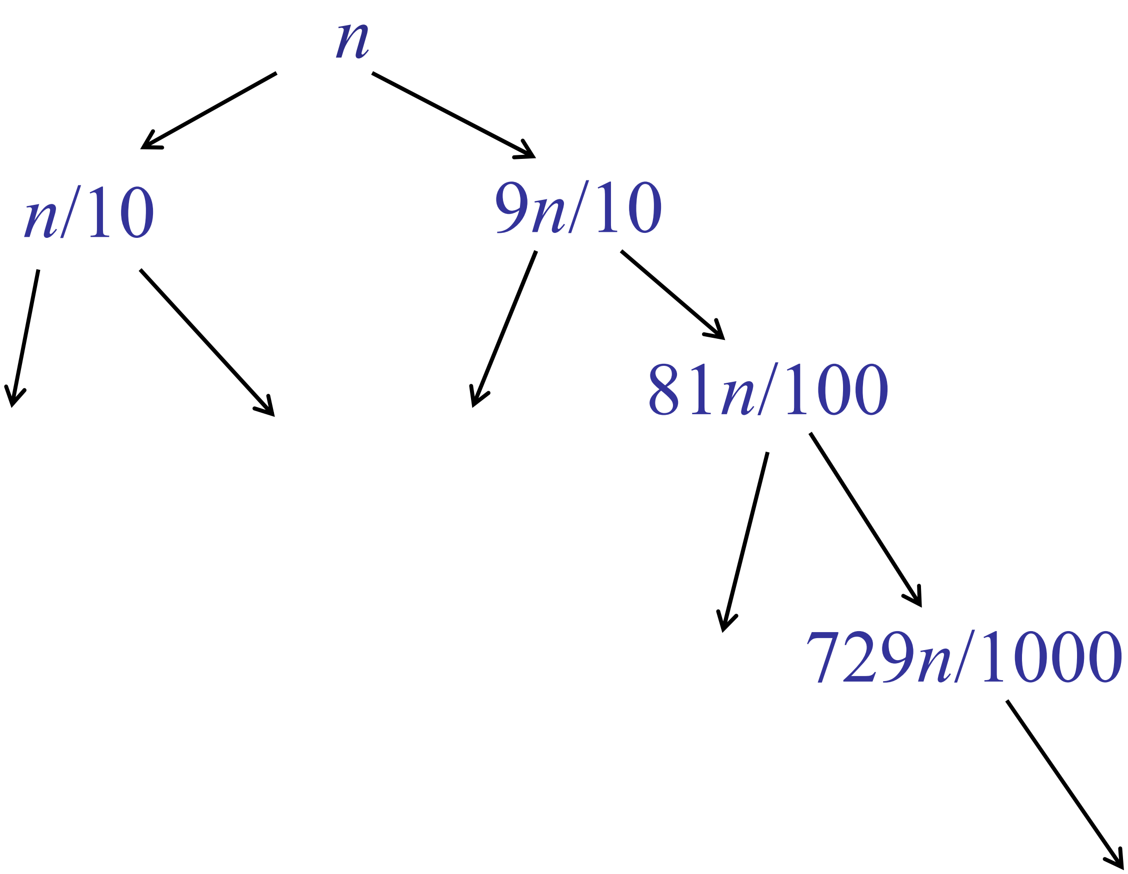


QuickSort Analysis



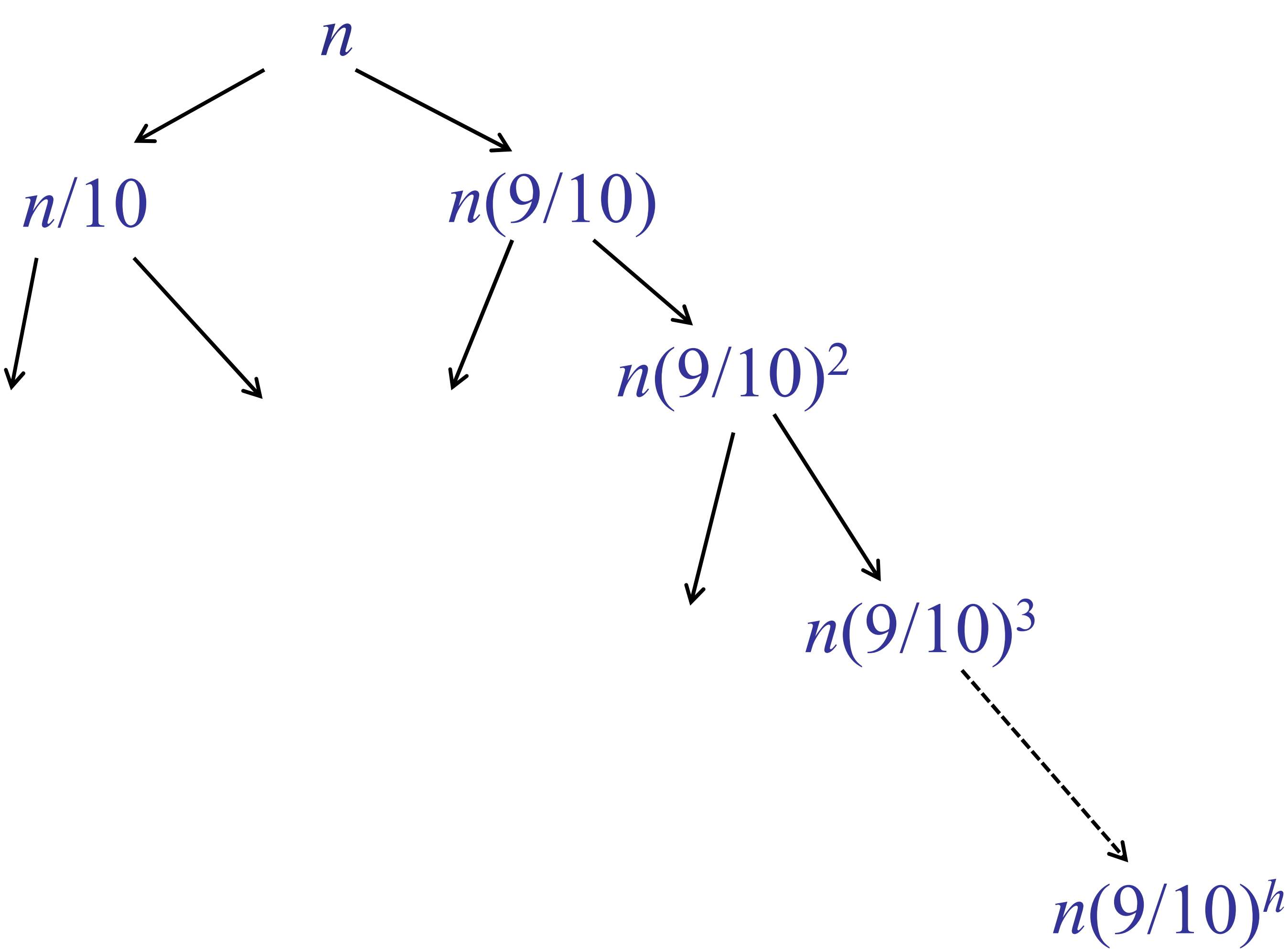
How many levels??

QuickSort Analysis



How many levels??

QuickSort Analysis



How many levels??

QuickSort Analysis

Maximum number of levels:

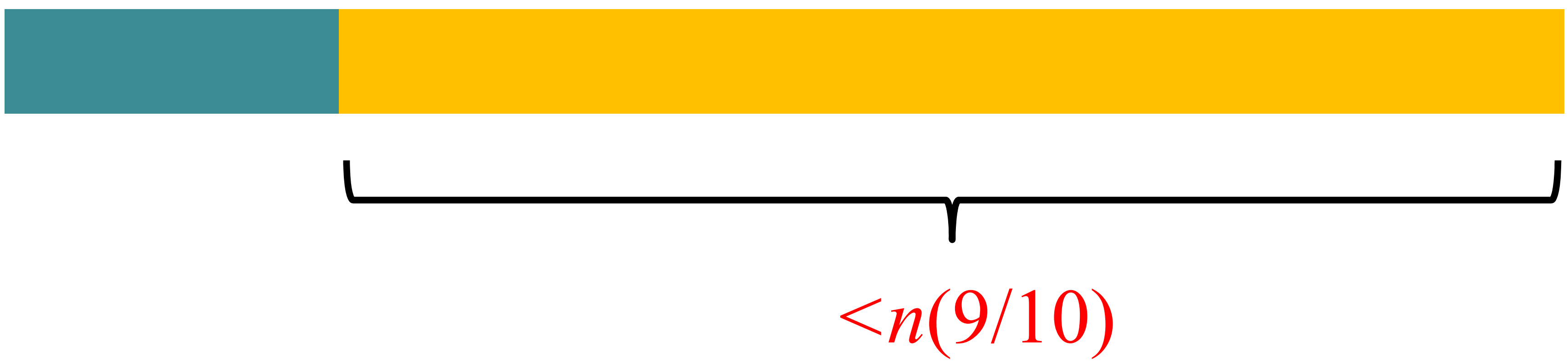
$$1 = n(9/10)^h$$

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

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

QuickSort Analysis

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



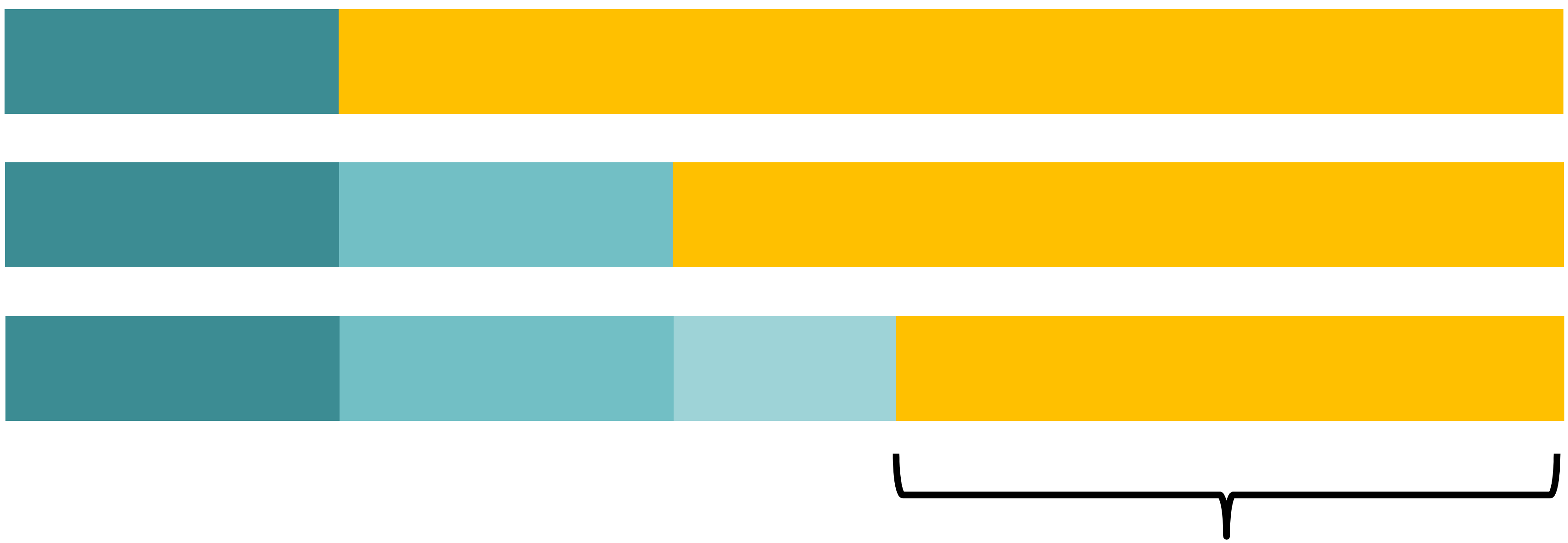
QuickSort Analysis

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

$<n(9/10)^2$

QuickSort Analysis

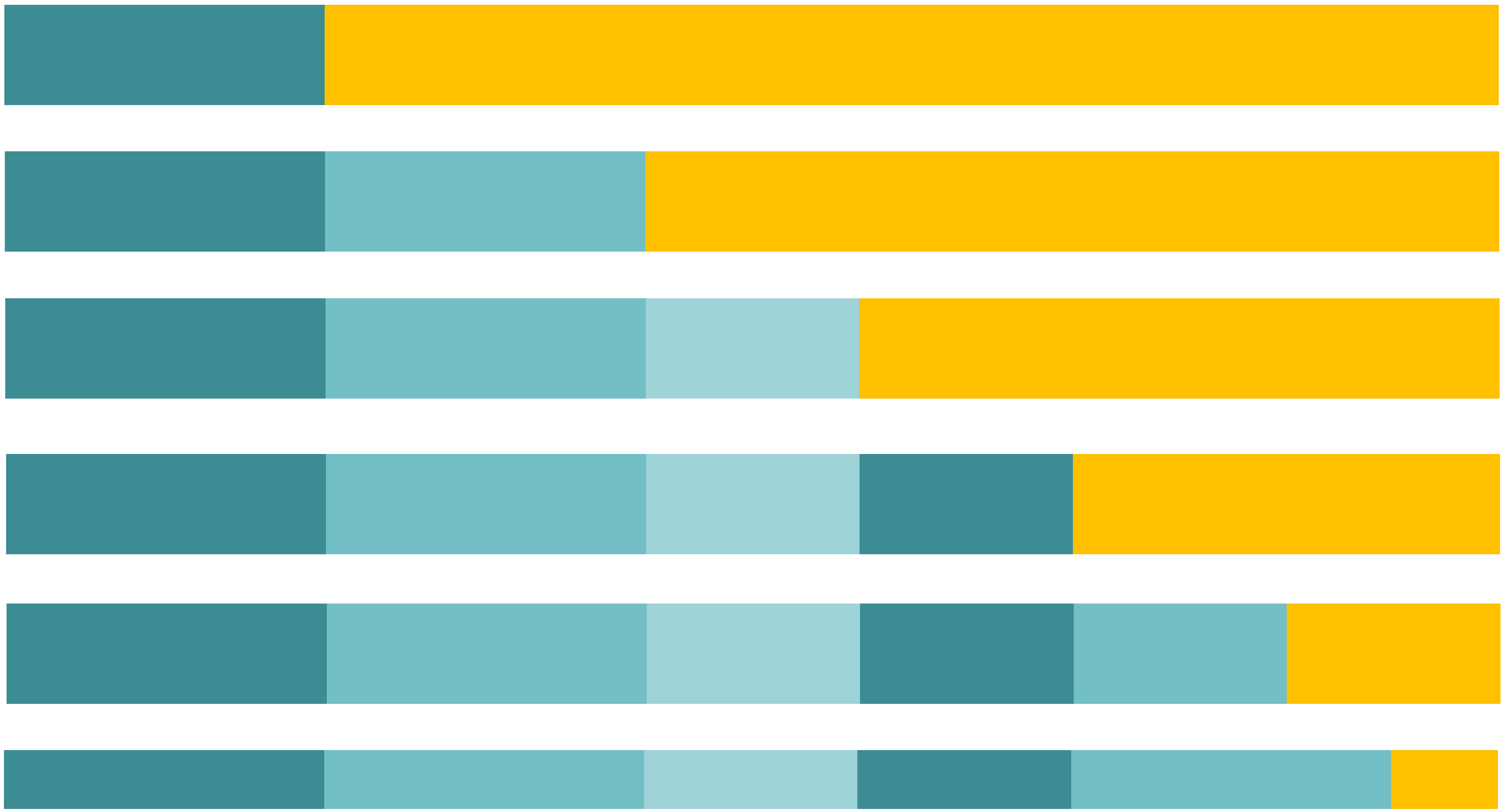
Assume larger part shrinks by at least 9/10 every iteration:



$< n(9/10)^3$

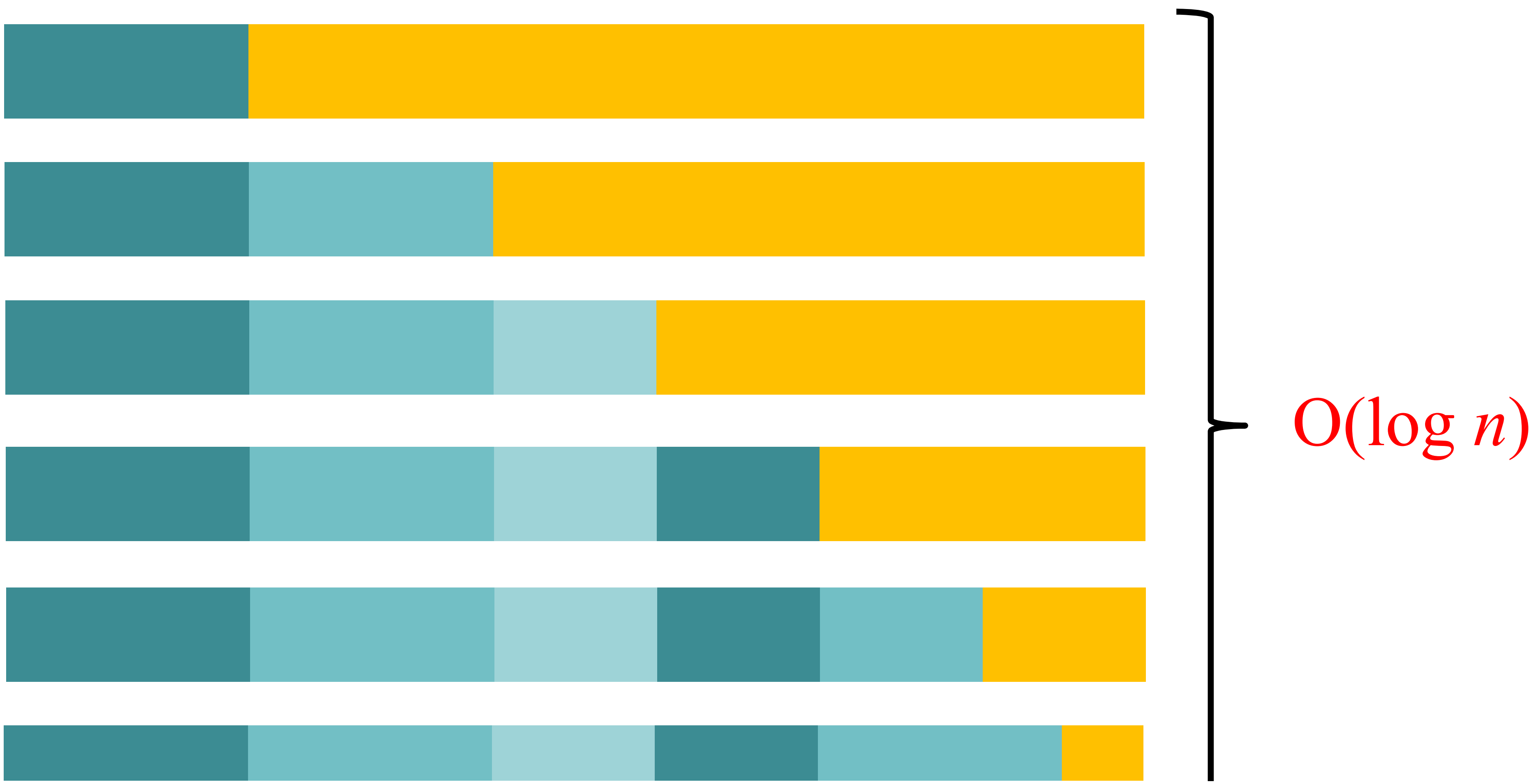
QuickSort Analysis

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

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

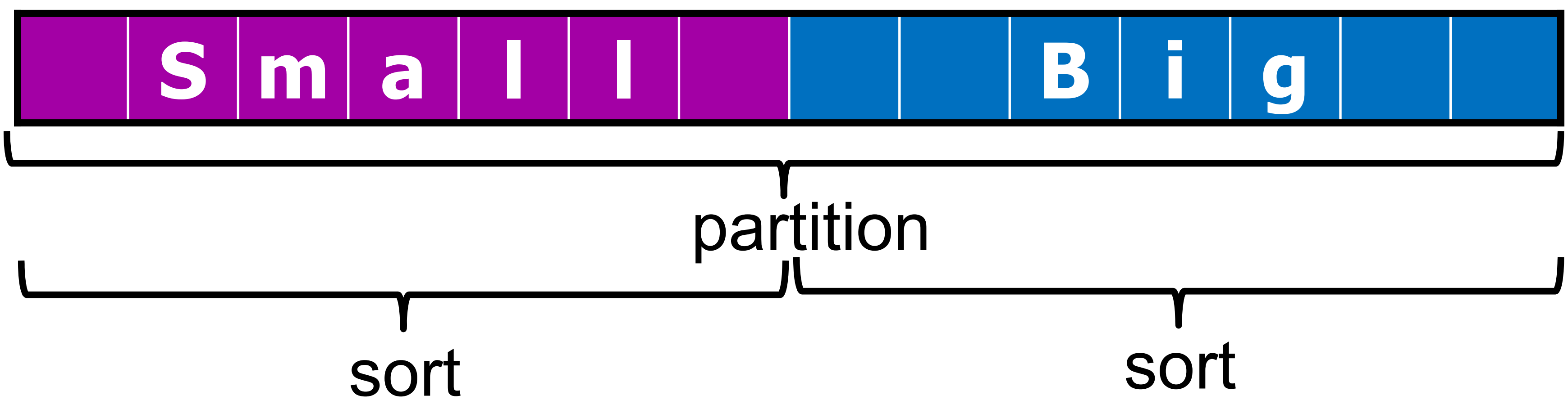
Randomized algorithm:

- 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)

1

??

T(k)

T(n-k)

```
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
```

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

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)

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
```

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. 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):

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

Independent Events

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

- Given: Pr(**A**), Pr(**B**)
- Given: **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?

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(**A = 12**) = ¼
- Pr(**A = 60**) = ¾

Expected value of A:

$E[A] = (¼)12 + (¾)60 = 48$

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

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 expect 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

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(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

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?

E[X]= p(1) +

(1 – p)(p)(2) +

(1 – p)(1 – p)(p)(3) +

(1 – p)(1 – p)(1 – p) (p)(4) +

…

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:

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

How many flips to get at least one head?

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

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

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(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:

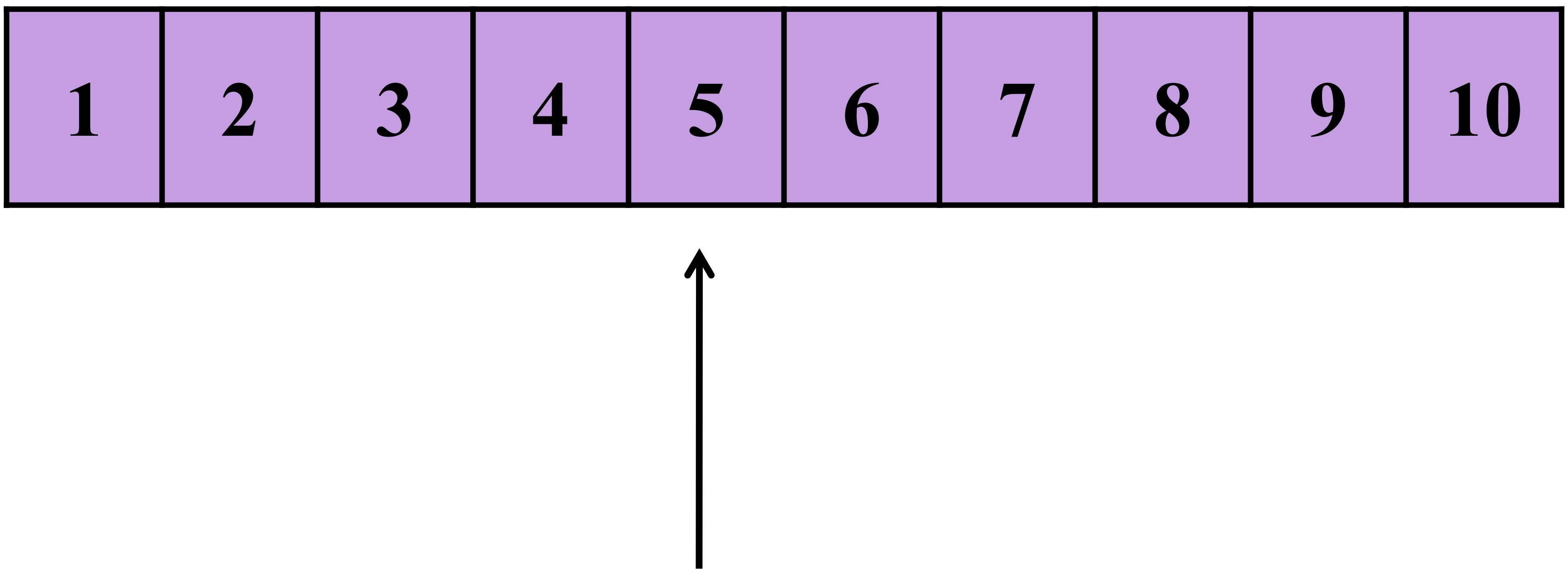
A *pivot* is <u>**good**</u> if it divides the array into two pieces, each of which is size at least $n/10$.

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

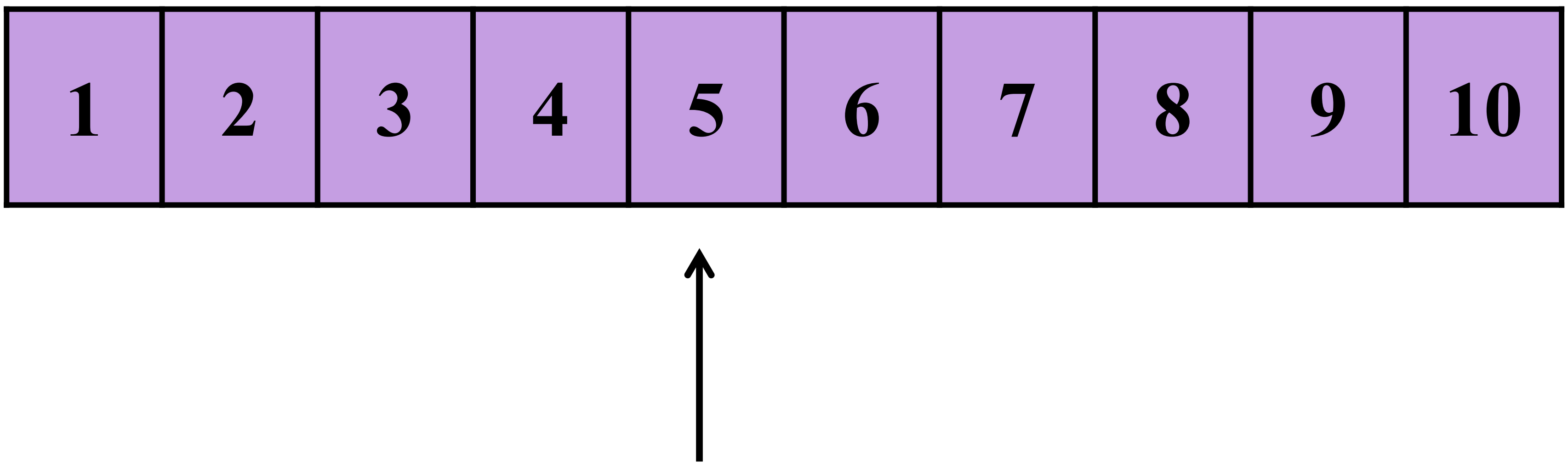
Imagine the array divided into 10 pieces:



Choose a random point at which to partition.

Choosing a Good Pivot

Imagine the array divided into 10 pieces:

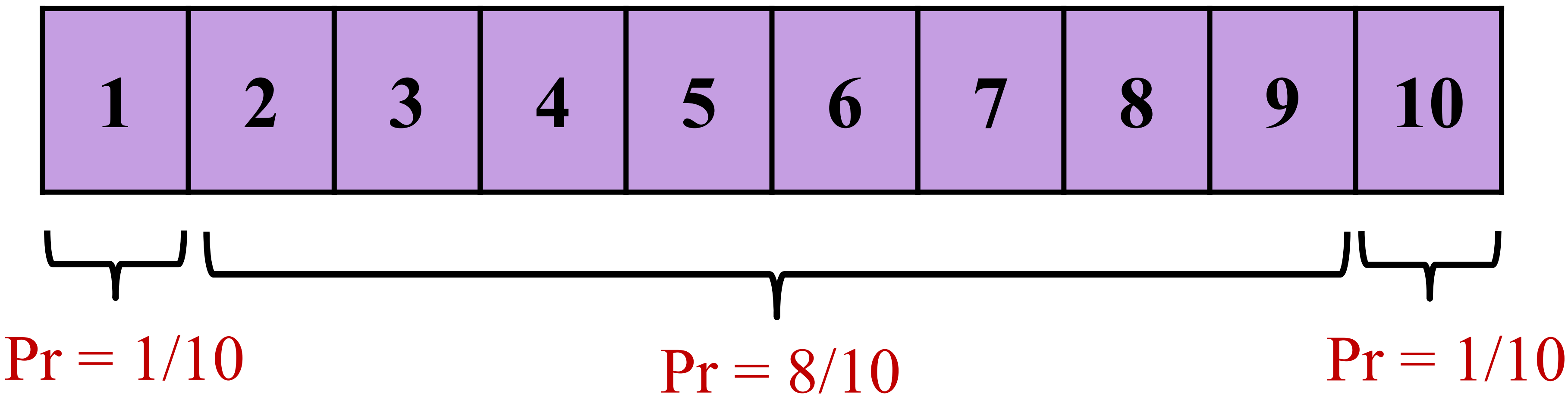


Choose a random point at which to partition.

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

Choosing a Good Pivot

Imagine the array divided into 10 pieces:

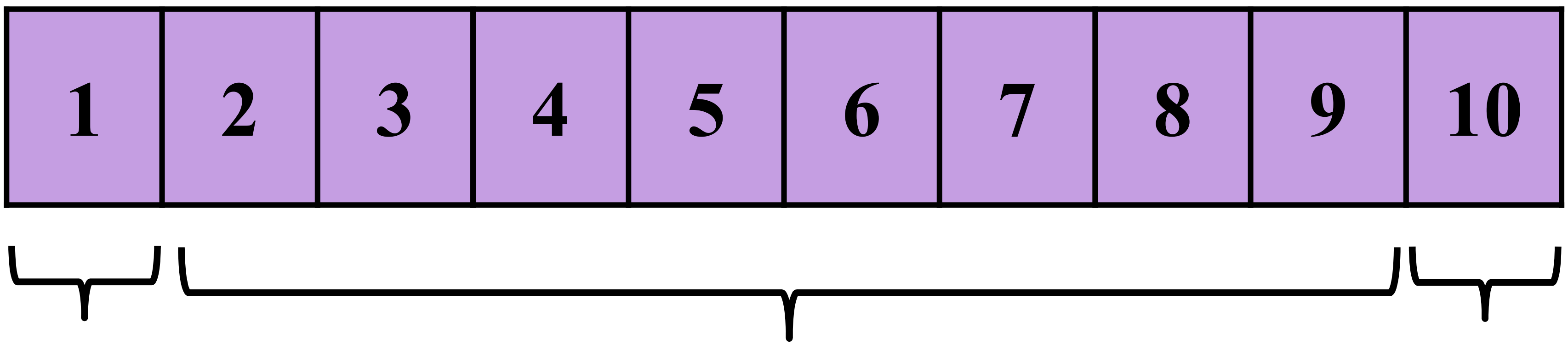


Choose a random point at which to partition.

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

Choosing a Good Pivot

Imagine the array divided into 10 pieces:



Pr = 1/10   Pr = 8/10   Pr = 1/10

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

Expected number of iterations: 10/8

```
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 < 2 times

(in expectation).

Then we know:

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

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

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

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: Θ(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 partioning 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:

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

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

Find the 7th 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

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

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 5th element

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

Order Statistics

Example: search for 5^{th} 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 5th element

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

Search for 5th element in left half.

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

Order Statistics

Example: search for 5th 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^{th} element

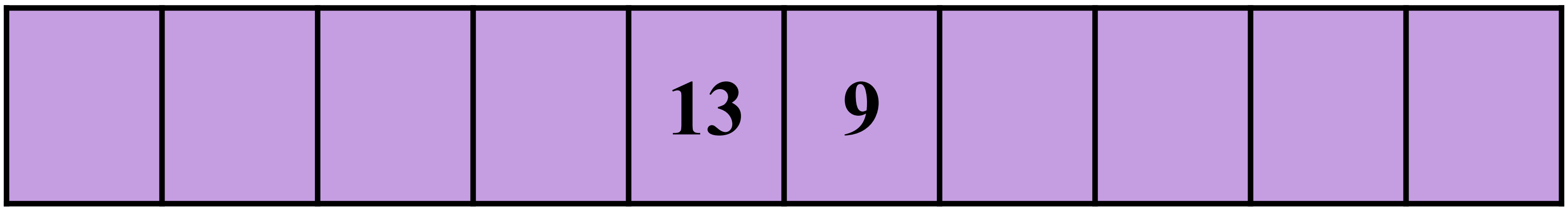
9	**8**	**13**	**5**	**3**	**6**				

Search for: 5 – 4 = 1 in right half

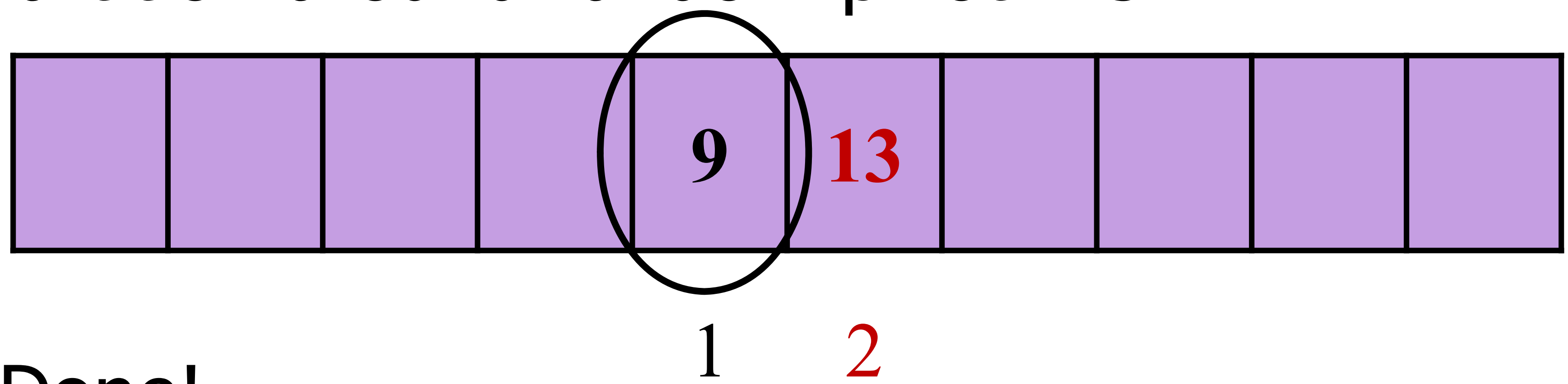
6	**3**	**5**	**8**	**13**	**9**				
1	2	3	4	5	6				

Order Statistics

Search for: 5 – 4 = 1 in right half



Partition around random pivot: 13



Done!

Finding the k^{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

Recursing right and left are not exactly the same.

Example: search for 5th 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 5th element on the left.

Order Statistics

Recursing right and left are not exactly the same.

Example: search for 8^{th} 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^{th} element on the right.

Order Statistics

Recursing right and left are not exactly the same.

Example: search for 4th 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^{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^{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 = partition(A[1..n], n, pIndex)
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:

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

Analysis

Paranoid-Select:

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

Recurrence:

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

$$\le 2n + \mathbf{E}[\mathrm{T(9n/10)}]$$

$$\le 2n + 2n\,(9/10) + (9/10)\,\mathbf{E}[\mathrm{T(9n/10)}]$$

$$\le 2n + 2n\,(9/10) + 2n\,(9/10)^2 + \ldots\ldots$$

Analysis

Paranoid-Select:

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

Recurrence:

$$\mathbf{E}[T(n)] \le \mathbf{E}[T(9n/10)] + \mathbf{E}[\#\text{ partitions}](n)$$
$$\le \mathbf{E}[T(9n/10)] + 2n$$
$$\le O(n)$$

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^{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^{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!