CS2020
Data Structures and Algorithms

Welcome!

Problem Set

Problem Set 3:

- Extension: due Monday, Jan. 7 --- midnight
- More clarifications / updates posted today.
- Any questions: ask!

Happy new year!

Upcoming…

This week: Lunar New Year

- No Friday lecture or recitation
- No discussion groups

Next week: Quiz 1

- Friday, in class
- Unexcused absences: no makeup quiz
- Excused absences: oral exam

Happy new year!

Today's Plan

QuickSort

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

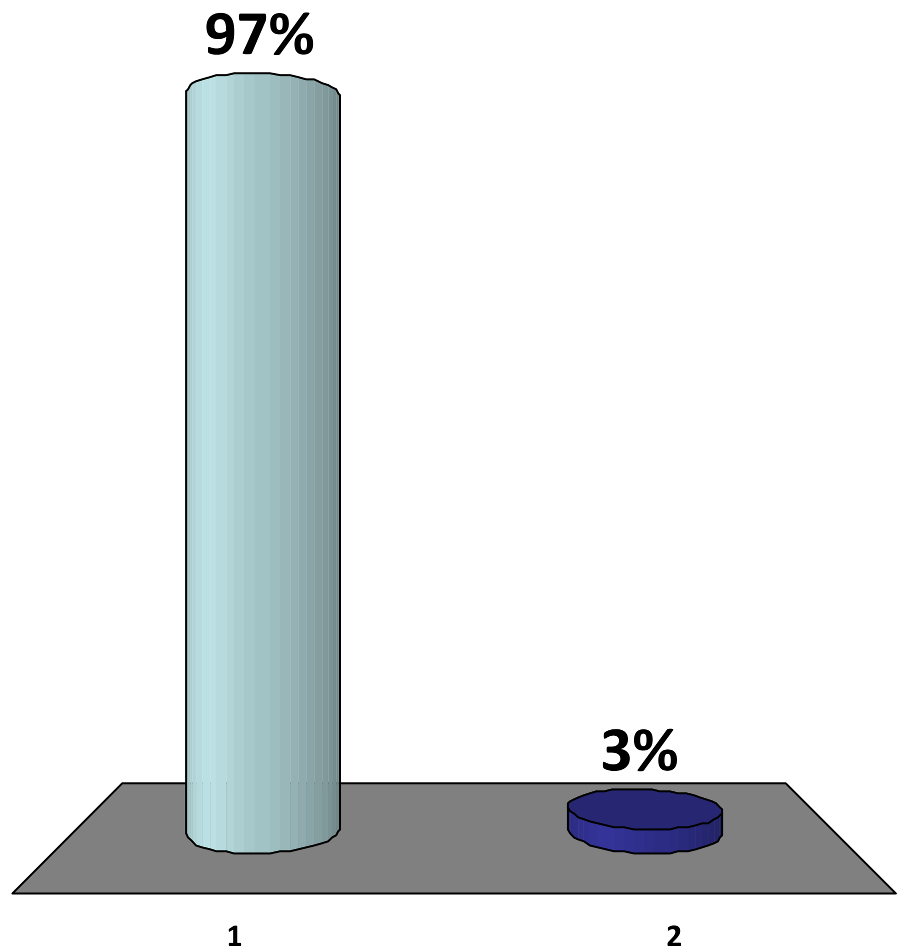
QuickSort

History:

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

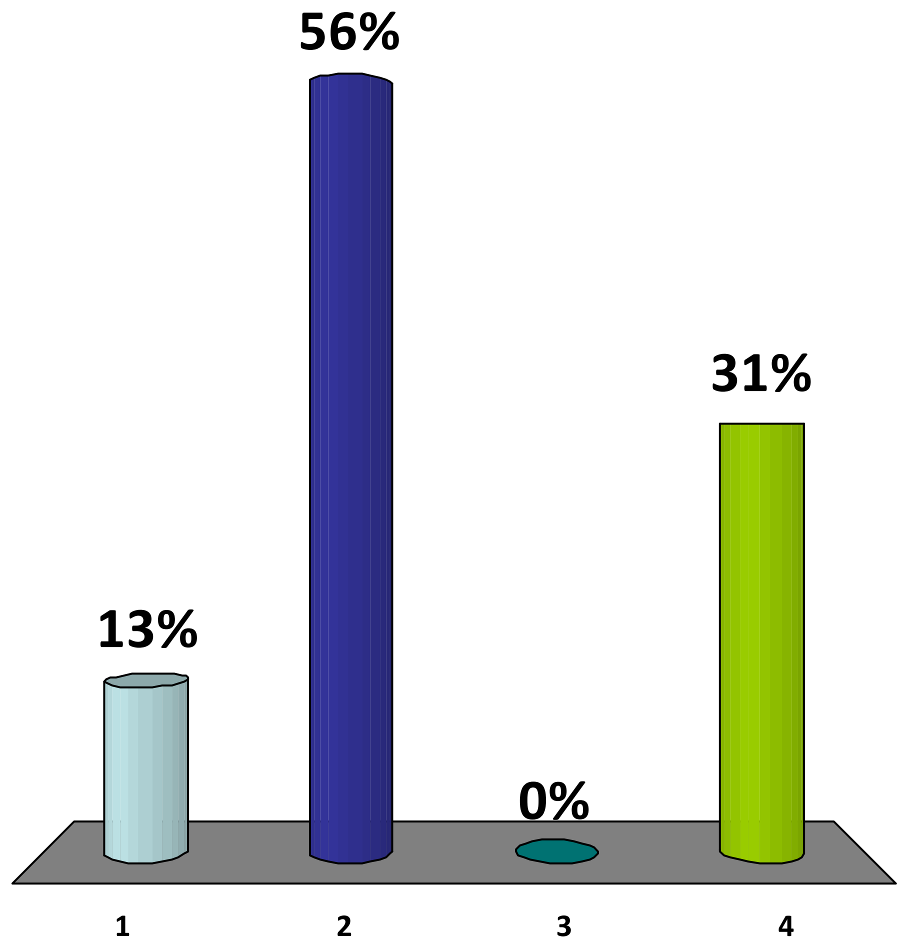
Have you heard of QuickSort?

1. Yes
2. No



Which is fastest?

1. MergeSort
2. QuickSort
3. InsertionSort
4. I don’t know



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

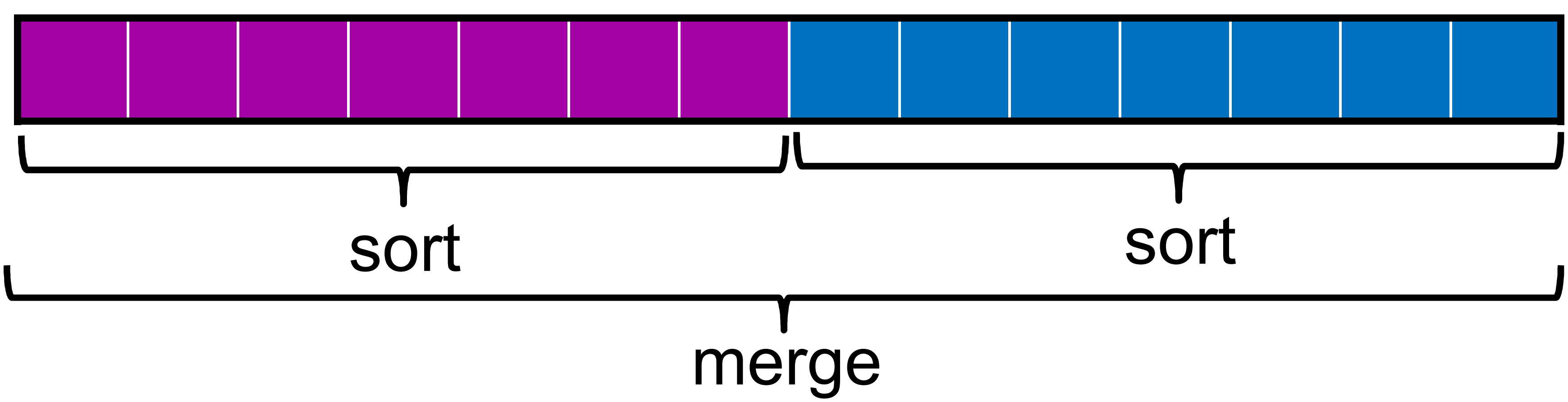
QuickSort

In class:

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

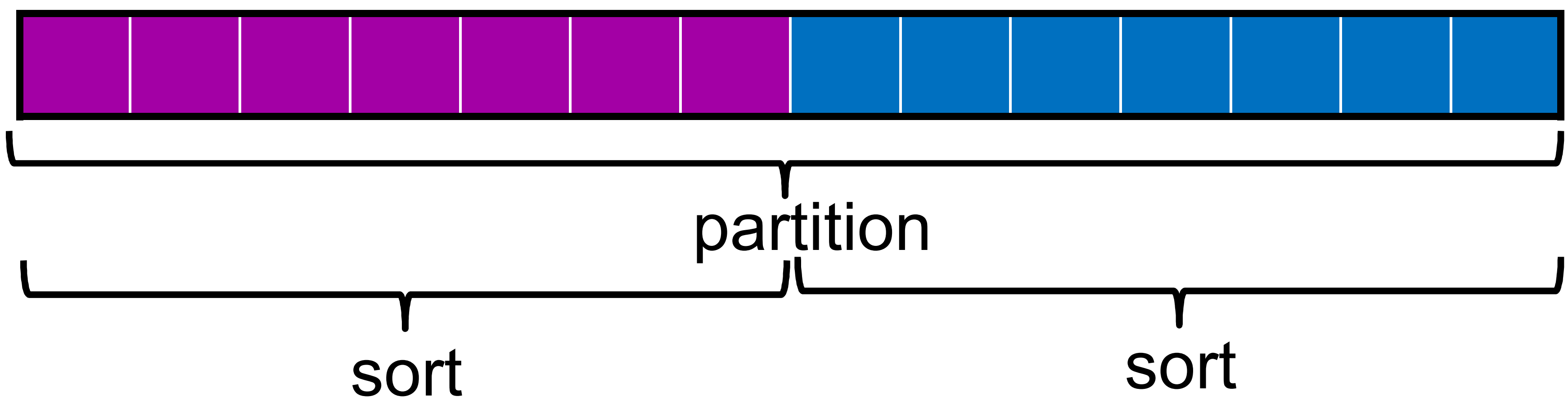
Recall: MergeSort

```
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

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	$> x$

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

Key: efficient *partition* sub-routine

Partitioning an Array

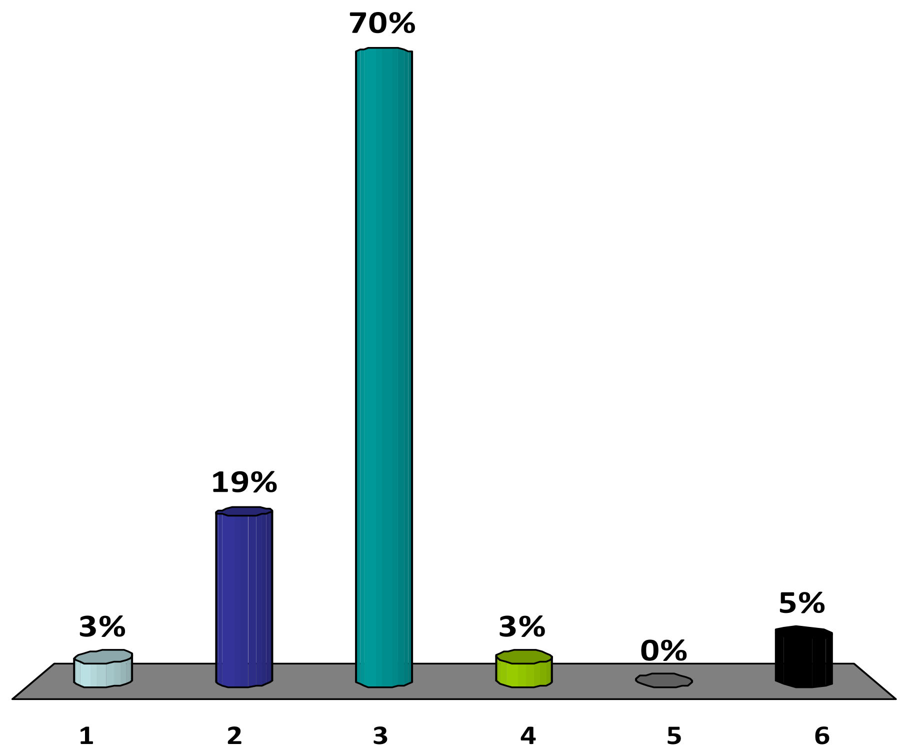
Three steps:

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

The following array has been partitioned around which element?

18	5	6	1	10	22	40	32	50

1. 6
2. 10
3. ✓ 22
4. 40
5. 32
6. I don't know.



Partitioning an Array

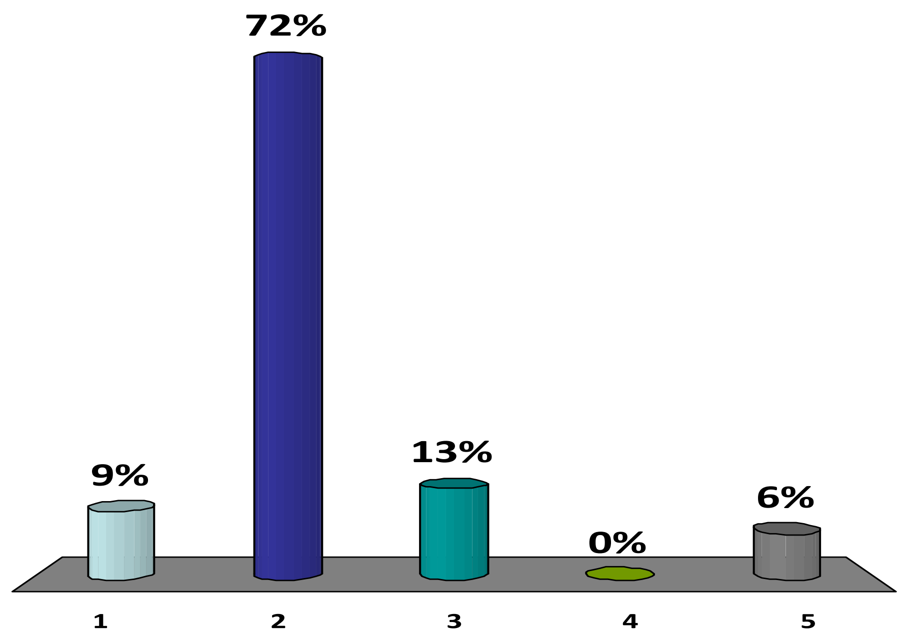
Example:

22	1	6	40	32	10	18	50	4

Goal: petition array around pivot 22

How long does it take to partition?

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



Partitioning an Array

Example: partition around 22

22	1	6	40	32	10	18	50	4

Output array:

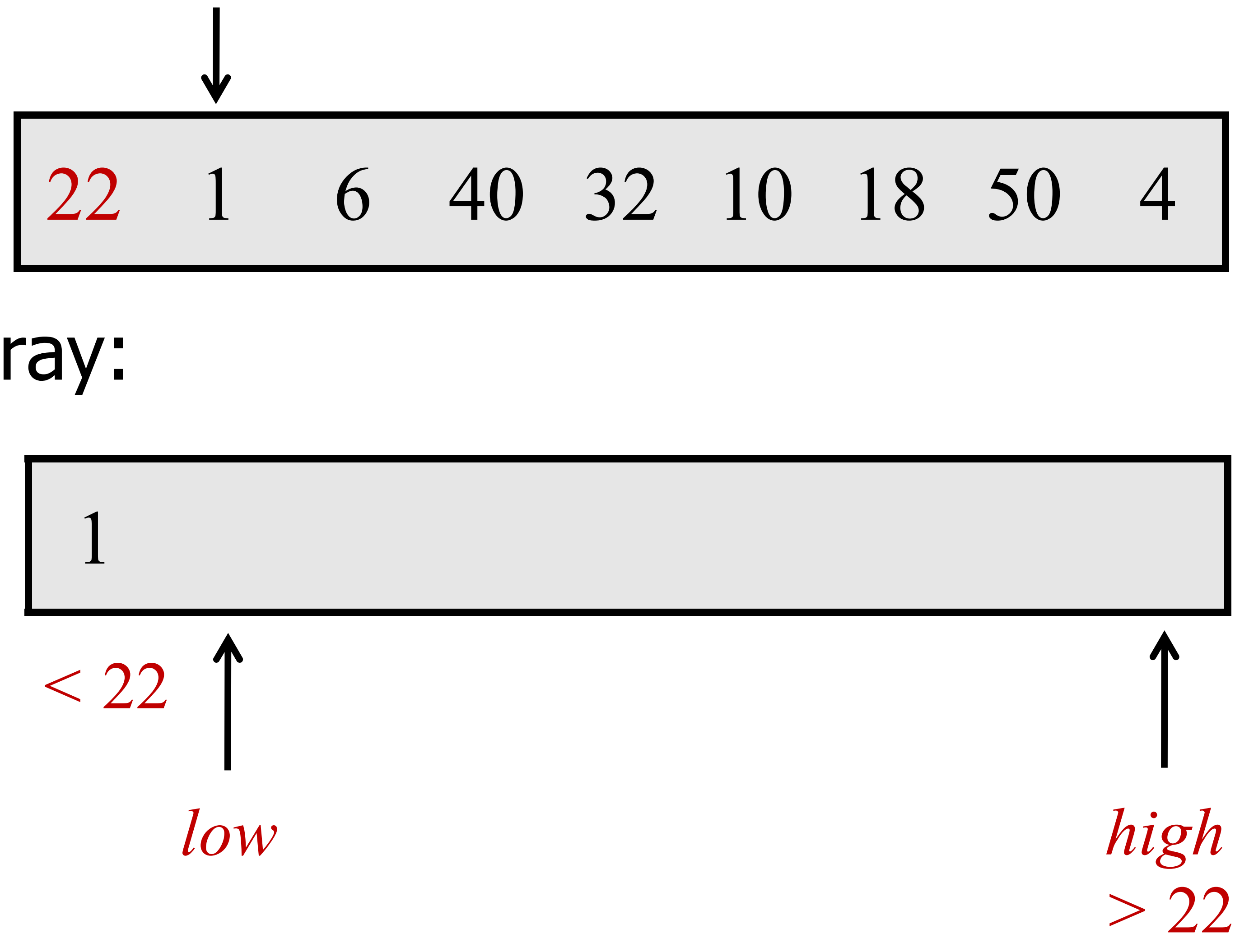
low
< 22

high
> 22

Partitioning an Array

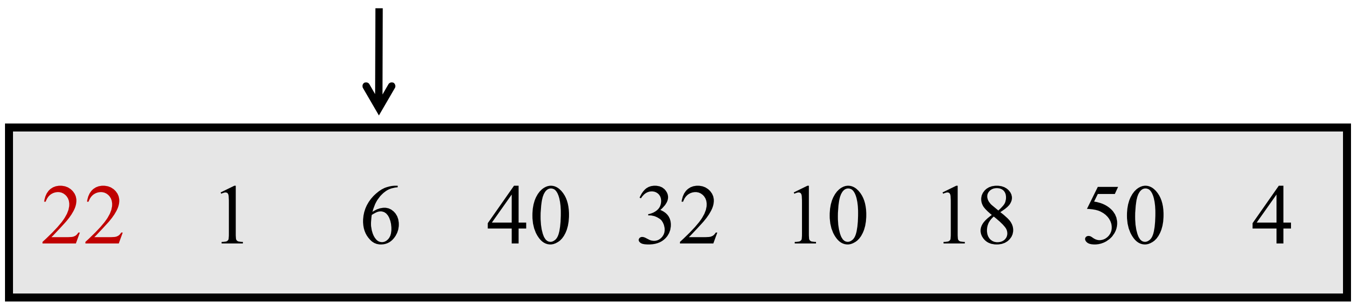
Example: partition around 22

Output array:

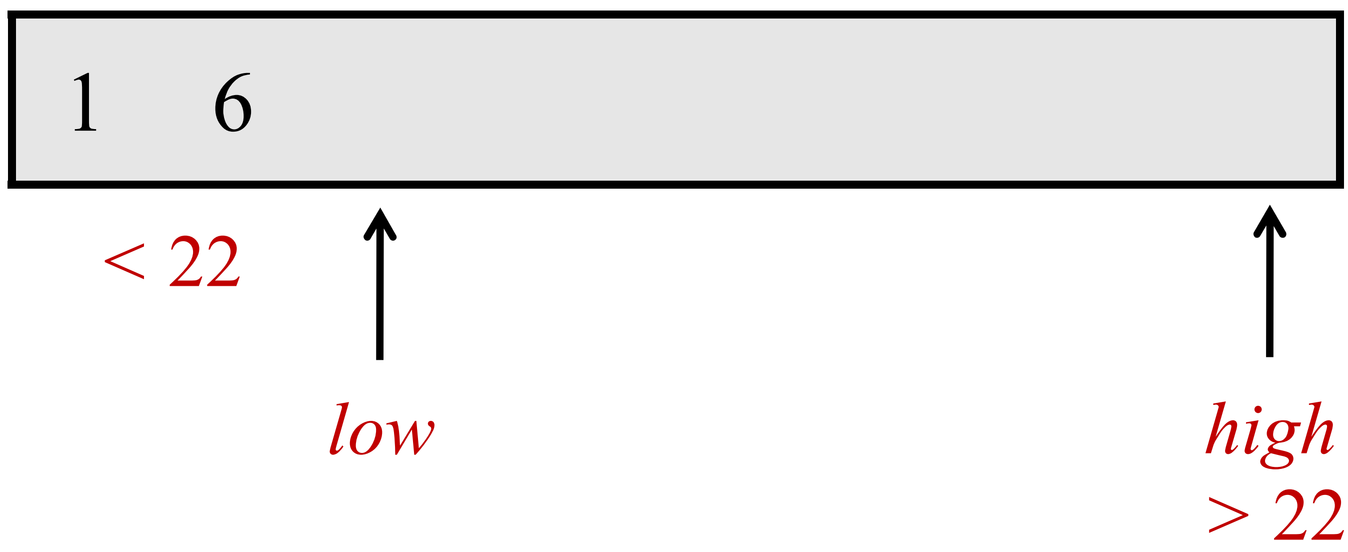


Partitioning an Array

Example: partition around 22

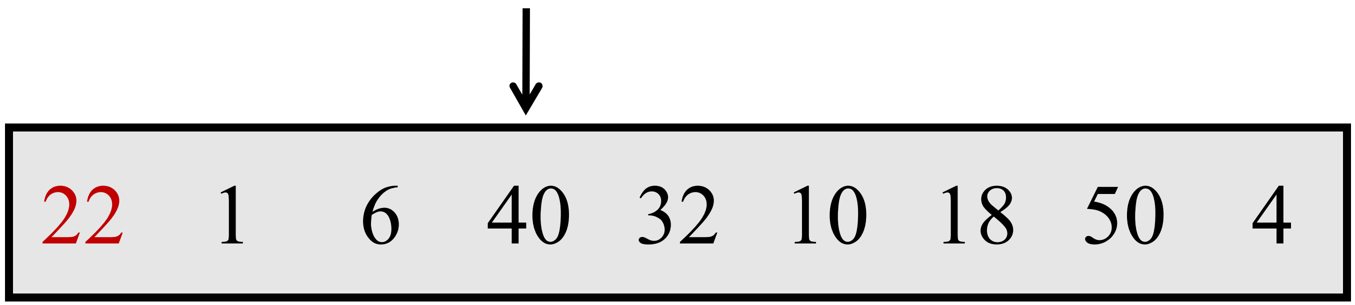


Output array:

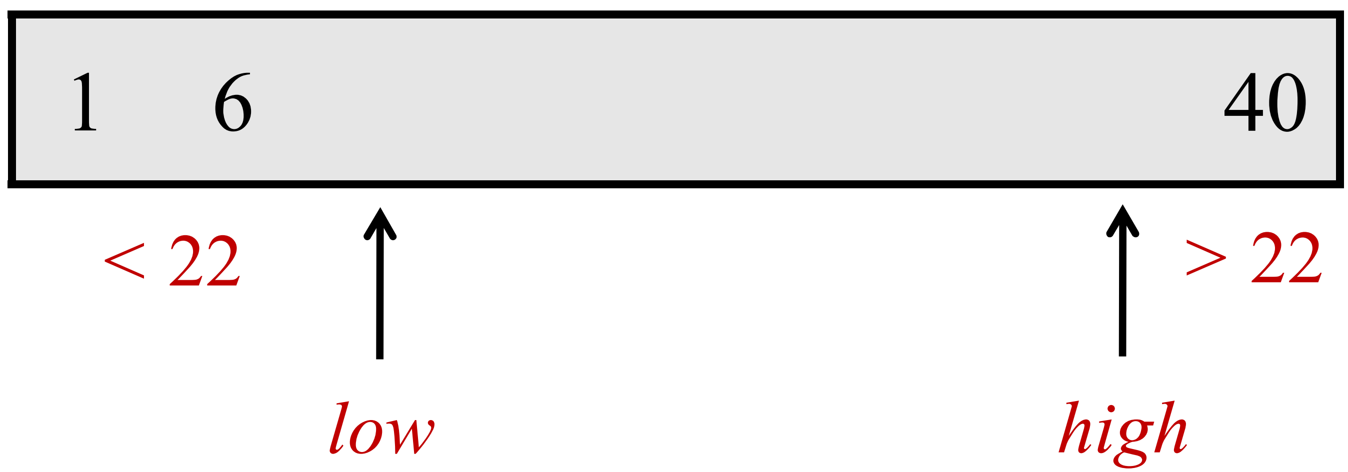


Partitioning an Array

Example: partition around 22

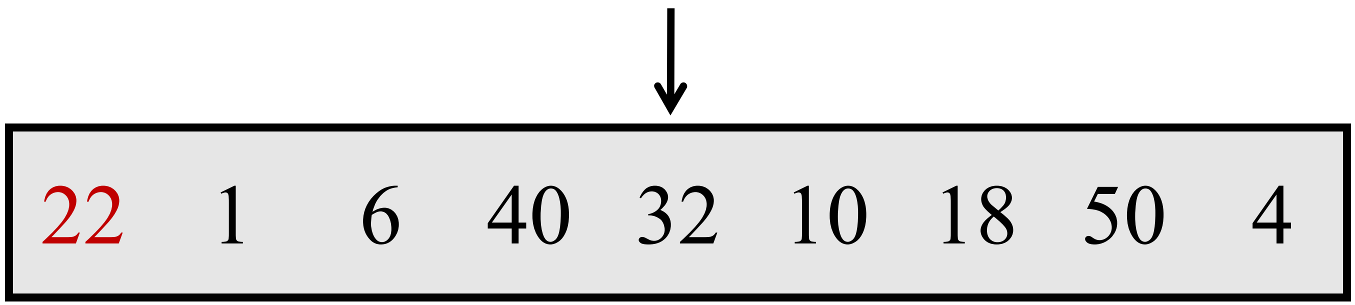


Output array:

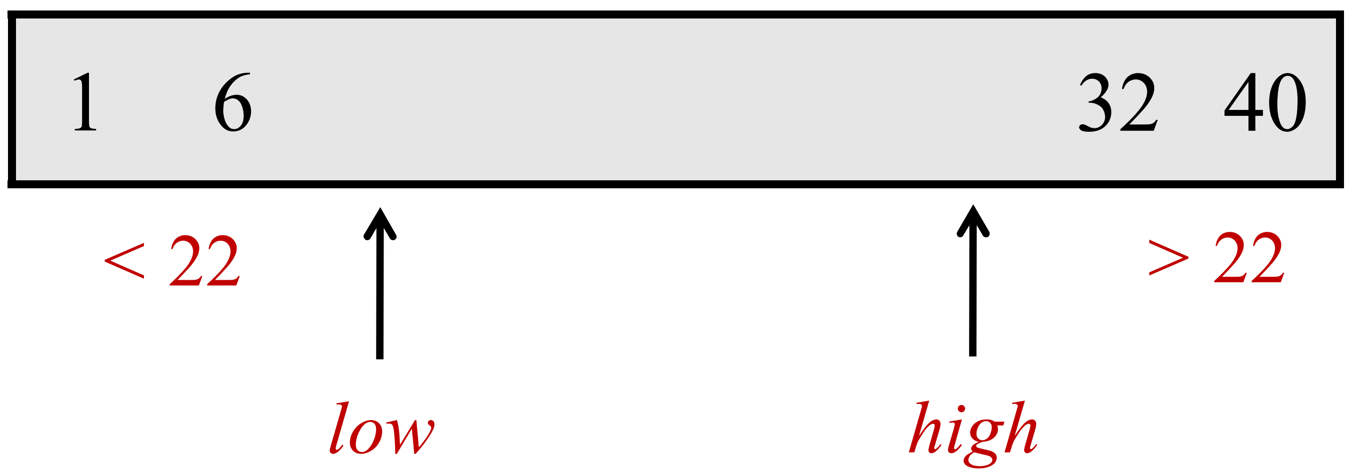


Partitioning an Array

Example: partition around 22

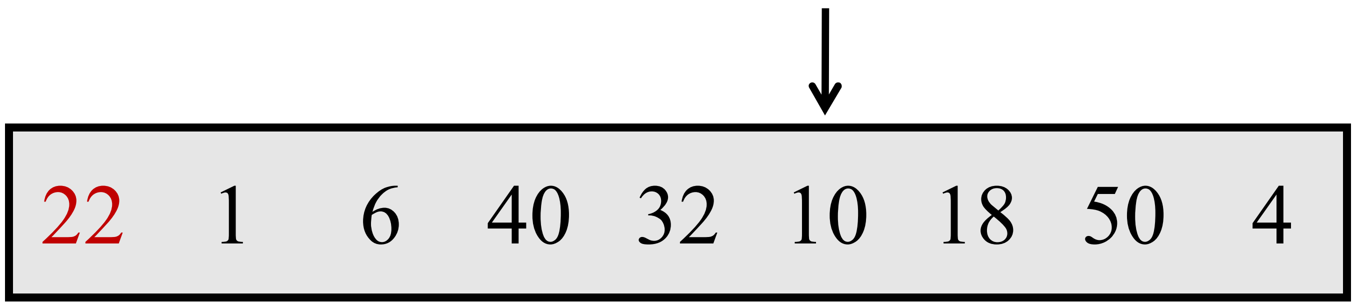


Output array:

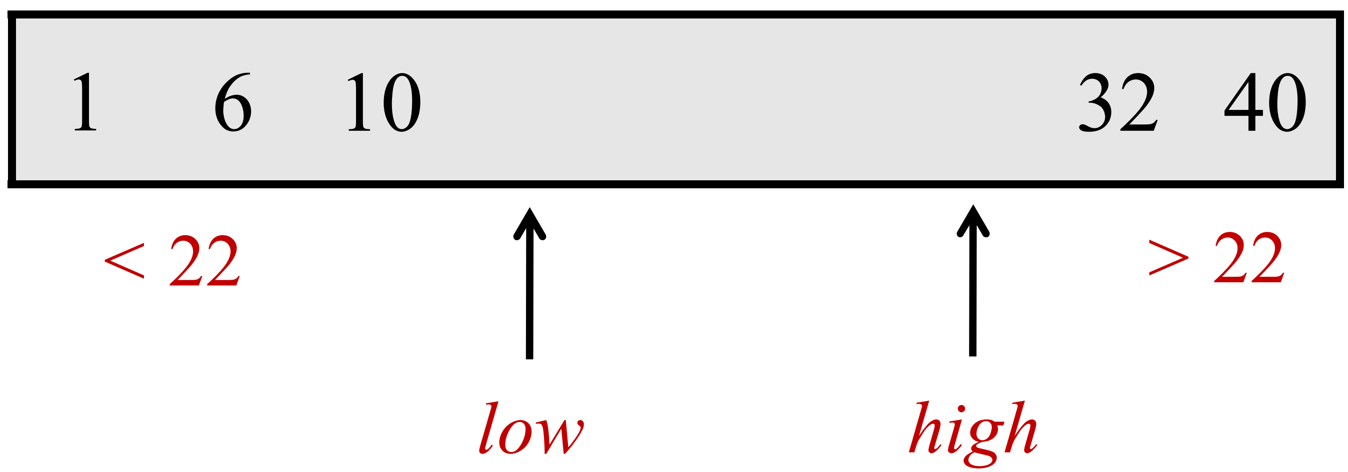


Partitioning an Array

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				32	40

< 22 *low*   *high* > 22

Partitioning an Array

Example: partition around 22

22	1	6	40	32	10	18	50	4

Output array:

1	6	10	18			50	32	40

< 22 > 22

low *high*

Partitioning an Array

Example: partition around 22

22	1	6	40	32	10	18	50	4

Output array:

1	6	10	18	4		50	32	40

< 22 > 22

high

low

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   > 22

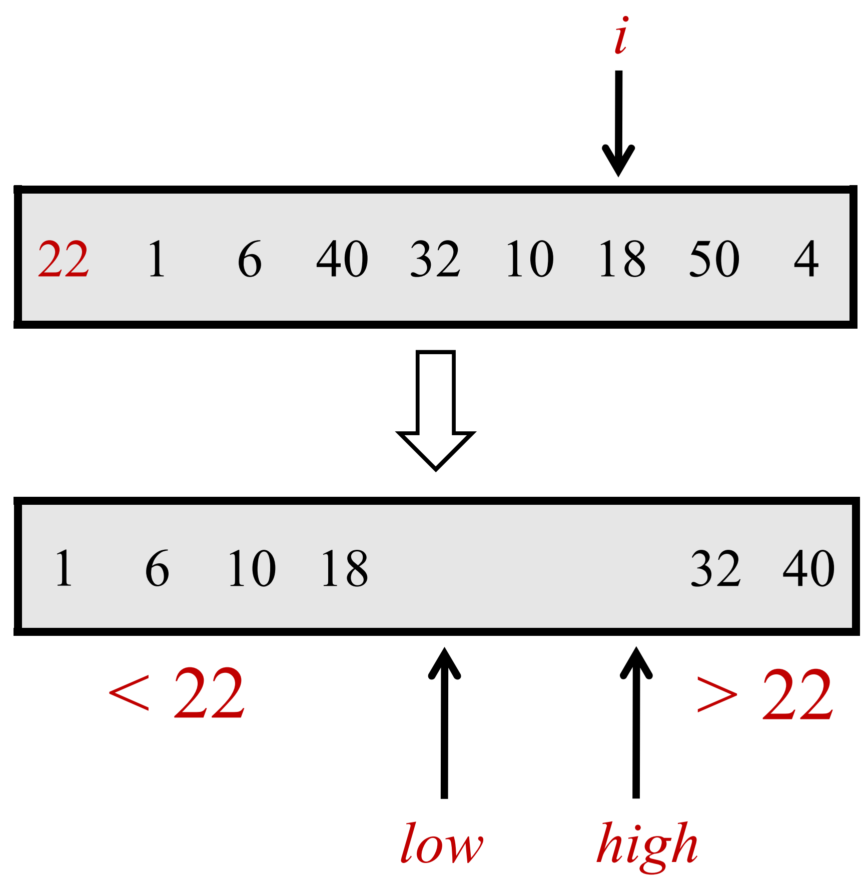
high
low

Partition

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

// Assume no duplicates

```
B = new n element array
low = 1;
high = n;
for (i = 1; 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 >
```



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: $B[i] = pivot$

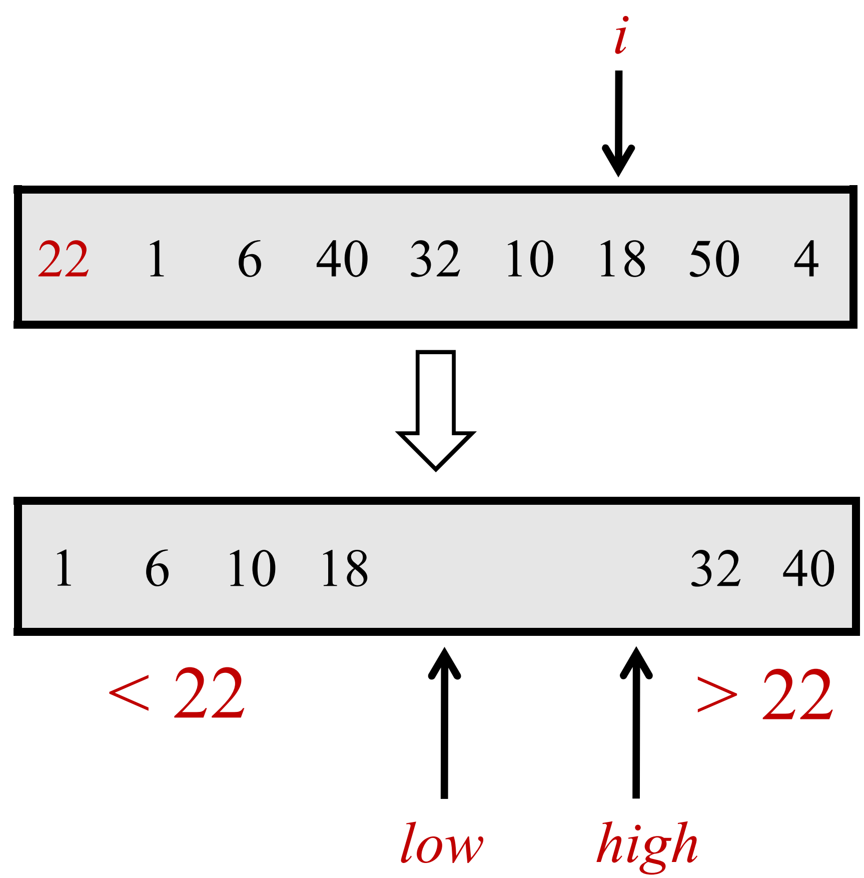
By invariants, B is partitioned around the pivot.

What is wrong with the partition procedure?

3%	1. There is a bug. It doesn't work.
50%	2. It uses too much memory.
11%	3. It is too slow.
0%	4. It only works for integers.
25%	5. It does not choose a good pivot.
11%	6. It works perfectly.

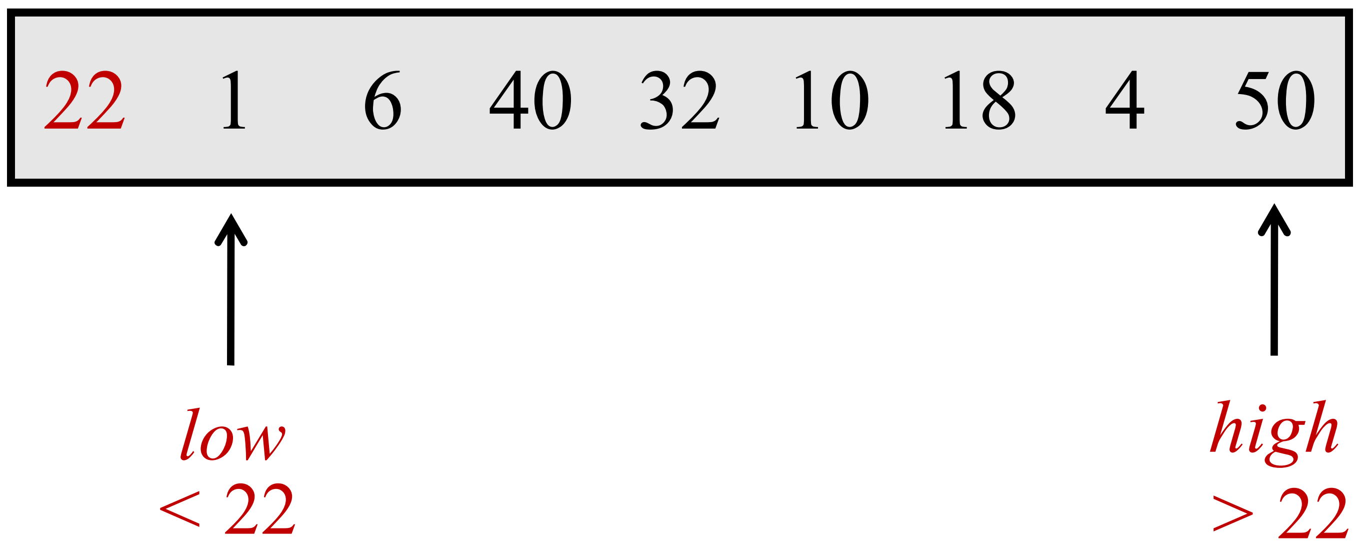
Partition

```
partition(A[1..n], n, pivot)                    // Assume no duplicates
    B = new n element array
    low = 1;
    high = n;
    for (i = 1; 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 >
```



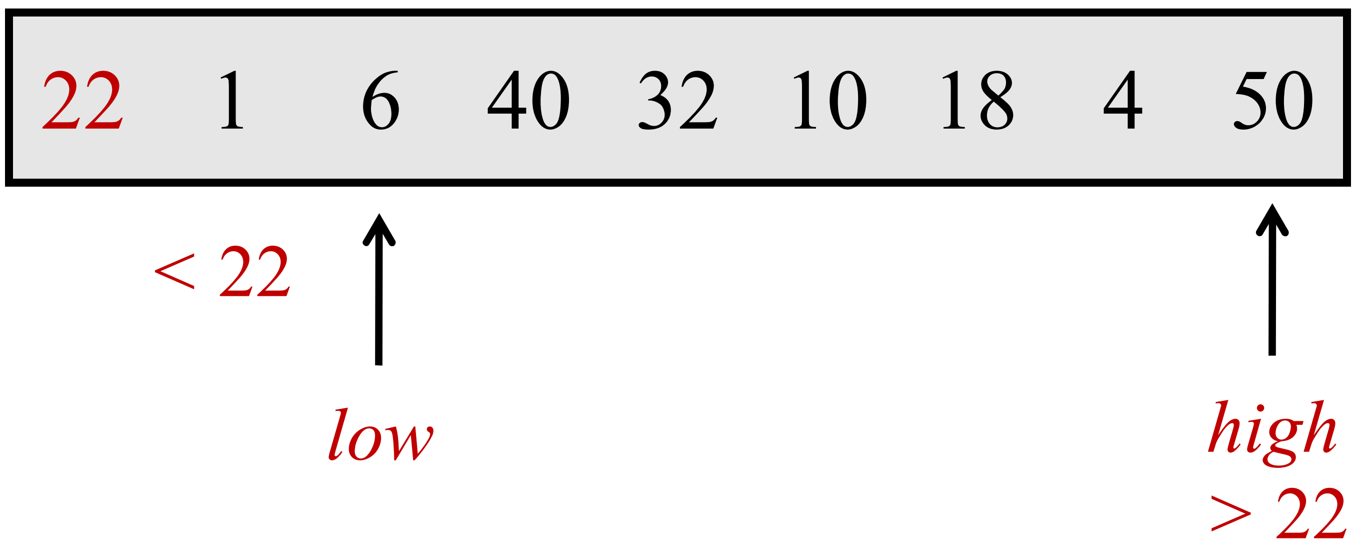
Partitioning an Array

Example: partition around 22



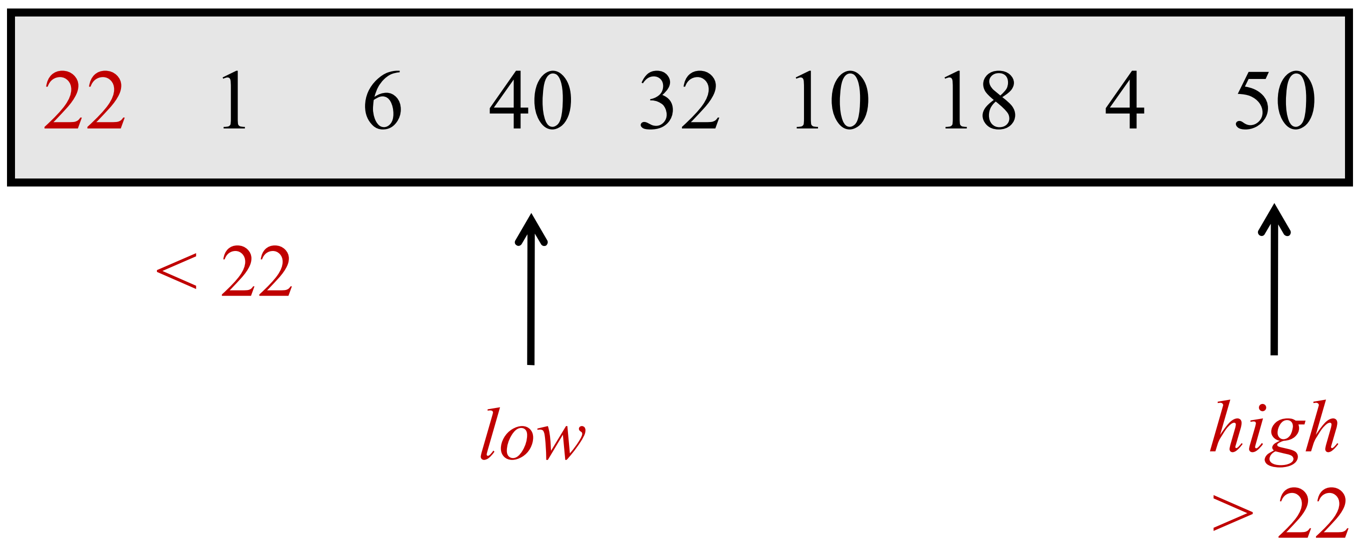
Partitioning an Array

Example: partition around 22



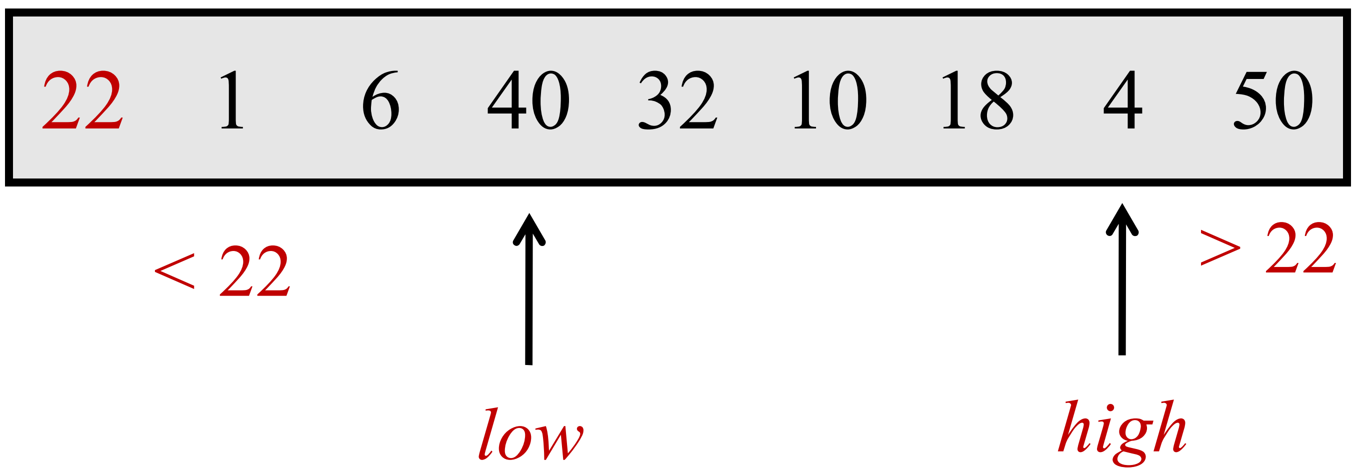
Partitioning an Array

Example: partition around 22



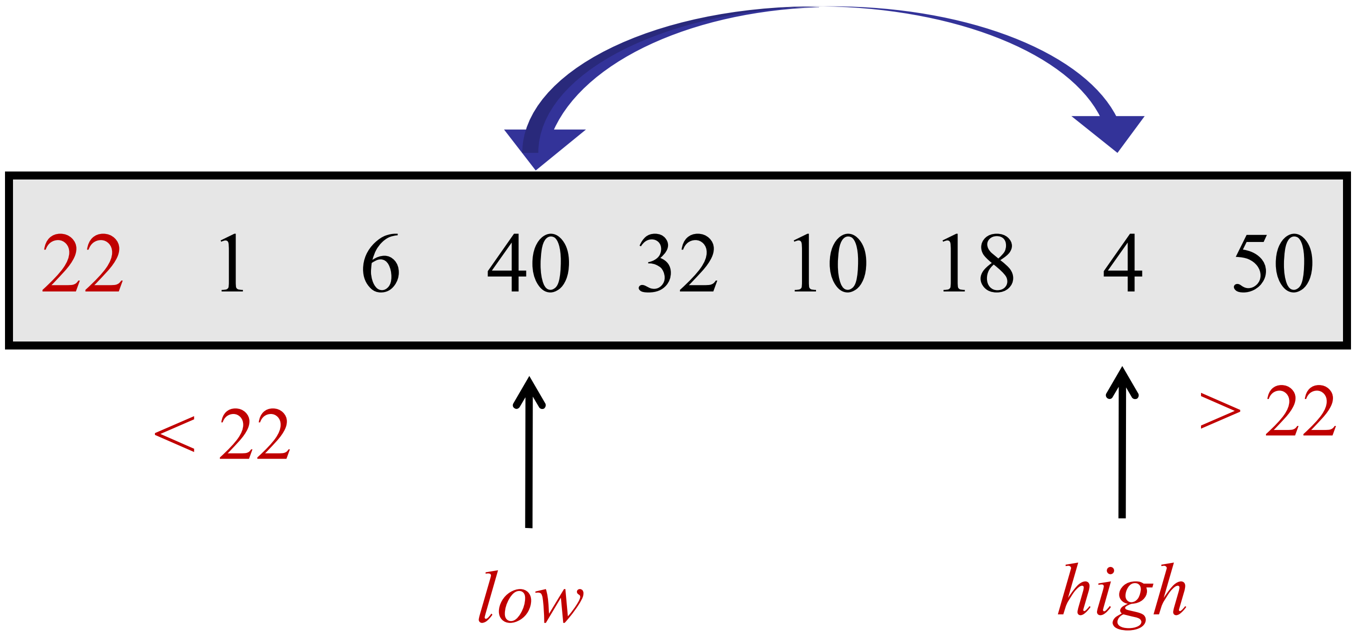
Partitioning an Array

Example: partition around 22



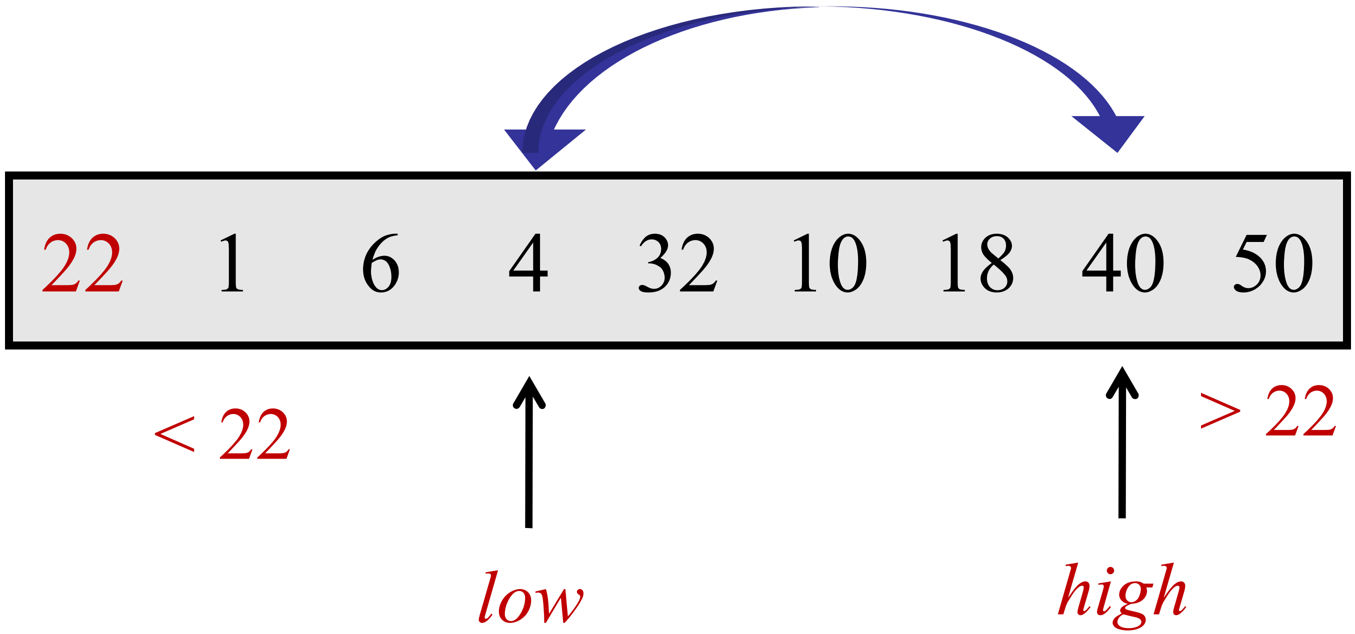
Partitioning an Array

Example: partition around 22



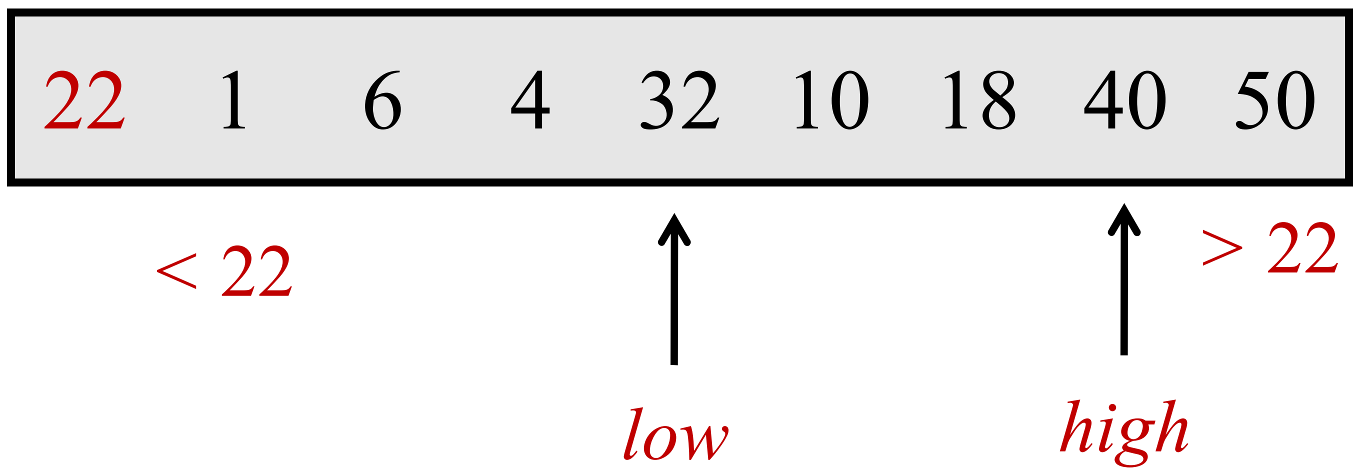
Partitioning an Array

Example: partition around 22



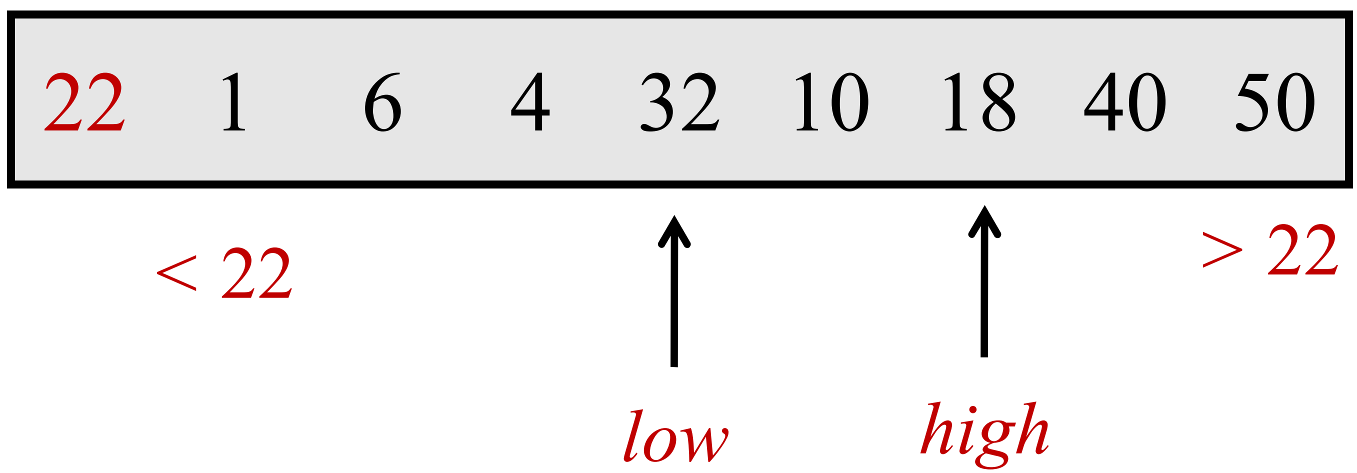
Partitioning an Array

Example: partition around 22



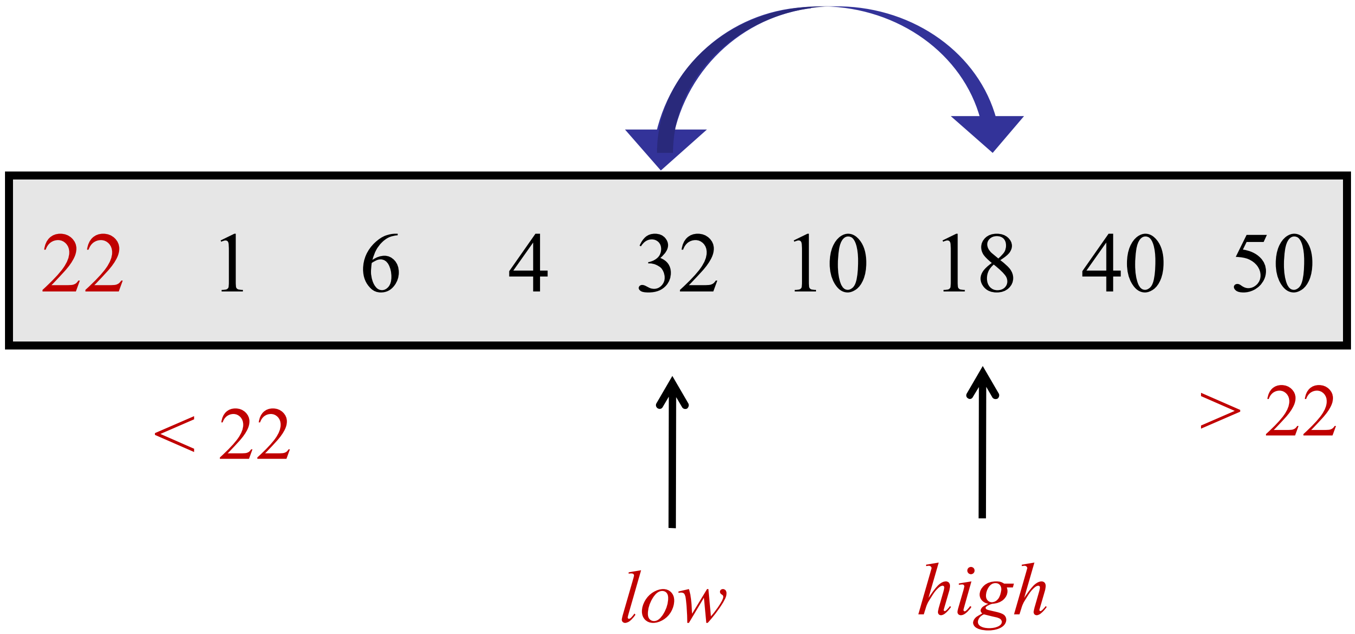
Partitioning an Array

Example: partition around 22



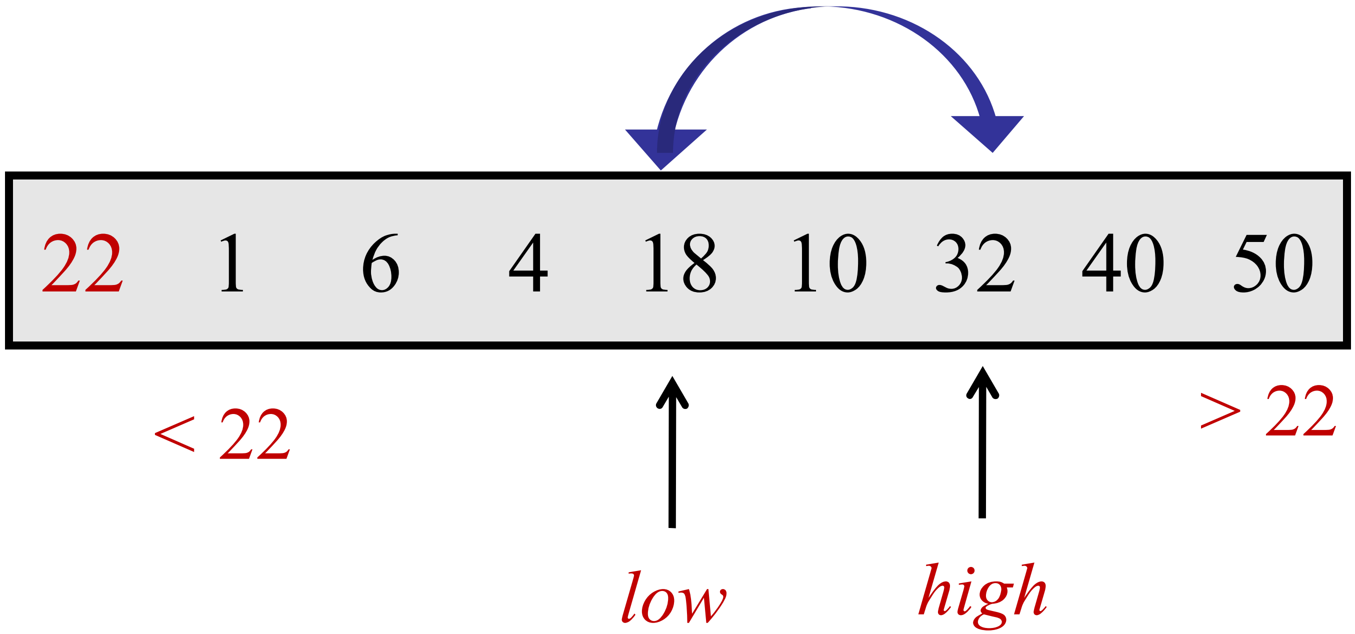
Partitioning an Array

Example: partition around 22



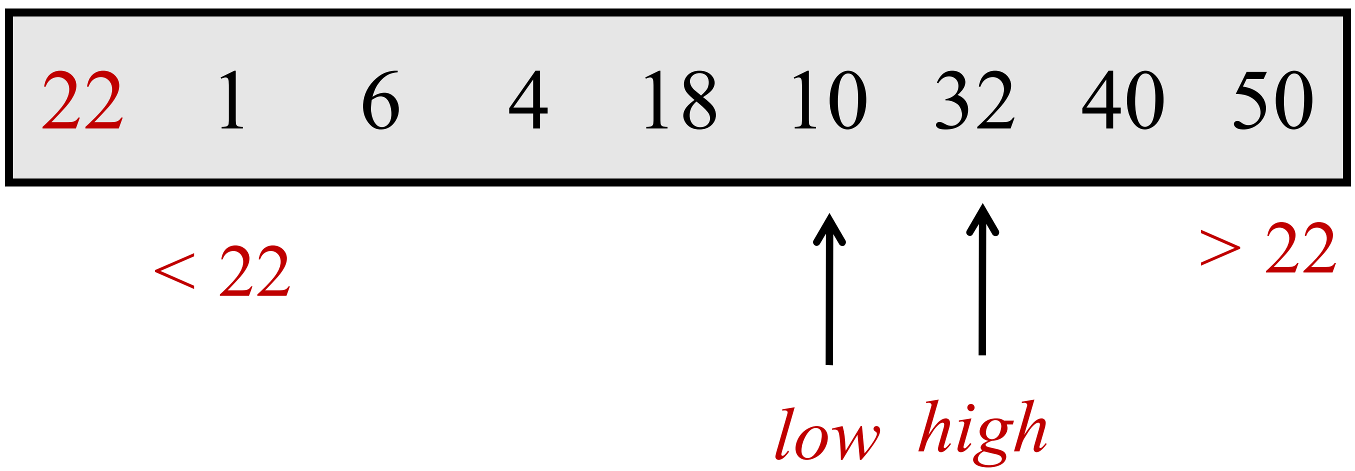
Partitioning an Array

Example: partition around 22



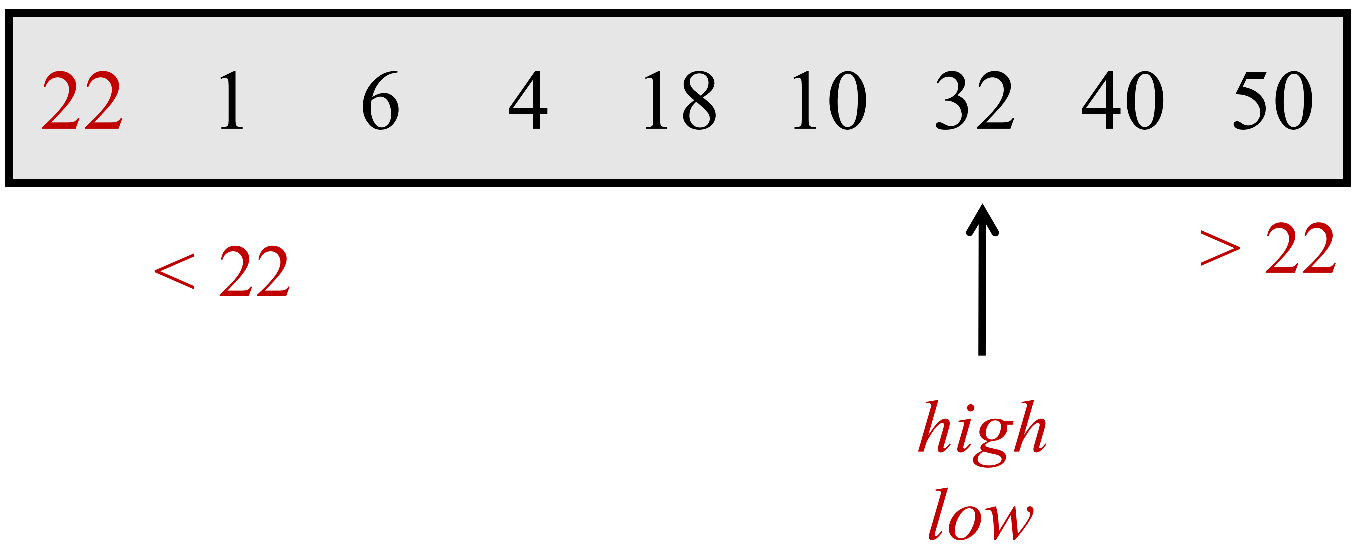
Partitioning an Array

Example: partition around 22



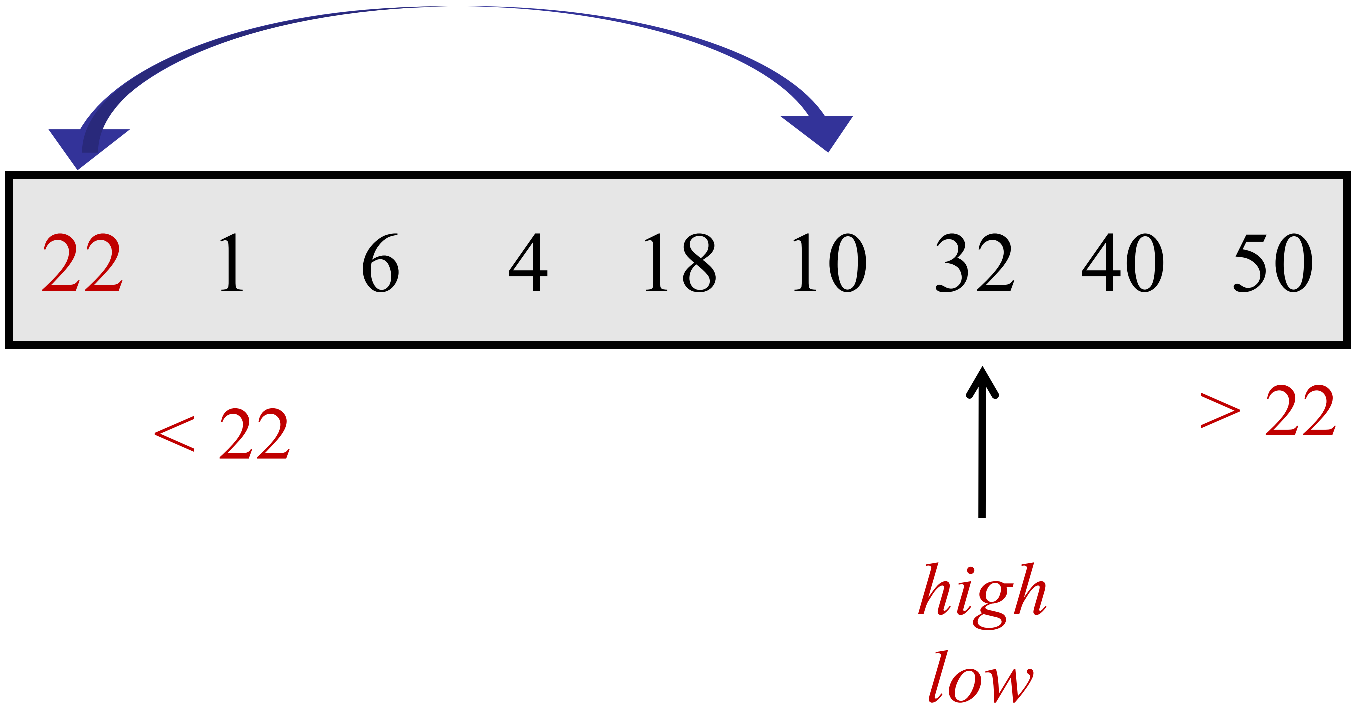
Partitioning an Array

Example: partition around 22



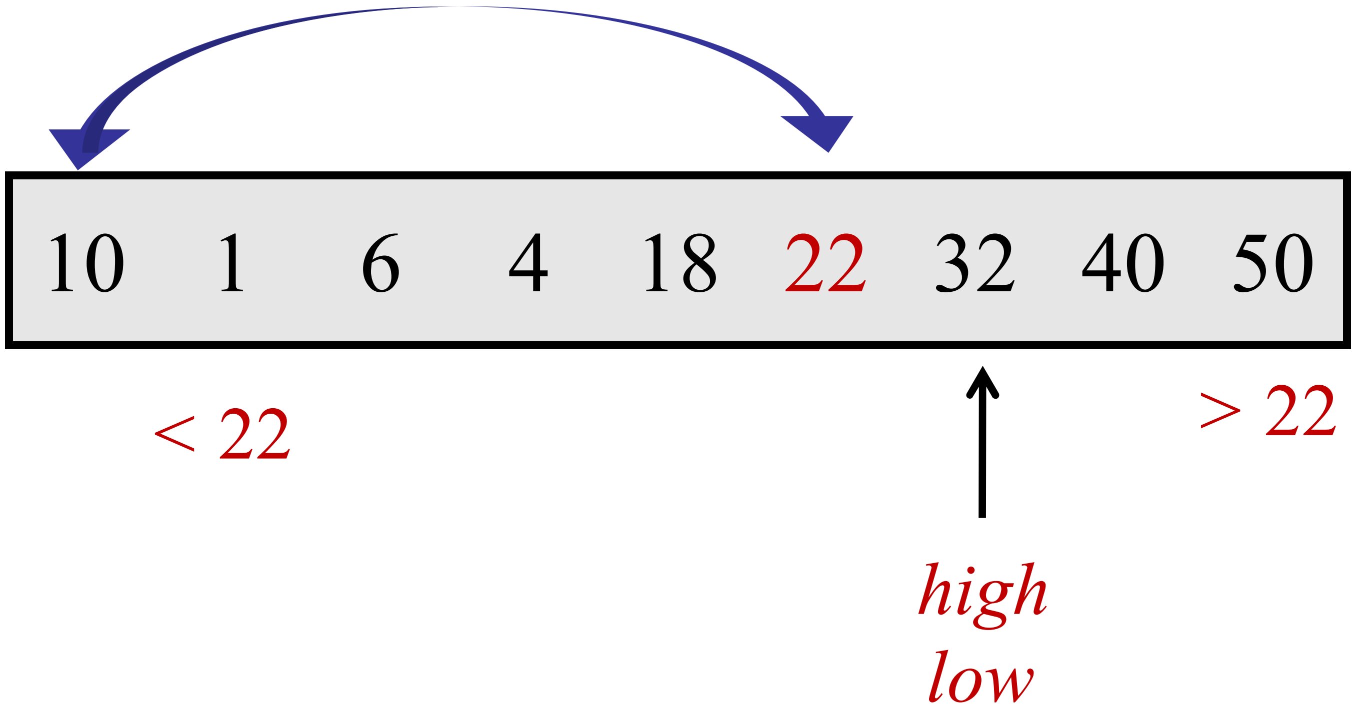
Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22



Partition

```
partition(A[1..n], n, pIndex)                   // Assume no duplicates, n>1
    pivot = A[pIndex];                          // pIndex is the index of the 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

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

Proof:

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

Partition

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

Proof: During loop:

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

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

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

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

Partition

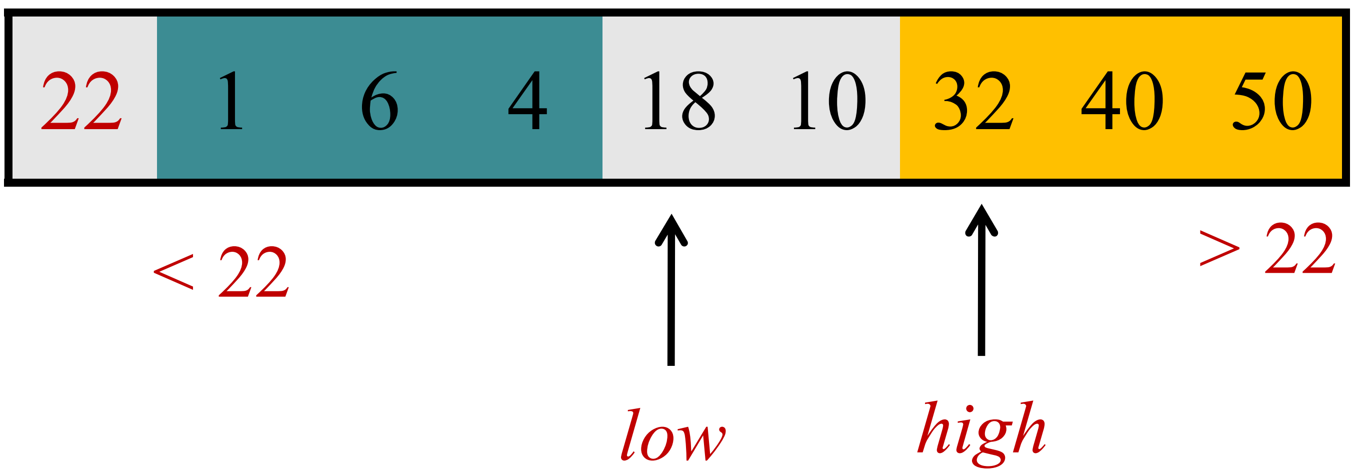
```
partition(A[1..n], n, pIndex)            // Assume no duplicates, n>1
    pivot = A[pIndex];                   // pIndex is the index of the pivot
    swap(A[1], A[pIndex]);               // store pivot in A[1]
    low = 2;                             // start after pivot in A[1]
    high = n+1;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

Claim: At the end of every loop iteration:

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

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

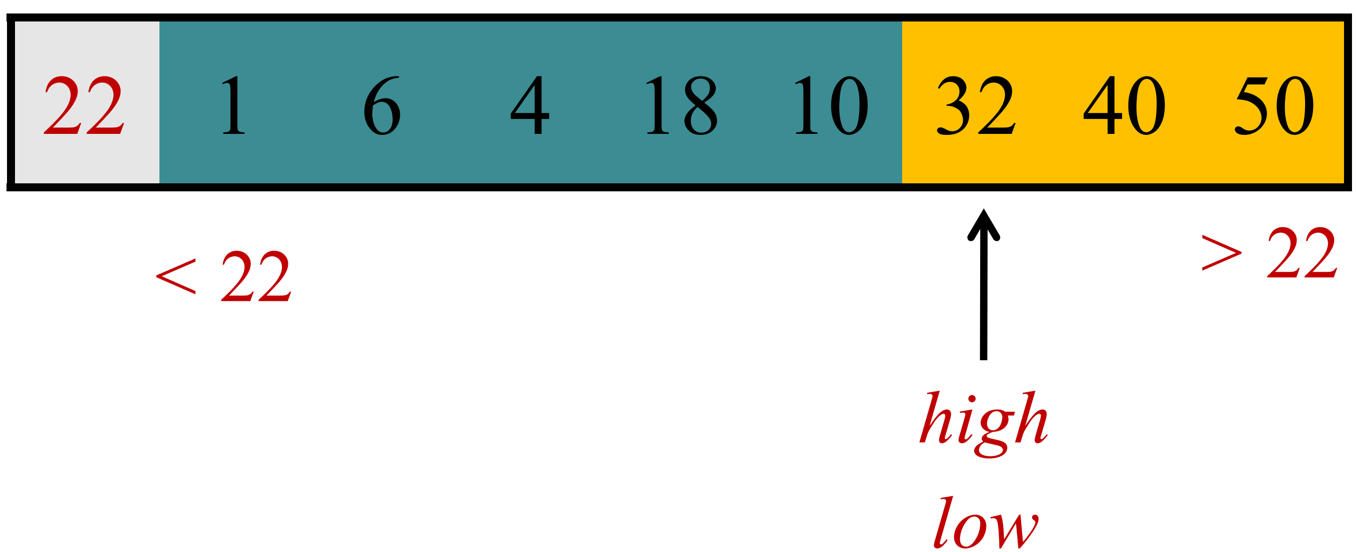


Partition

Claim: At the end of every loop iteration:

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

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



Partition

Claim: At the end of every loop iteration:

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

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

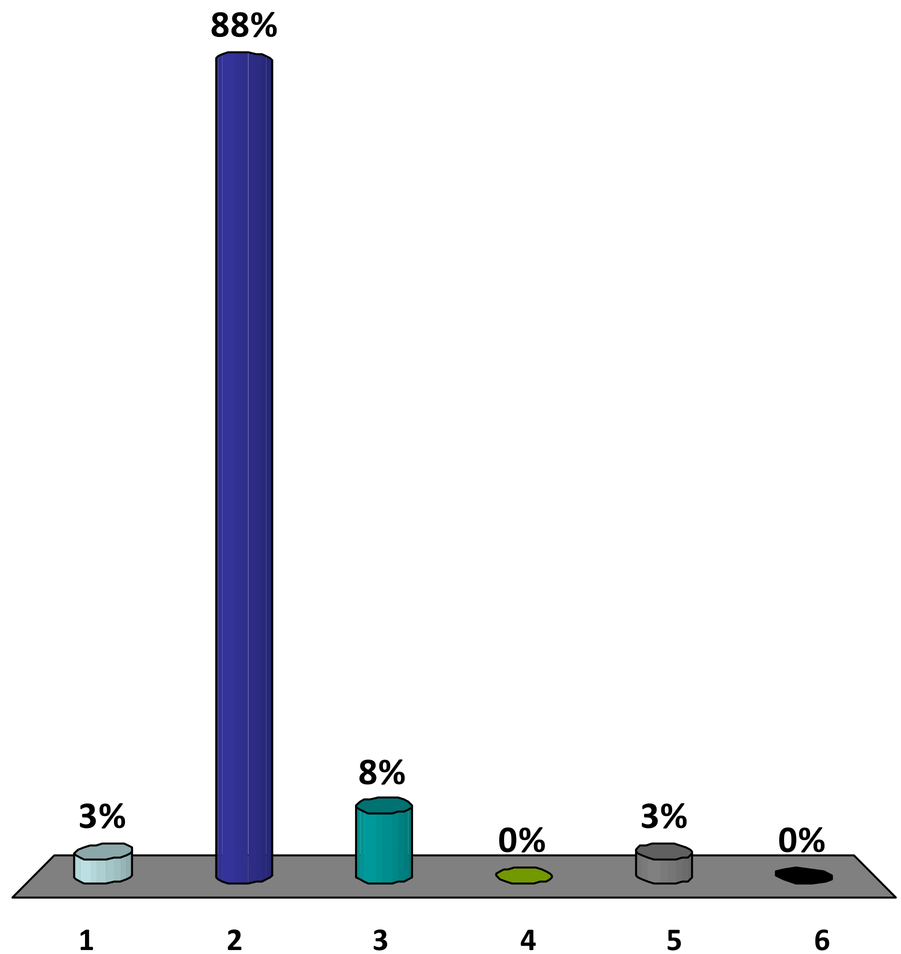
Claim: Array A is partitioned around the pivot

Partition

```
partition(A[1..n], n, pIndex)                  // Assume no duplicates, n>1
    pivot = A[pIndex];                         // pIndex is the index of the 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;
```

The running time for (in-place) partition is:

1. $O(\log n)$
2. $O(n)$
3. $O(n \log n)$
4. $O(n^{1.5})$
5. $O(n^2)$
6. None of the above.



QuickSort

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

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

else

Choose pivot index *pIndex*.

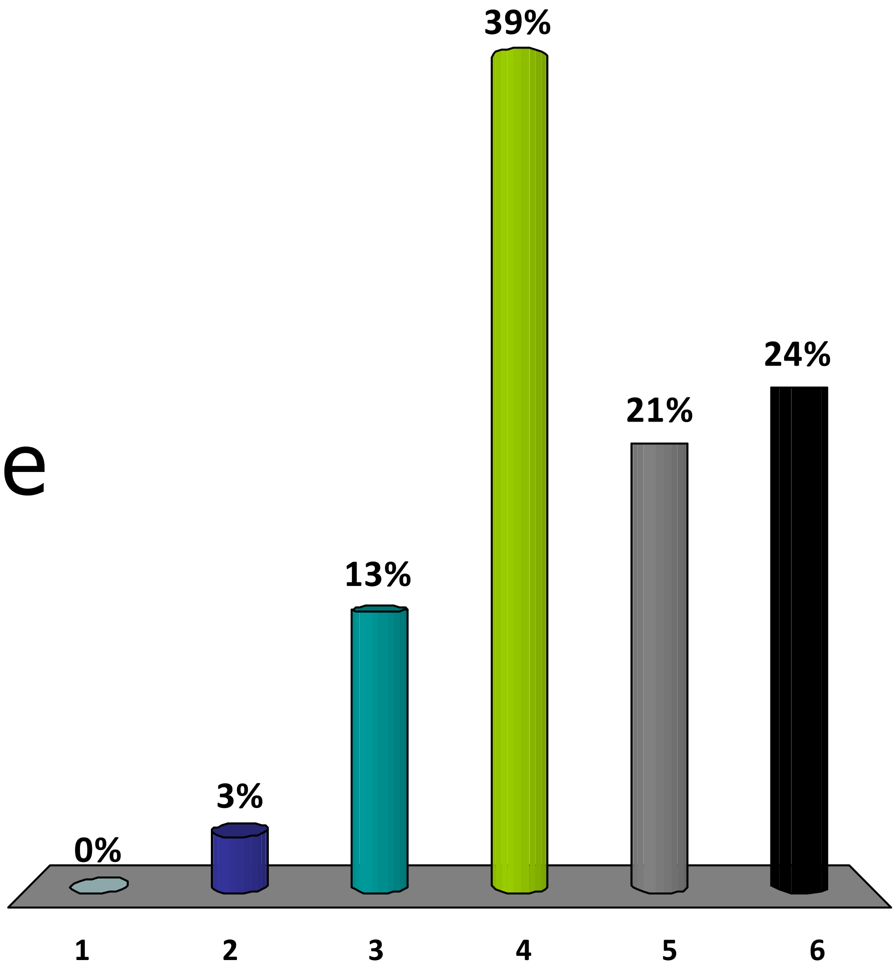
p = partition($A[1..n]$, n, *pIndex*)

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

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

What is a good (deterministic) choice for the pivot?

1. The first element.
2. The last element.
3. The middle element.
4. The median of the first, the last, and the middle element.
5. It does not matter.
6. None of the above.



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

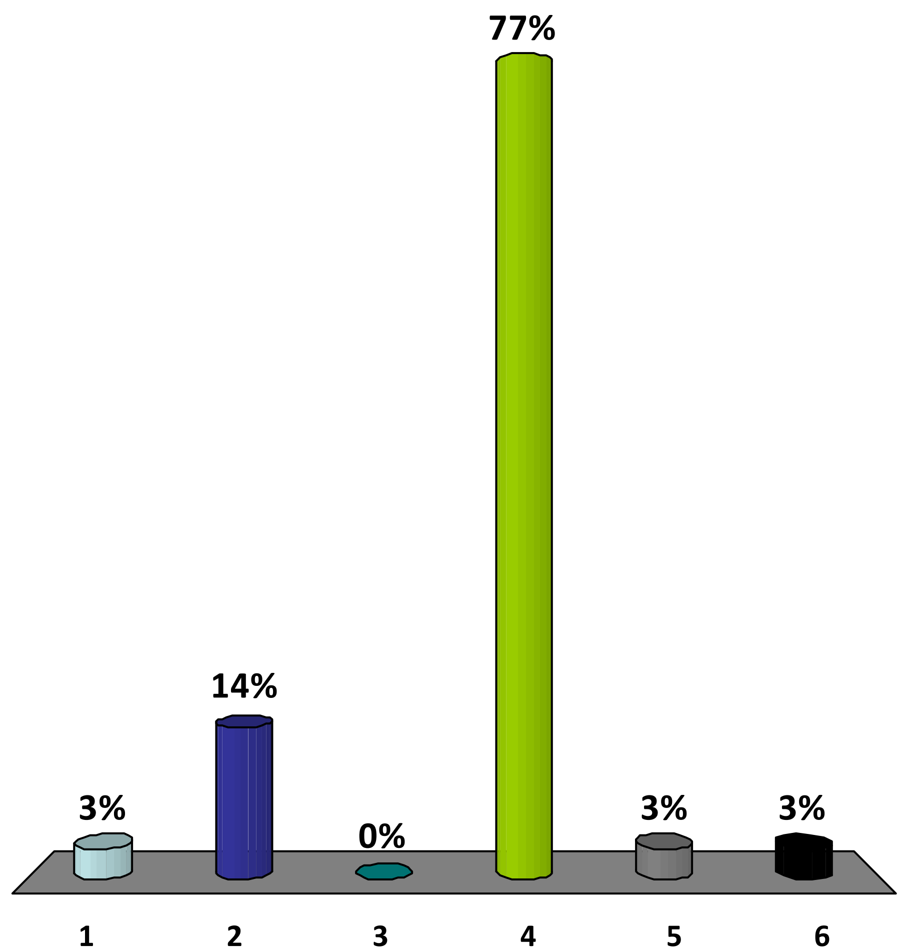
Choose $A[1]$ for pivot:

98	97	96	95	94	93	92	99	100

97	96	95	94	93	92	98	99	100

The worst-case running time for QuickSort where pivot=A[1] is:

1. O(log n)
2. O(n)
3. O(n log n)
4. O(n^2)
5. O($n*2^{\log\log(n)}$)
6. None of the above.



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

Which recurrence best describes QuickSort when the pivot is chosen as A[1]?

.8%	1.	$T(n) = 2T(n/2) + cn$
%)	2.	$T(n) = 2T(n/2) + c$
%	3.	$T(n) = T(n/2) + cn$
4%	✓ 4.	$T(n) = T(n-1) + T(1) + cn$
%	5.	$T(n) = T(n-1) + T(1) + c$
%	6.	$T(n) = T(n/4) + T(3n/4) + cn$

Deterministic QuickSort

QuickSort Recurrence:

$$T(n) = T(n\text{-}1) + T(1) + cn$$

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

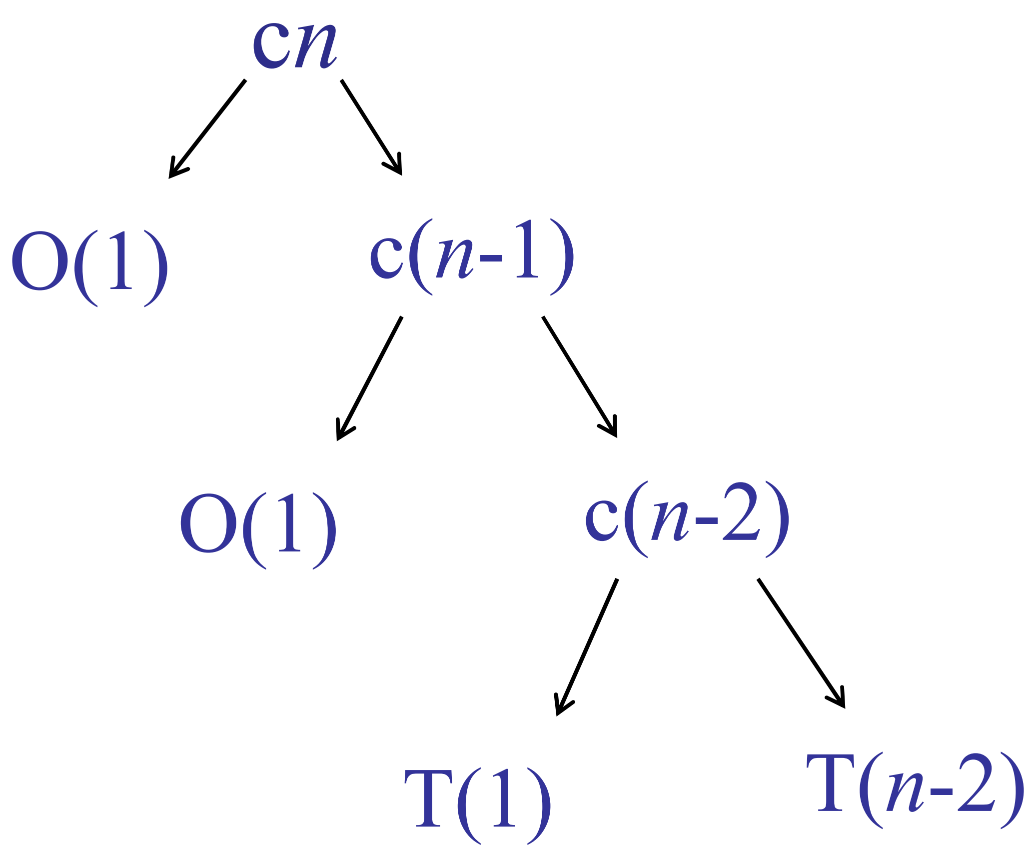
Deterministic QuickSort

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

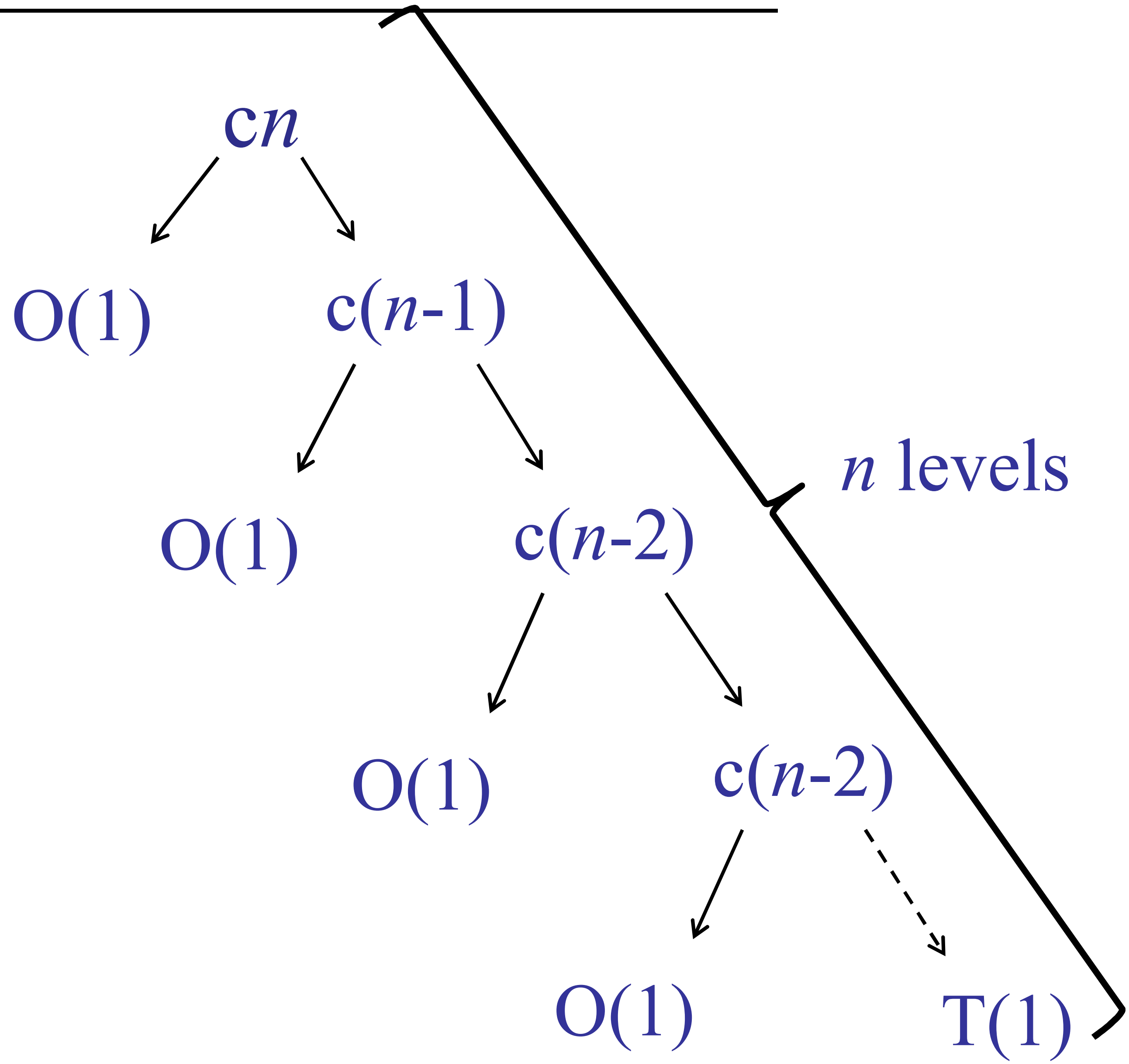
Deterministic QuickSort

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

Deterministic QuickSort

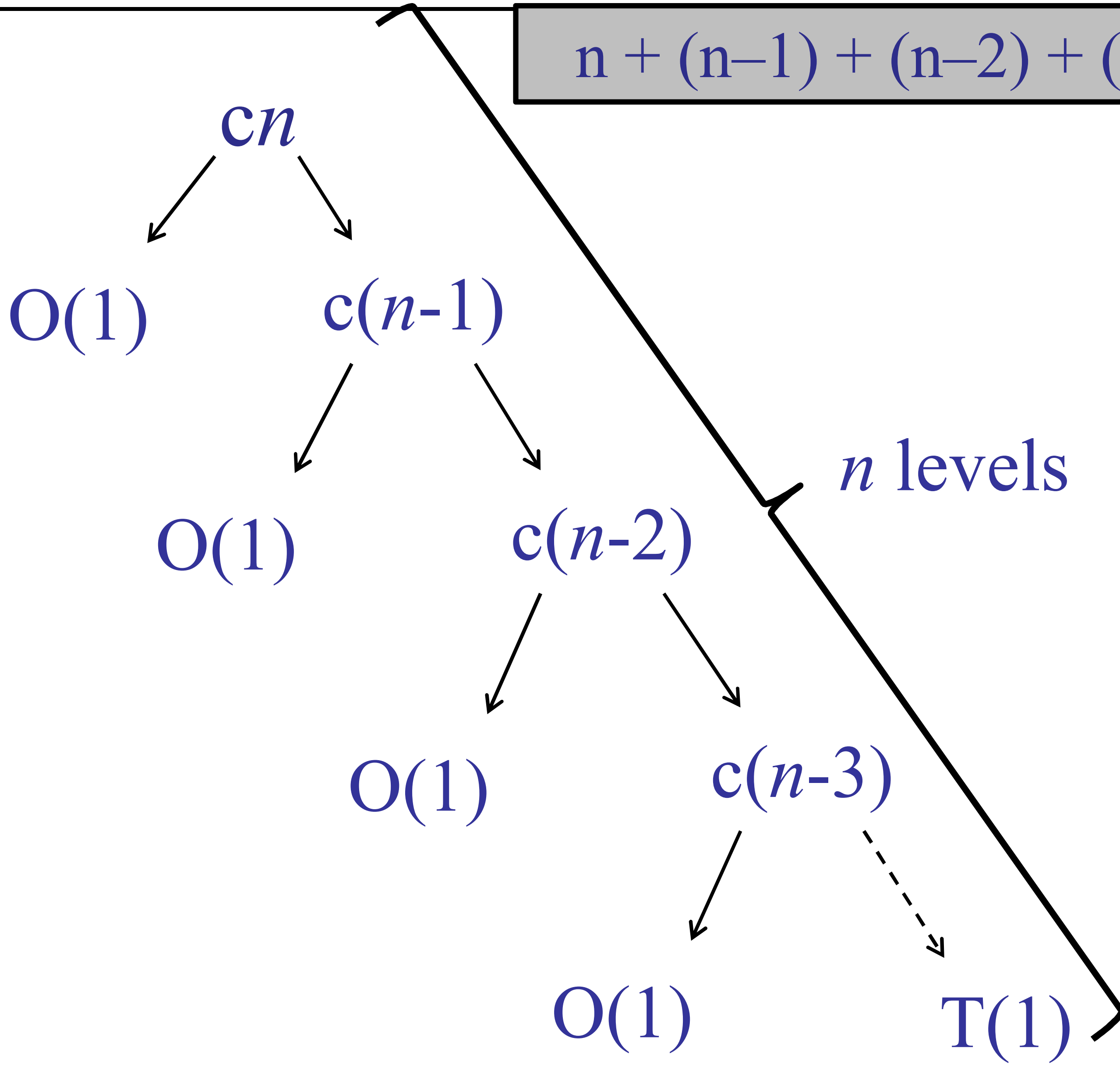


Deterministic QuickSort



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?

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

Cost of partition on n elements

Cost of QuickSort on *high* elements

Cost of QuickSort on *low* elements

Cost of QuickSort on n elements

What is the performance of QuickSort where the pivot = median(A)?

3%	1. O(log n)
8%	2. O(n)
79% ✓	3. O(n log n)
5%	4. O(n^2)
3%	5. O(n^3)
3%	6. None of the above.

Lucky 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 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:

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

Cost of partition on n elements

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

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

Cost of QuickSort on n elements

QuickSort

Tempting solution:

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

What is wrong?

QuickSort

Tempting solution:

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

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

$k = n/10$

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

QuickSort Analysis

$$k = n/10$$

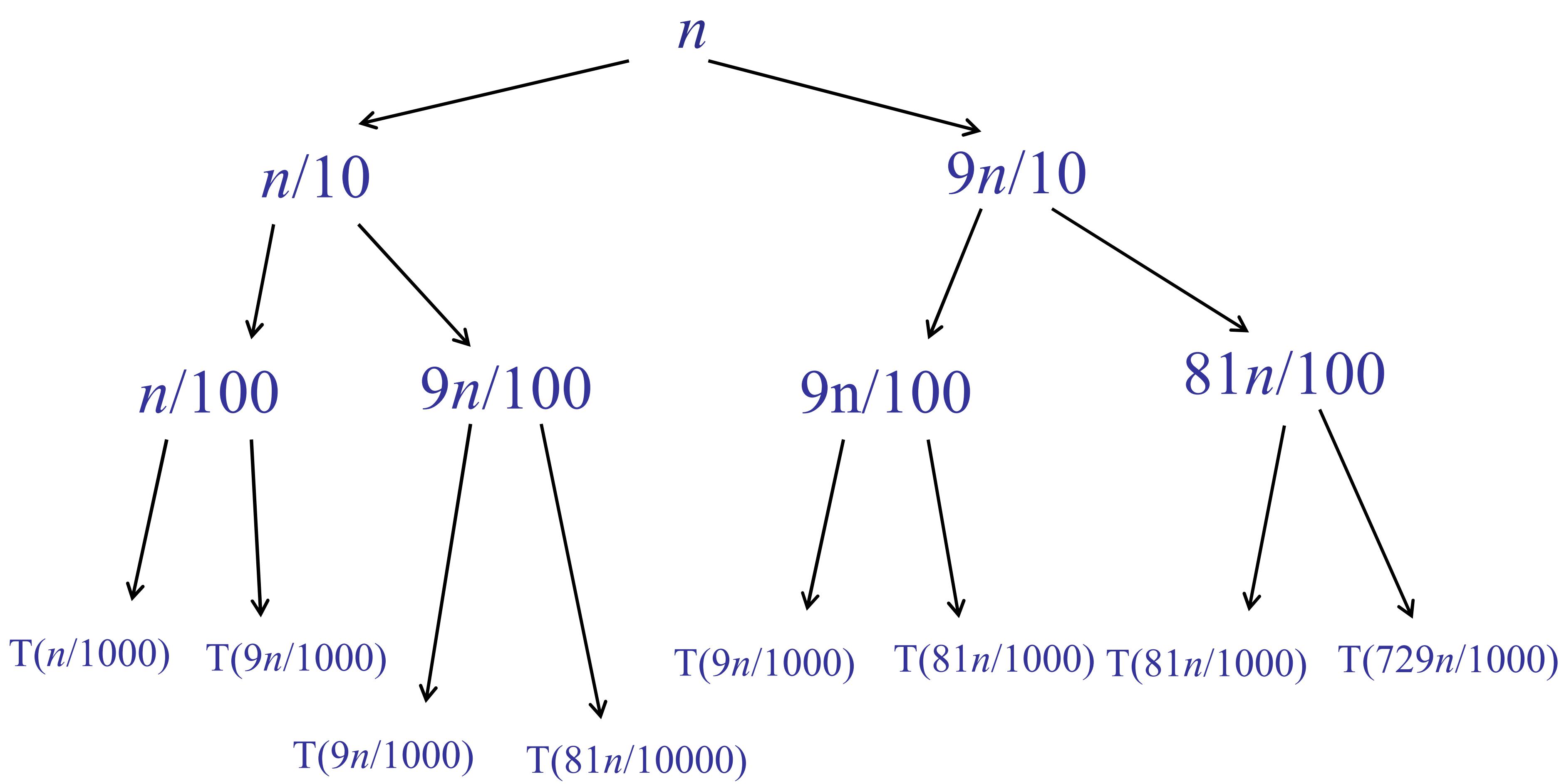
$$n$$

$$n/10 \qquad\qquad 9n/10$$

$$\mathrm{T}(n/100) \quad \mathrm{T}(9n/100) \quad \mathrm{T}(9\mathrm{n}/100) \quad \mathrm{T}(81n/100)$$

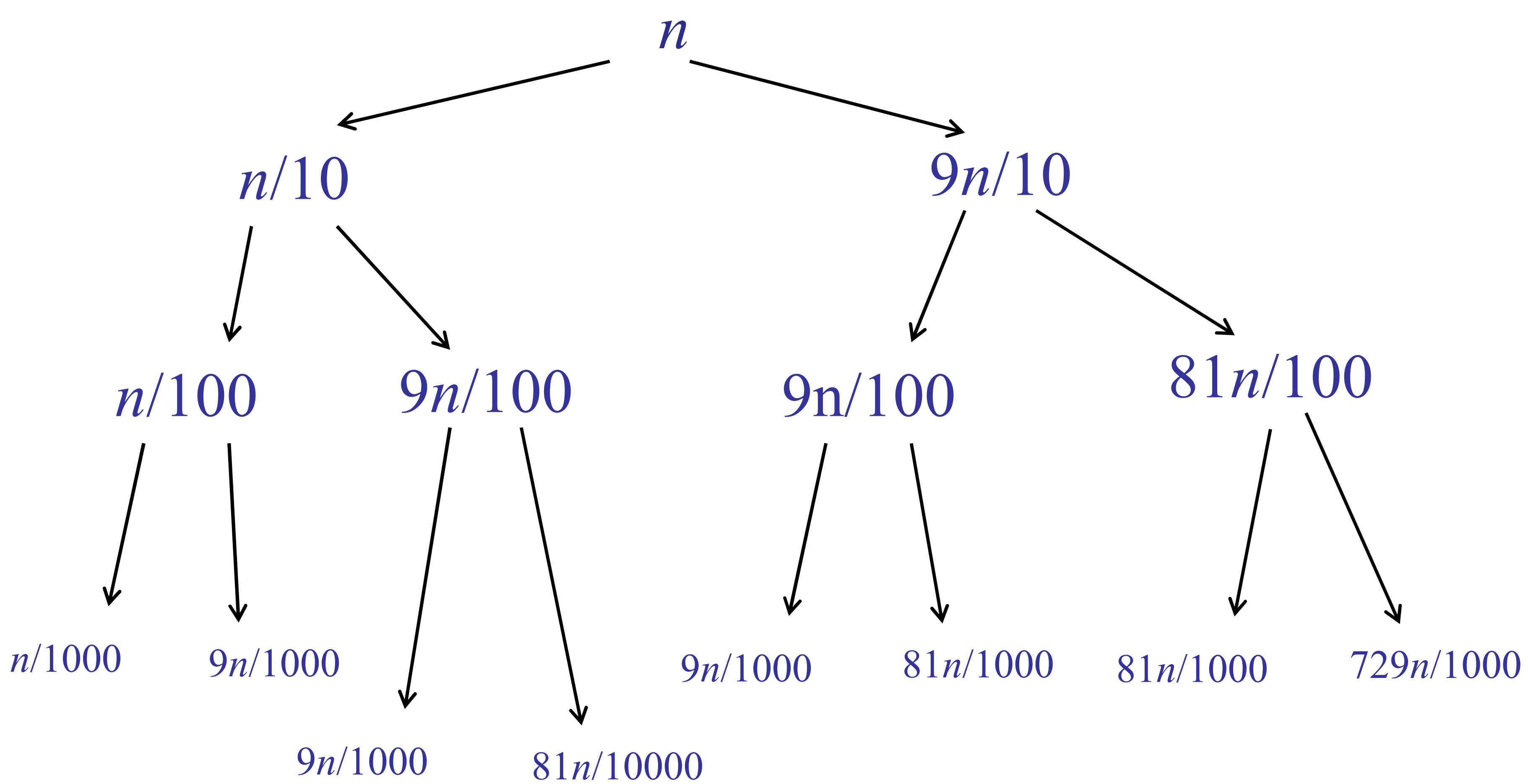
QuickSort Analysis

$k = n/10$

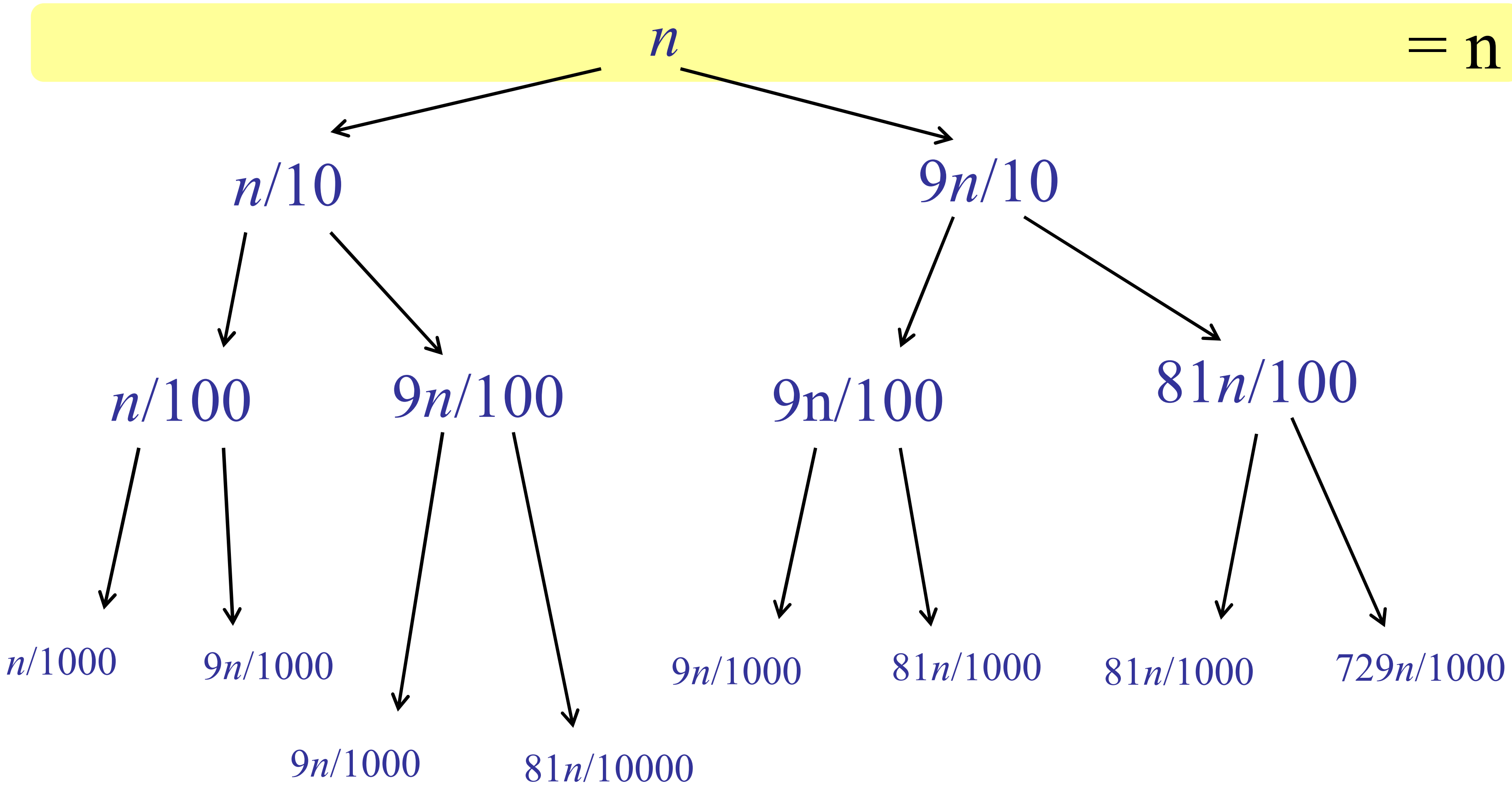


QuickSort Analysis

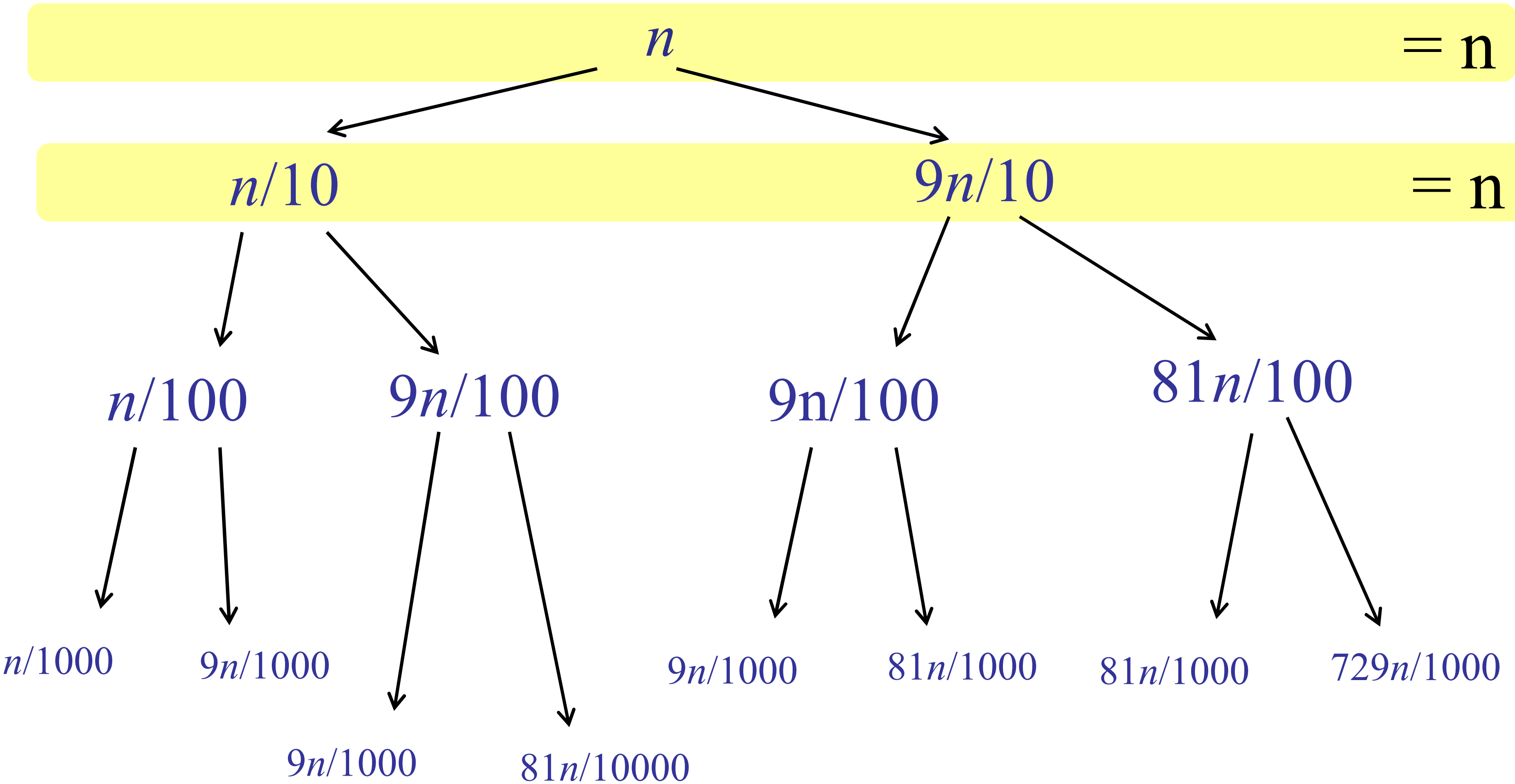
$k = n/10$



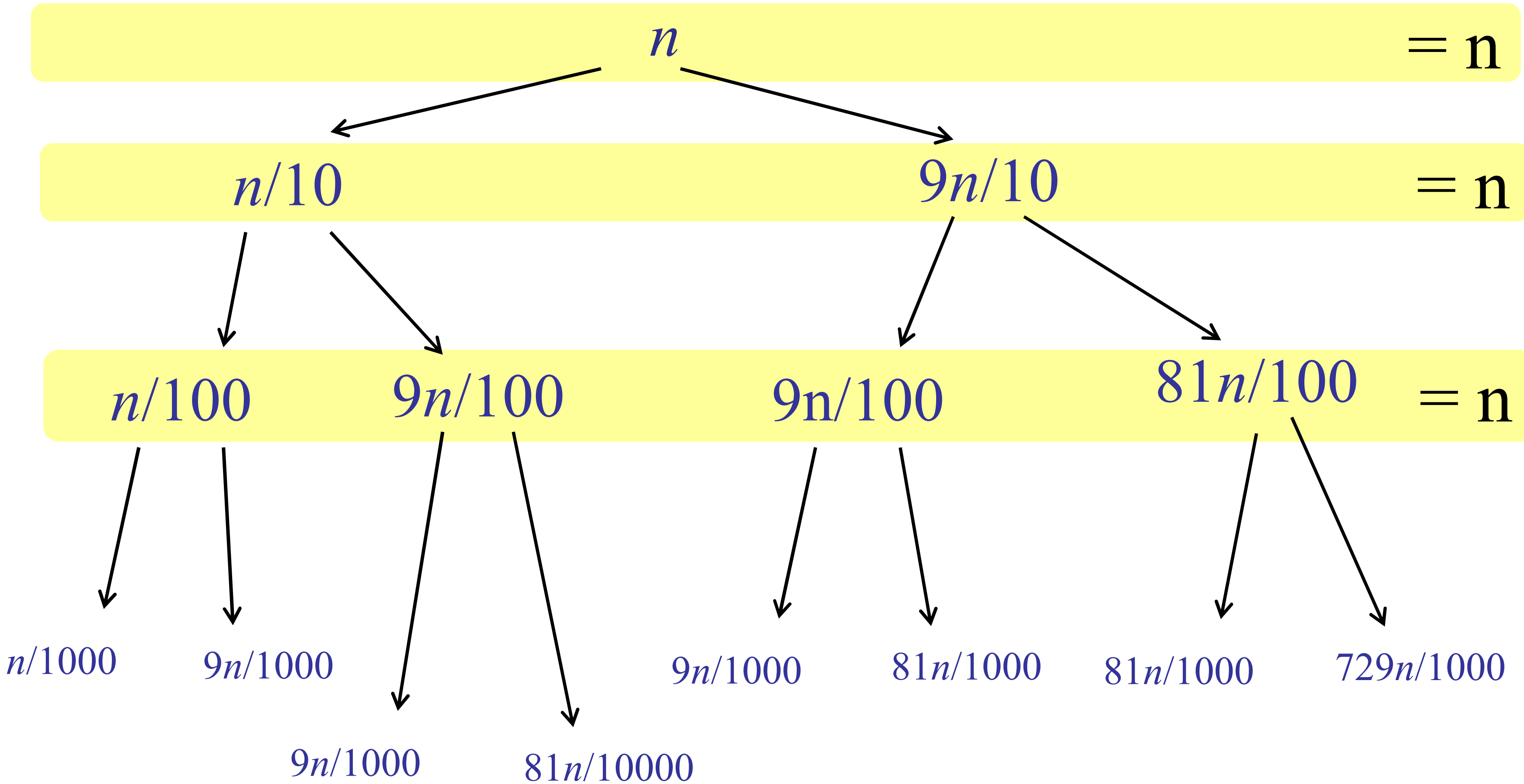
QuickSort Analysis



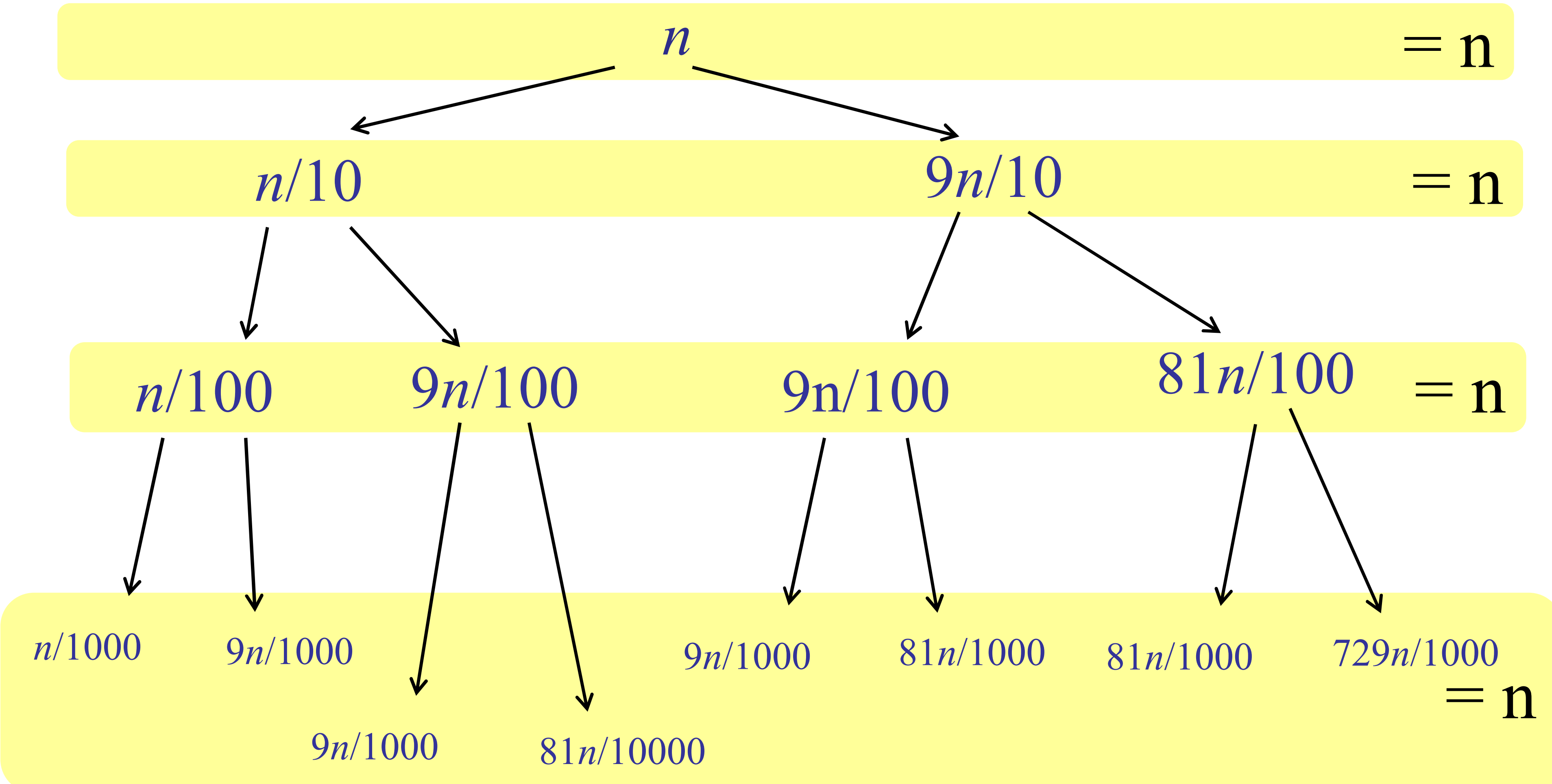
QuickSort Analysis



QuickSort Analysis



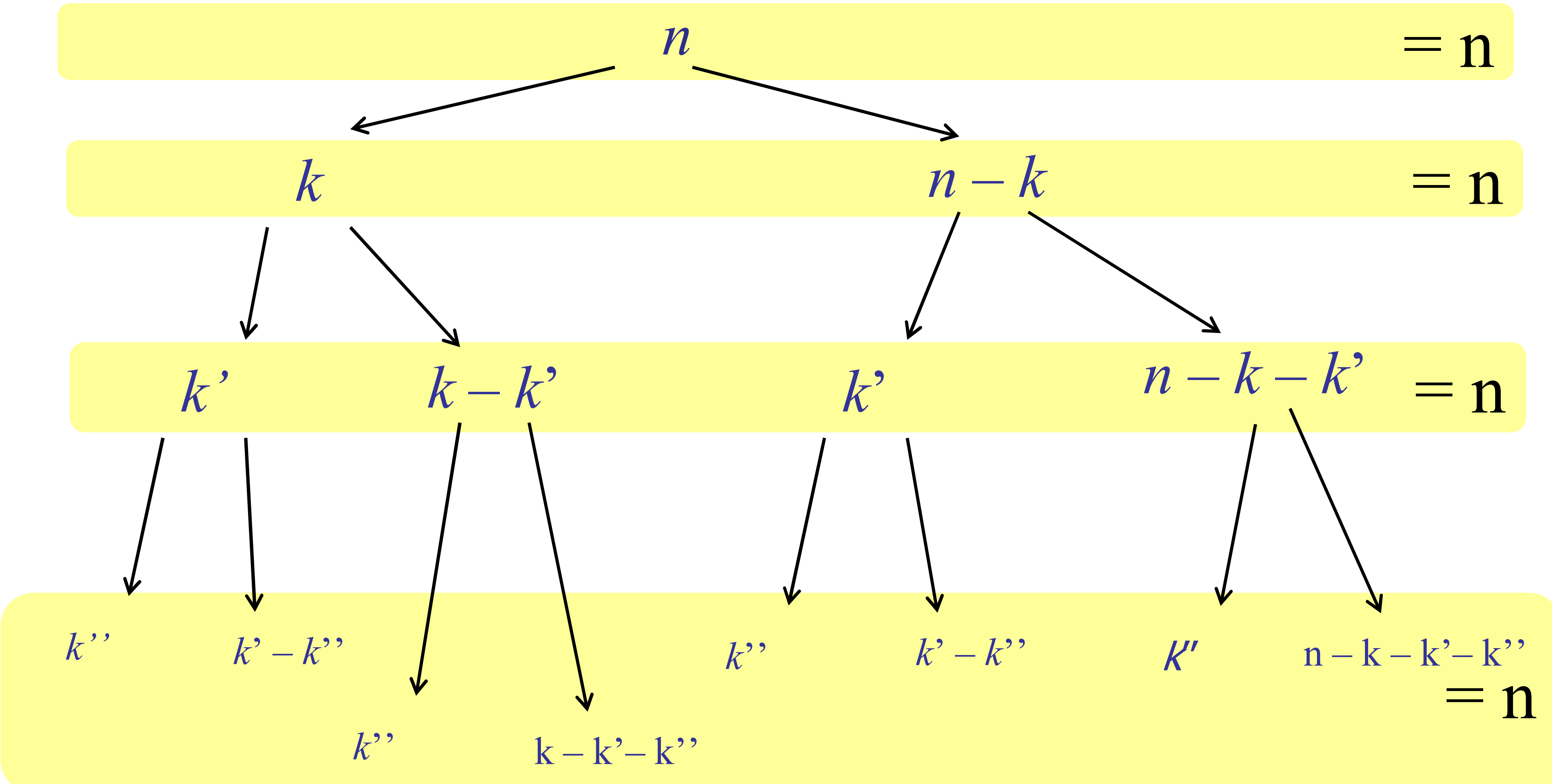
QuickSort Analysis



How many levels??

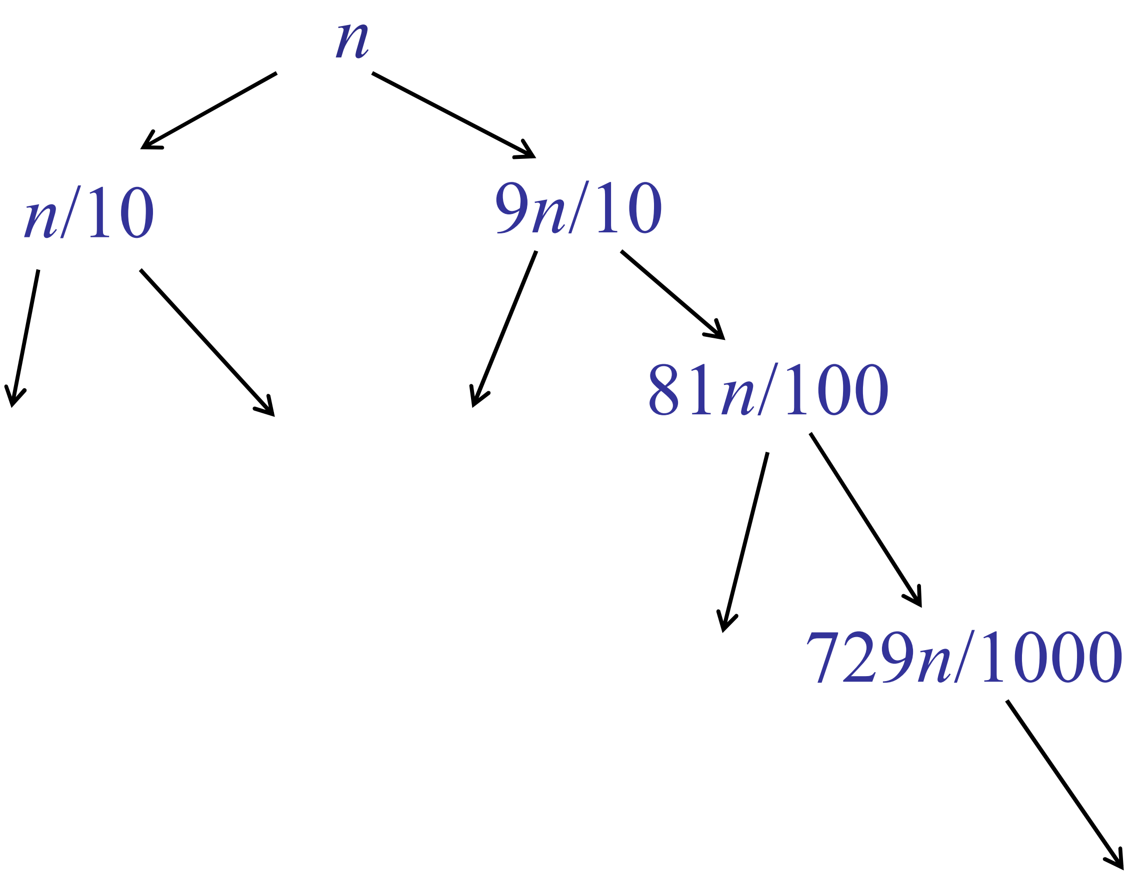
QuickSort Analysis

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



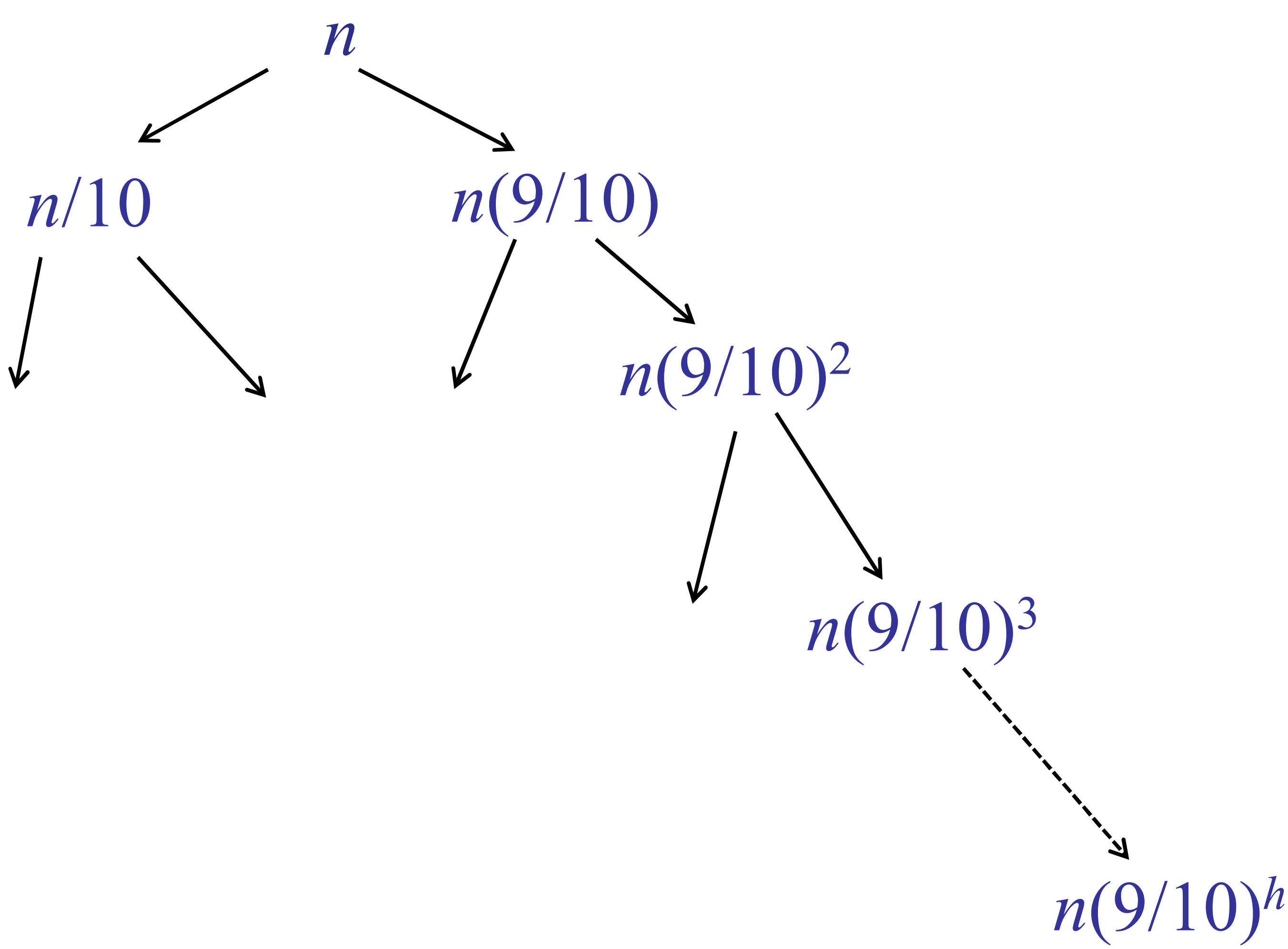
How many levels??

QuickSort Analysis



How many levels??

QuickSort Analysis



How many levels??

QuickSort Analysis

Maximum number of levels:

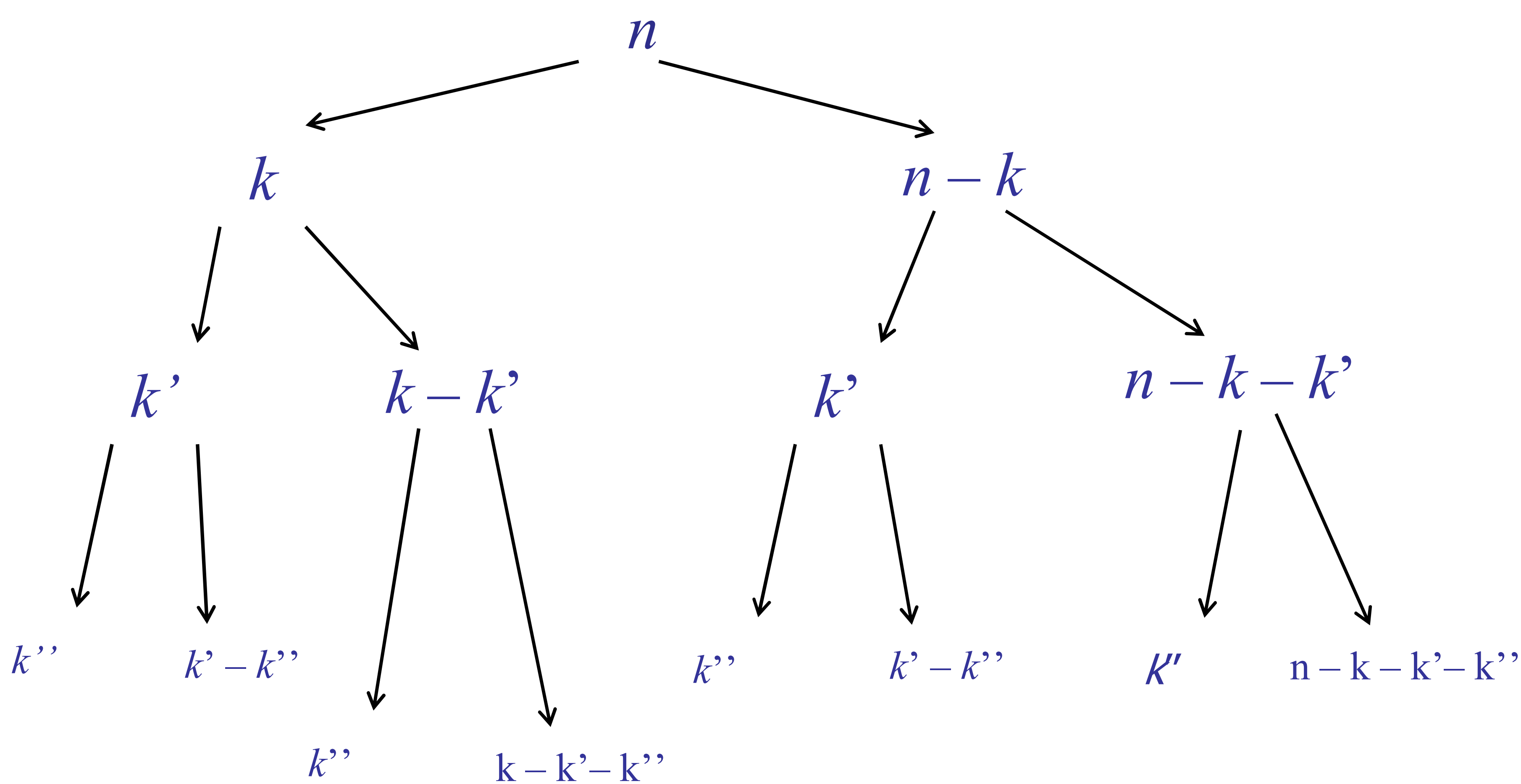
$$1 = n(9/10)^h$$

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

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

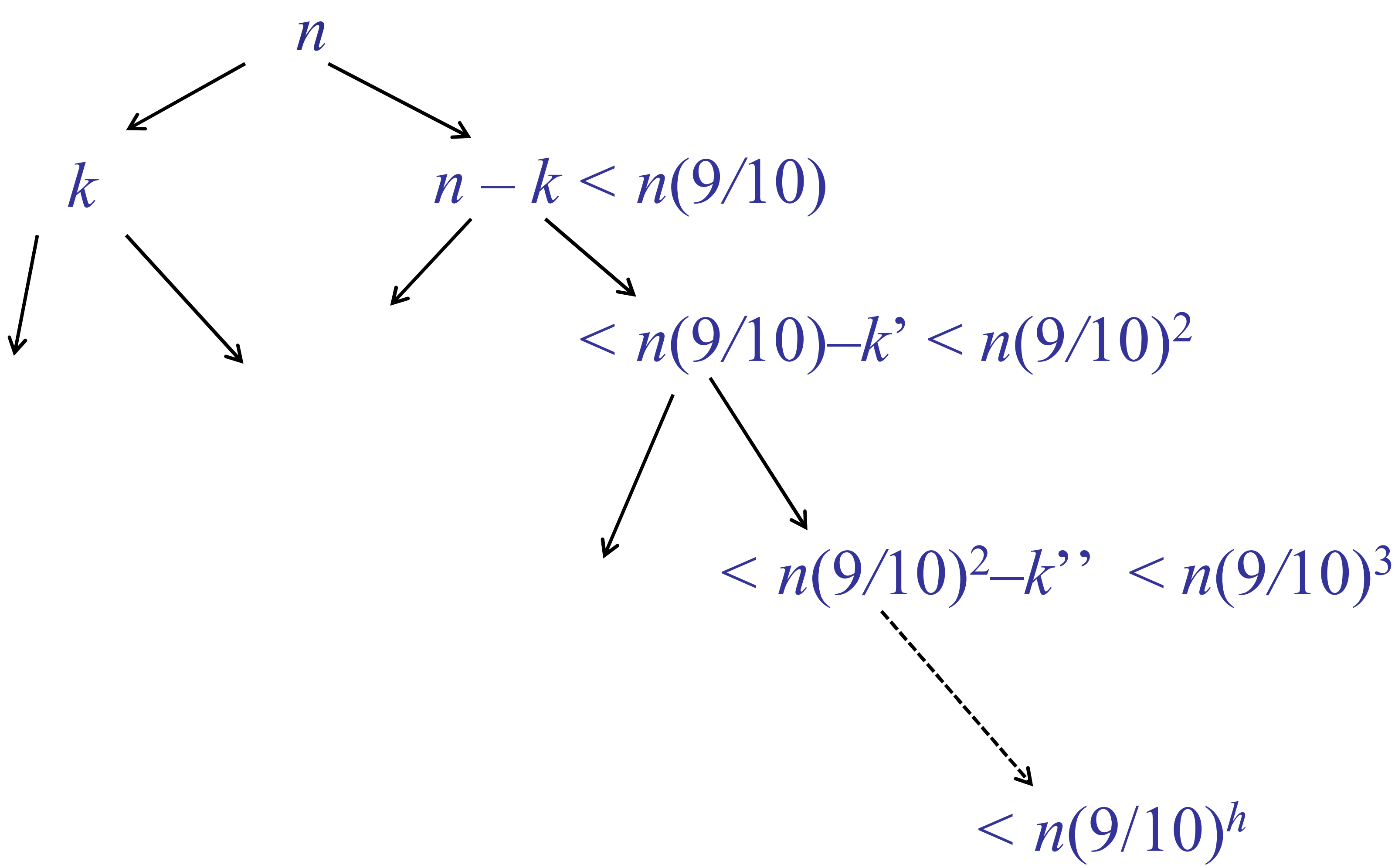
QuickSort Analysis

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



How many levels??

QuickSort Analysis



How many levels??

QuickSort Analysis

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

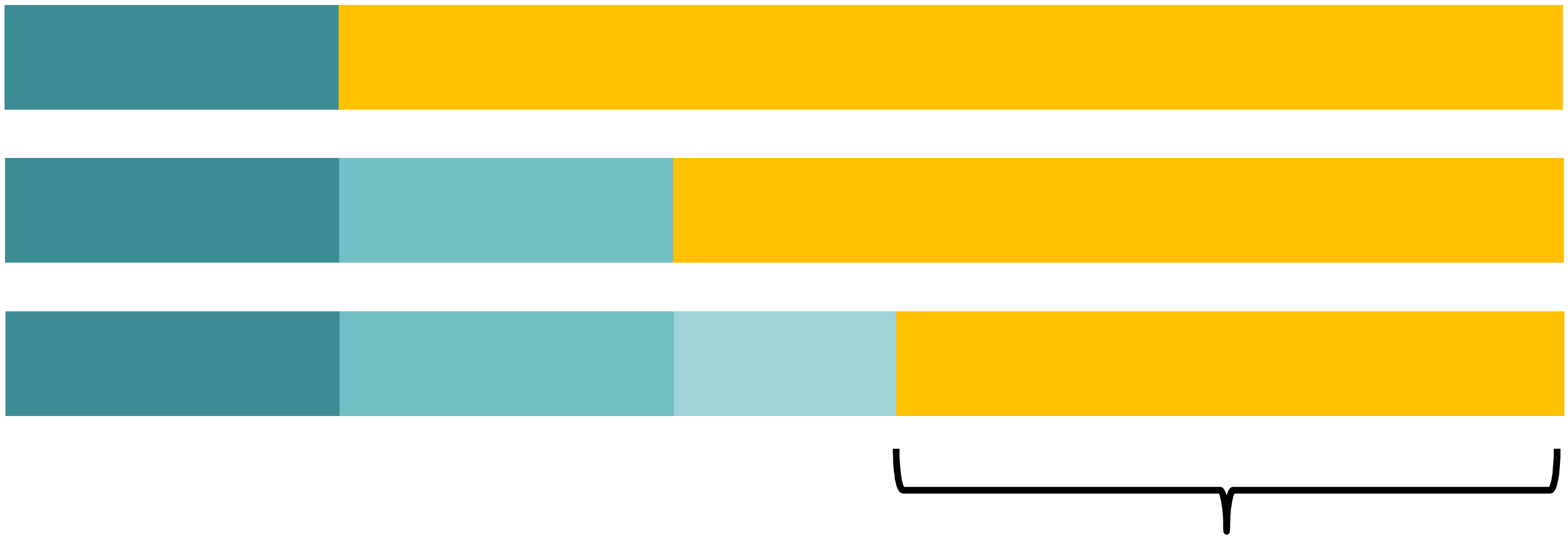
$< n(9/10)$

QuickSort Analysis

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

QuickSort Analysis

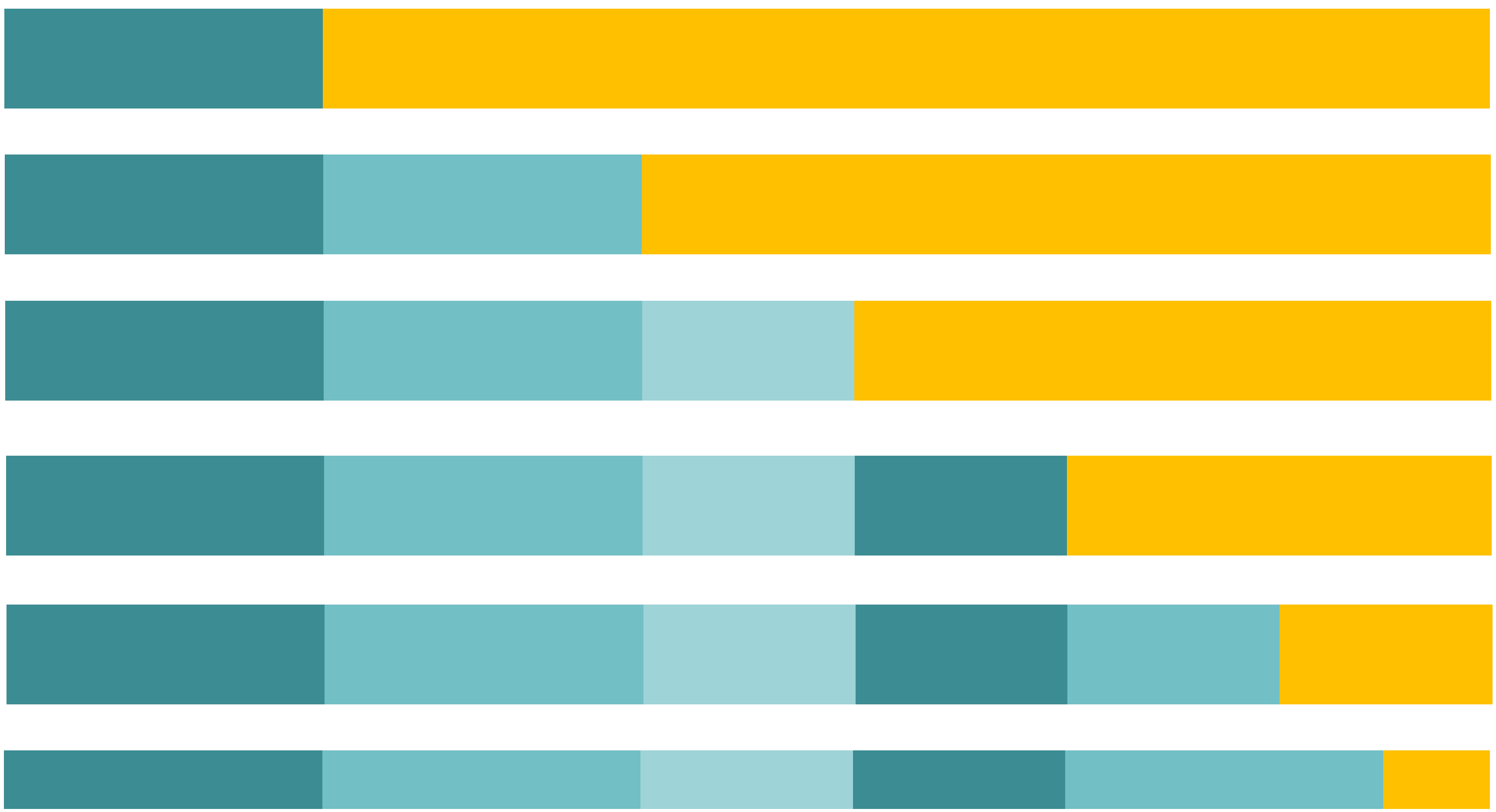
Assume larger part shrinks by at least 9/10 every iteration:



$< n(9/10)^3$

QuickSort Analysis

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



QuickSort Analysis

Maximum number of levels:

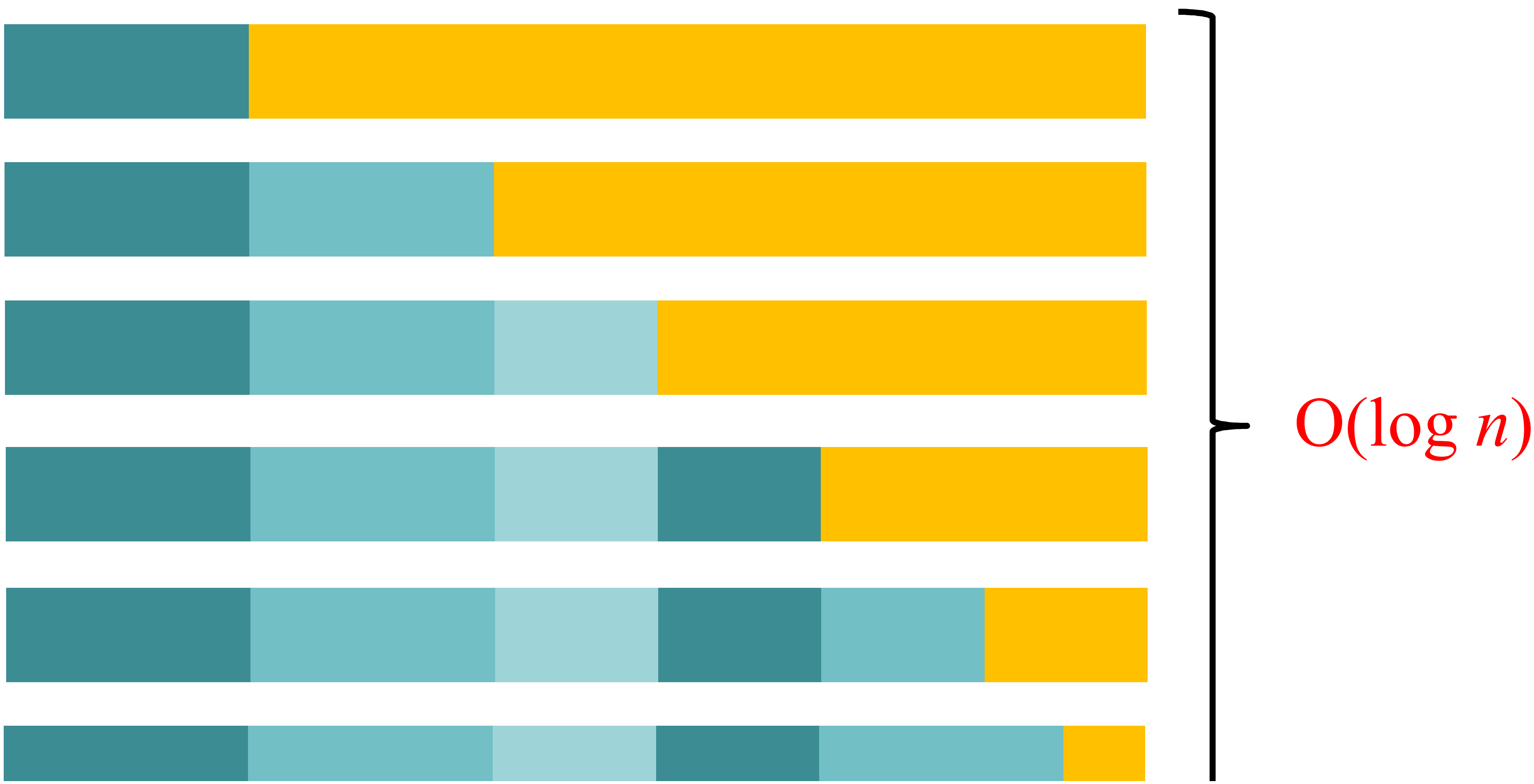
$$1 = n(9/10)^h$$

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

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

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*.
- Assume all random choices are *independent*.
- This is **not** *average case analysis.*

QuickSort

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

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

else

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

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

x = QuickSort(A[1..p–1], p–1)

y = QuickSort(A[p+1..n], n–p)

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..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).
- See CLRS (or talk to me).

Paranoid QuickSort

Key claim:

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

Then we know:

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

$$\text{(where } k > 1/10\text{)}$$

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

Probability Theory

Probability Theory

Flipping a coin:

- $\Pr(\text{heads}) = ½$
- $\Pr(\text{tails}) = ½$

Coin flips are independent:

- $\Pr(\text{heads, heads}) = ½ * ½ = ¼$
- $\Pr(\text{heads, tails, heads}) = ½ * ½ * ½ = 1/8$

Probability Theory

Flipping a coin:

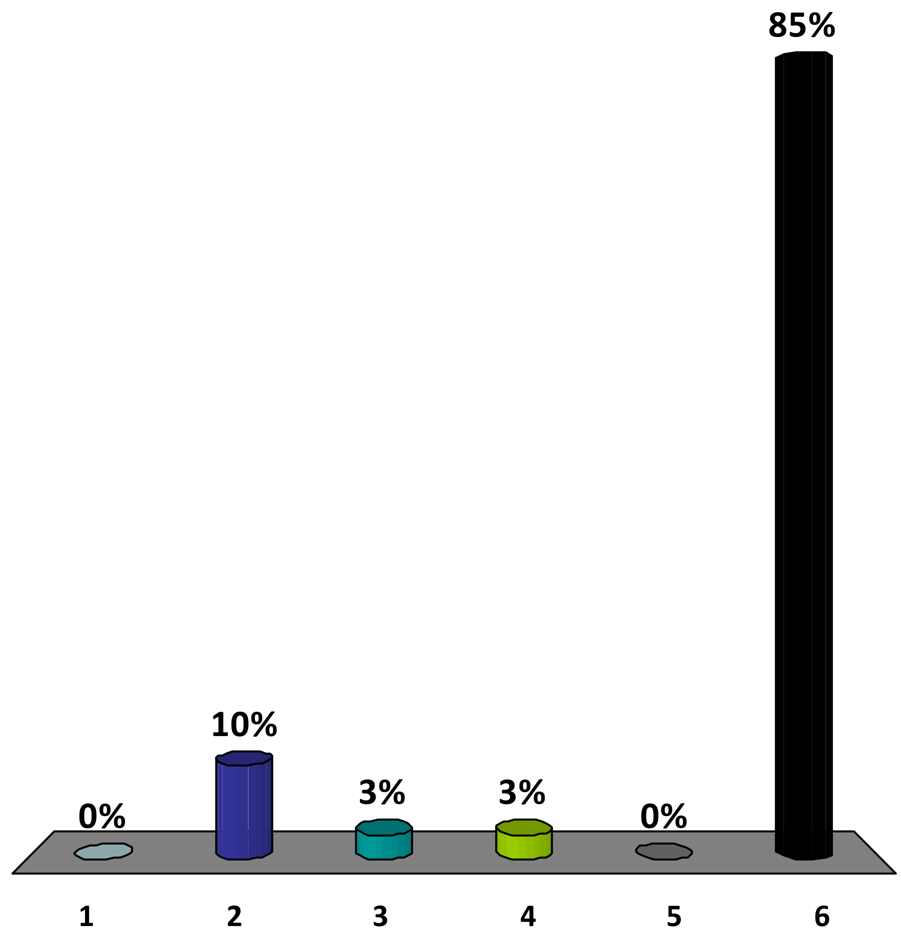
- $Pr(heads) = ½$
- $Pr(tails) = ½$

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

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

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

1. one time
2. two times
3. three times
4. four times
5. ten times
6. Huh??



Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

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

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

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
Weighted average…		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

Probability Theory

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

- $Pr(e_1) = p(e_1)$
- $Pr(e_2) = p(e_2)$
- …
- $Pr(e_k) = p(e_k)$

Expected outcome:

$$E[X] = e_1 * p(e_1) + e_2 * p(e_2) + \ldots + e_k * p(e_k)$$

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 flip) +

(1 – p)*p*(2 flips) +

(1 – p)*(1 – p)*p*(3 flips) +

(1 – p)*(1 – p)*(1 – p)*p*(4 flips) +

…

Probability Theory

Flipping an (unfair) coin:

– Pr(heads) = p

– Pr(tails) = (1 – p)

How many flips to get at least one head?

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

How many <u>more</u> flips to get a head?

Idea: if I flip a head and get a tails, the expected number of flips to get a head now (after one flip) is the same as the expected number of flips before I started.

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1–p)

How many flips to get at least one head?

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

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

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1–p)

How many flips to get at least one head?

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

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

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

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1–p)

How many flips to get at least one head?

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

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

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

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

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

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..n], n–p)

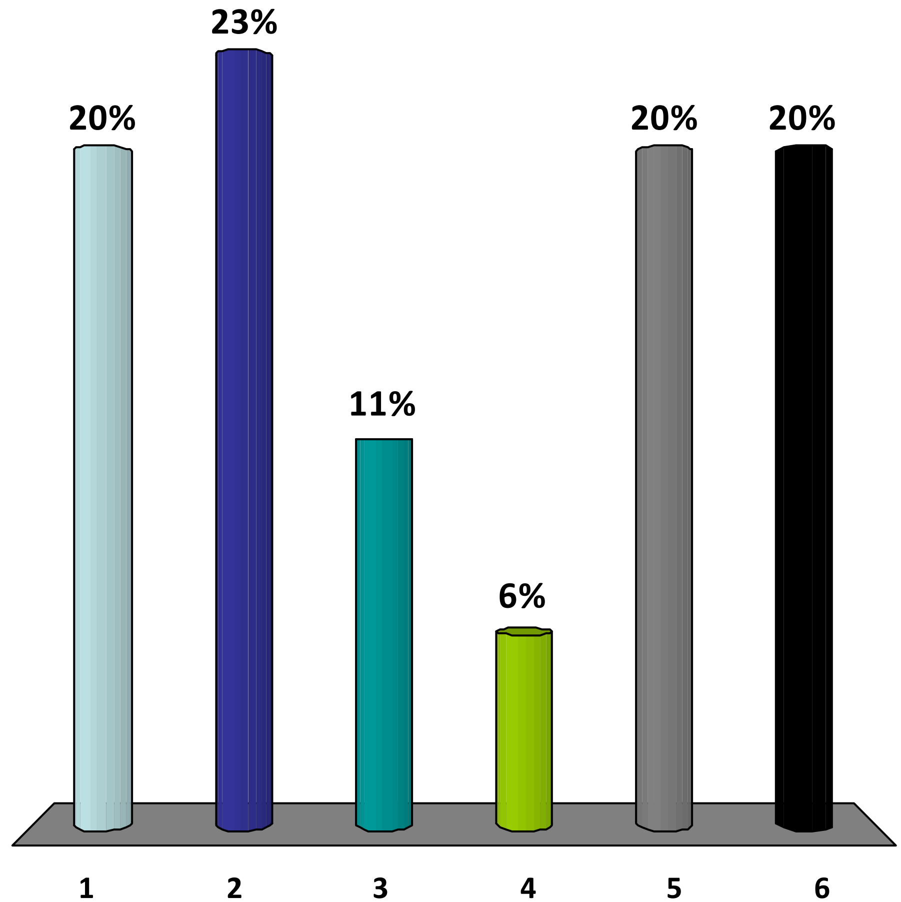
QuickSort Partition

Remember:

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

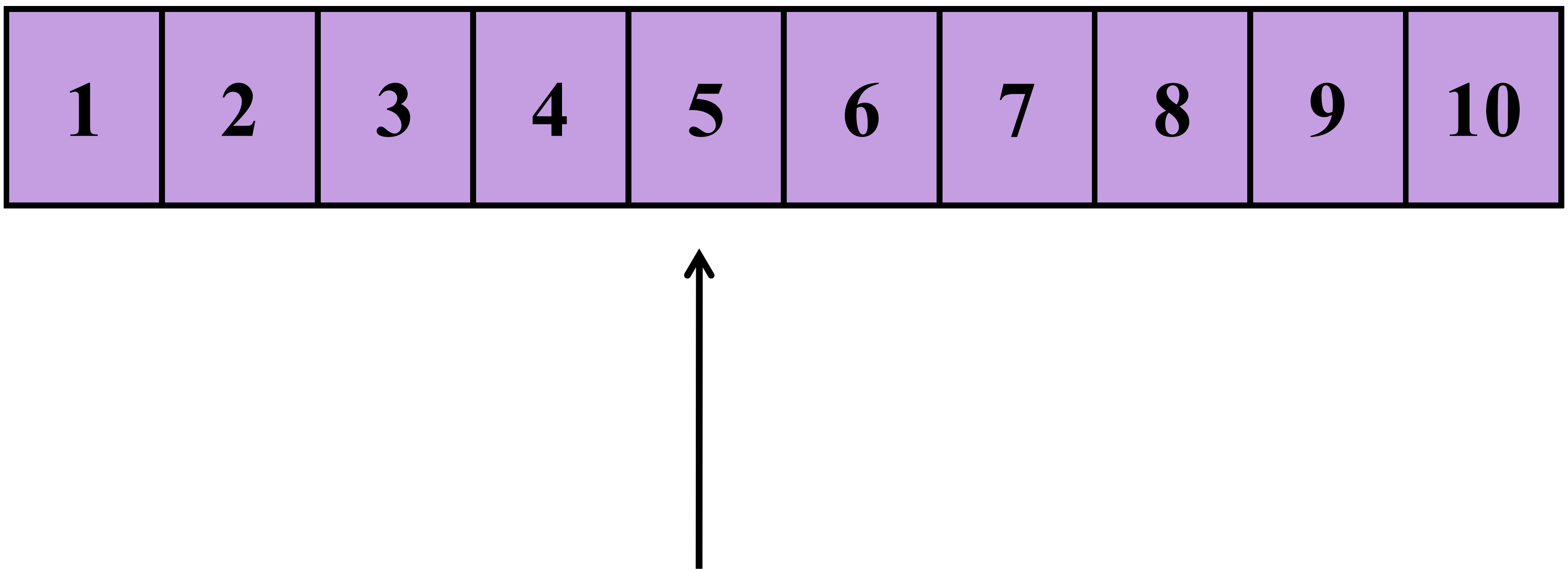
If we choose a pivot at random, what is the probability that it is <u>good</u>?

1. 1/10
2. 2/10
3. ½
4. 1/log(n)
5. 1/n
6. I have no idea.



Choosing a Good Pivot

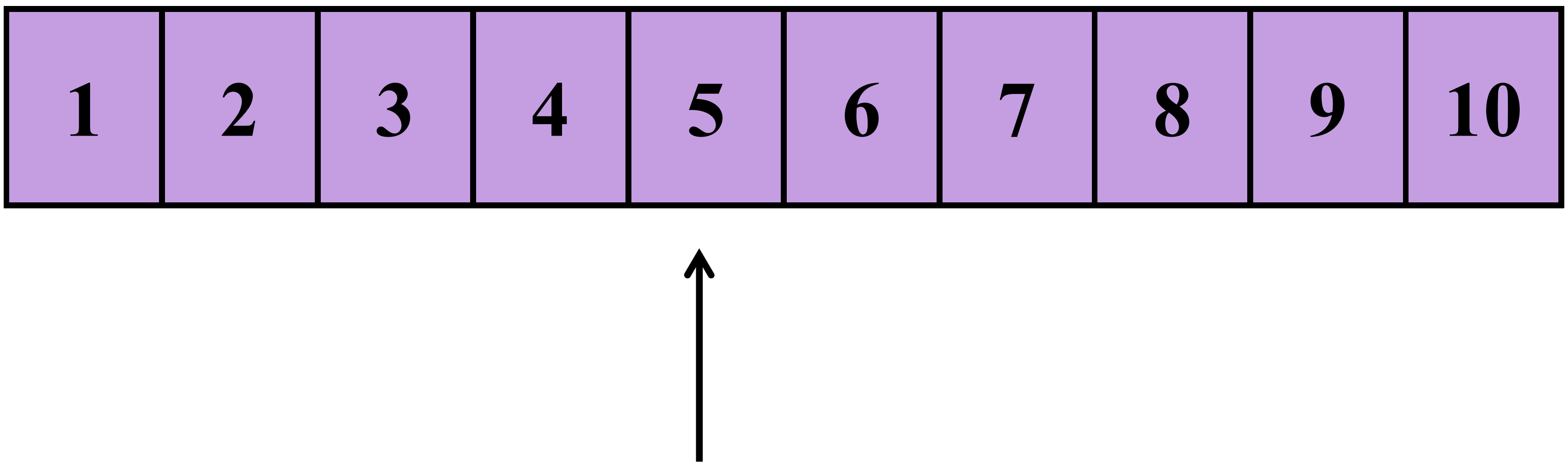
Imagine the array divided into 10 pieces:



Choose a random point at which to partition.

Choosing a Good Pivot

Imagine the array divided into 10 pieces:

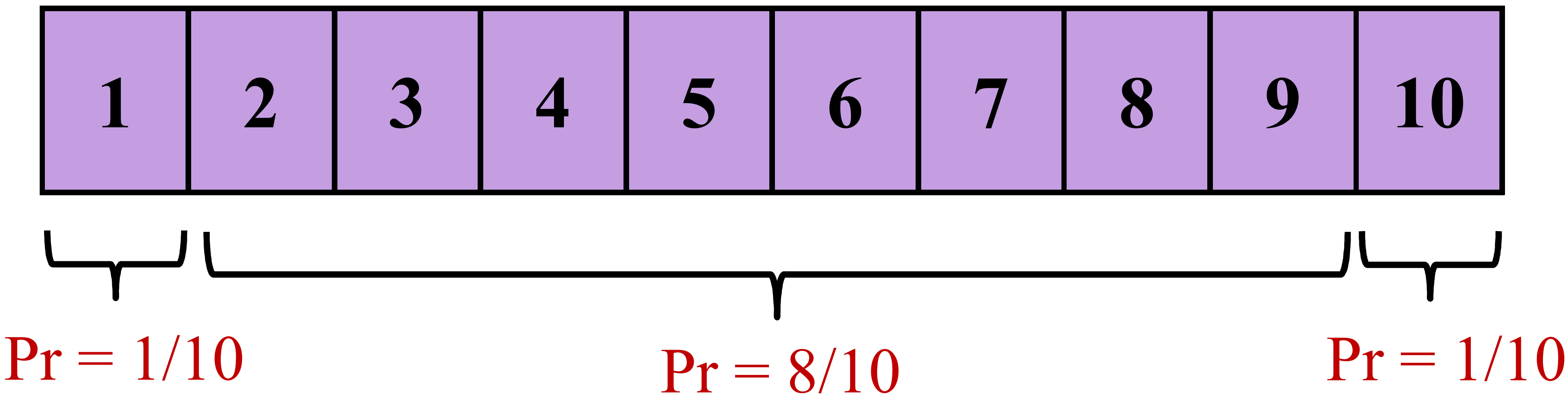


Choose a random point at which to partition.

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

Choosing a Good Pivot

Imagine the array divided into 10 pieces:

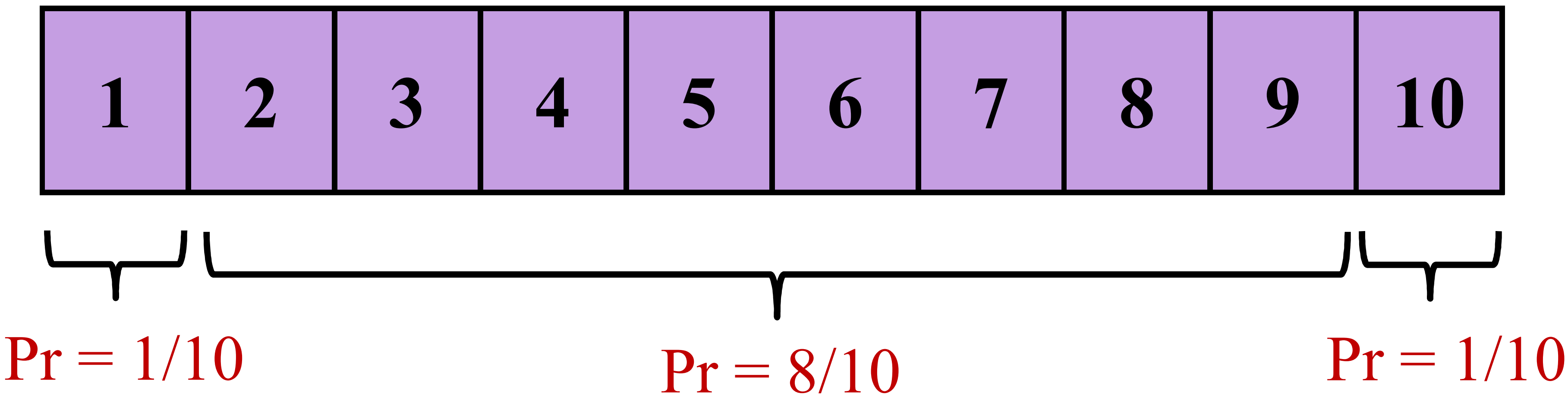


Choose a random point at which to partition.

- 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..n], n–p)

Paranoid QuickSort

Key claim:

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

Then we know:

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

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

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

QuickSort Optimizations

Many, many optimizations and variants:

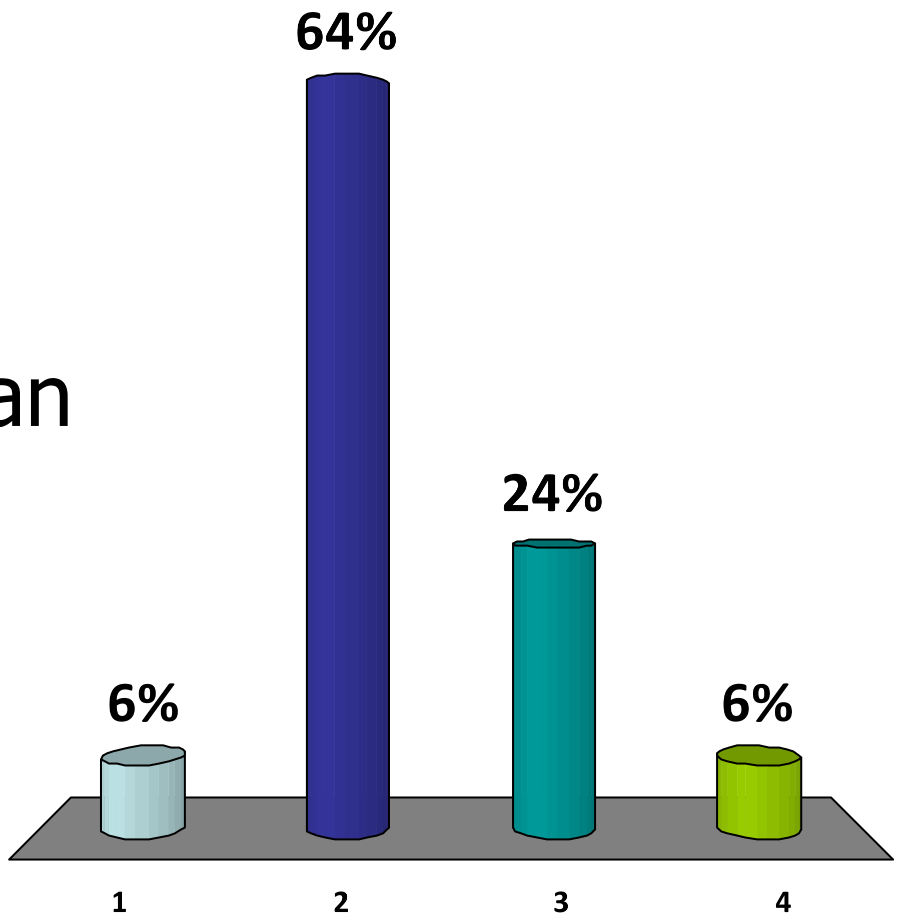
1. To save space, recurse into smaller half first.
   - Only need O(log n) extra space.

2. For small arrays, use InsertionSort.
   - Stop recursion at arrays of size MinQuickSort.
   - Do one InsertionSort on full array when done.

3. If array contains repeated keys, be careful!

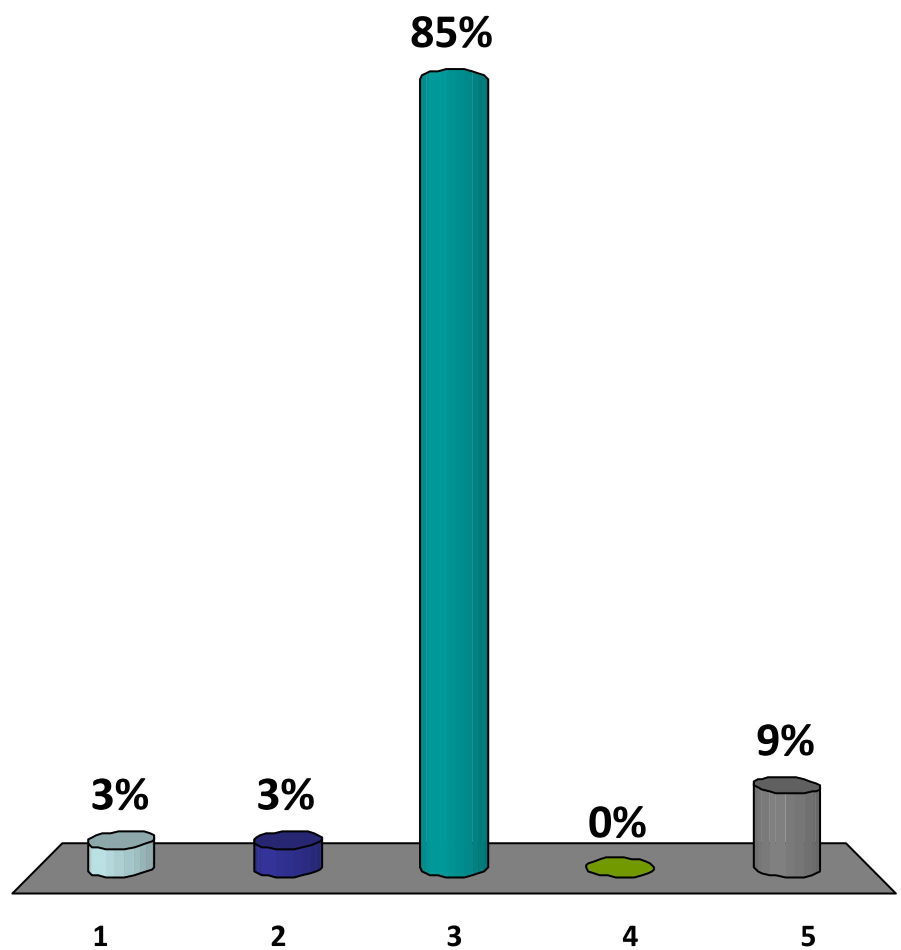
Which of the following is most important for QuickSort to be efficient?

1. A good memory manager.
2. An efficient partition implementation.
3. A deterministic median implementation.
4. A work-efficient scheduler.



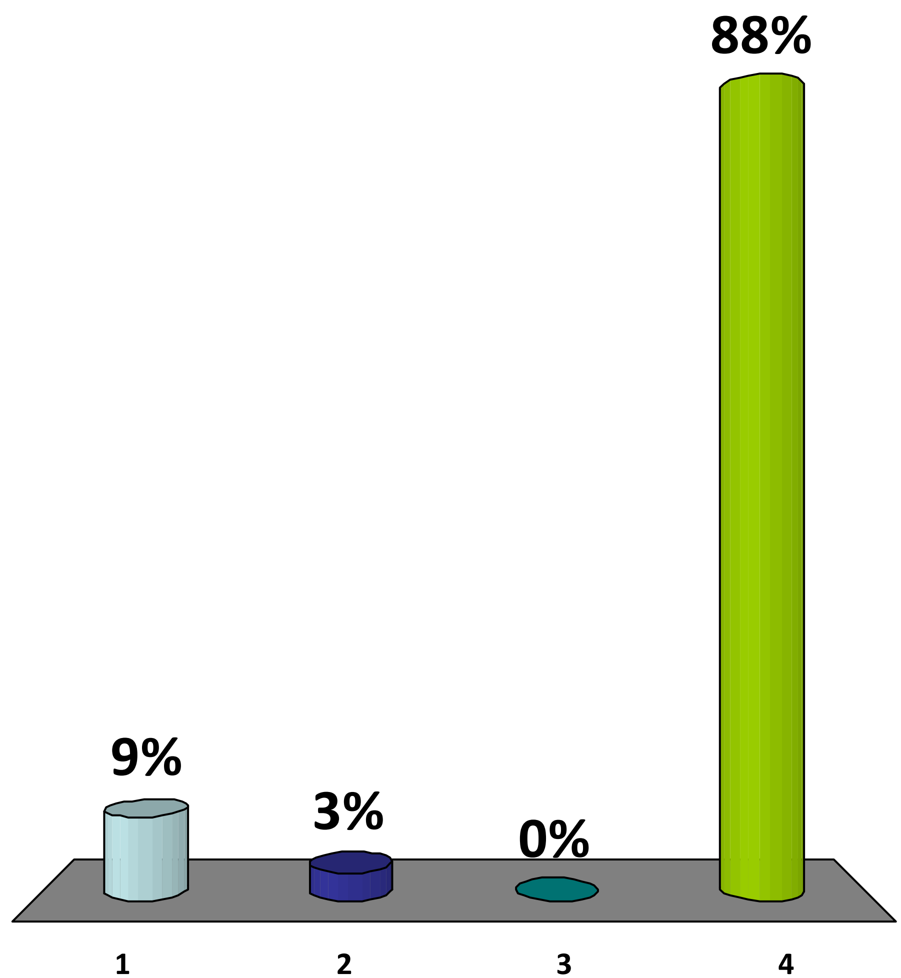
Which of the following is **not** true of the partition algorithm?

1. It is in-place.
2. It runs in O(n) times.
3. ✓ It uses 2n space.
4. It relies on the choice of a good pivot.
5. It is not stable.



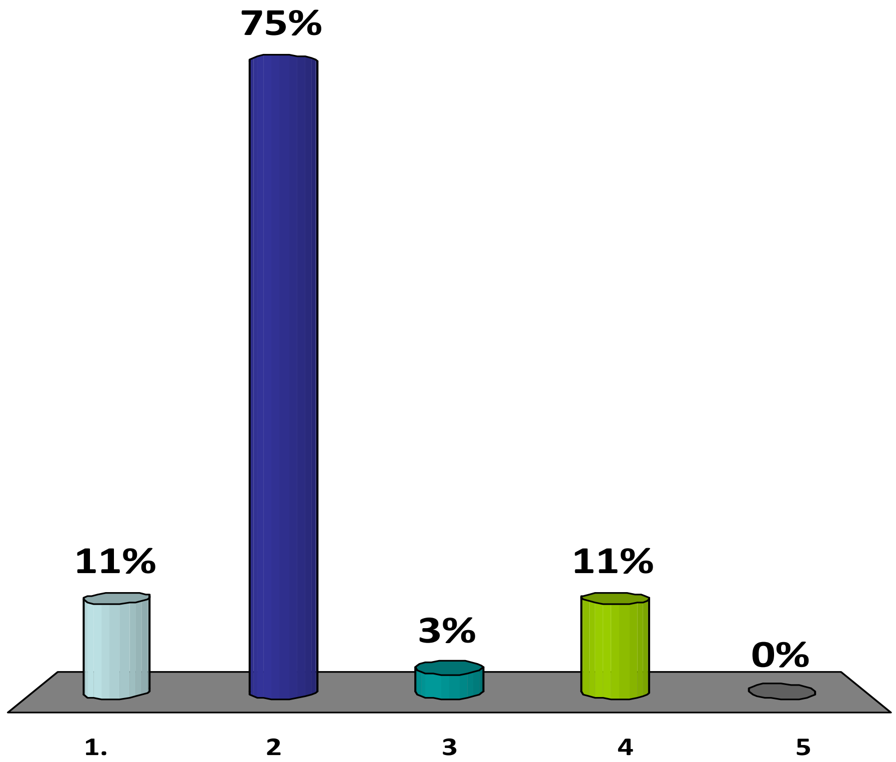
If the pivot is chosen to be A[1], which of the following has the worst running time?

1. [1, 2, 3, 4, 5, 6, 7]
2. [1, 3, 2, 4, 5, 7, 6]
3. [2, 4, 6, 1, 3, 5, 7]
4. [7, 6, 5, 4, 3, 2, 1]



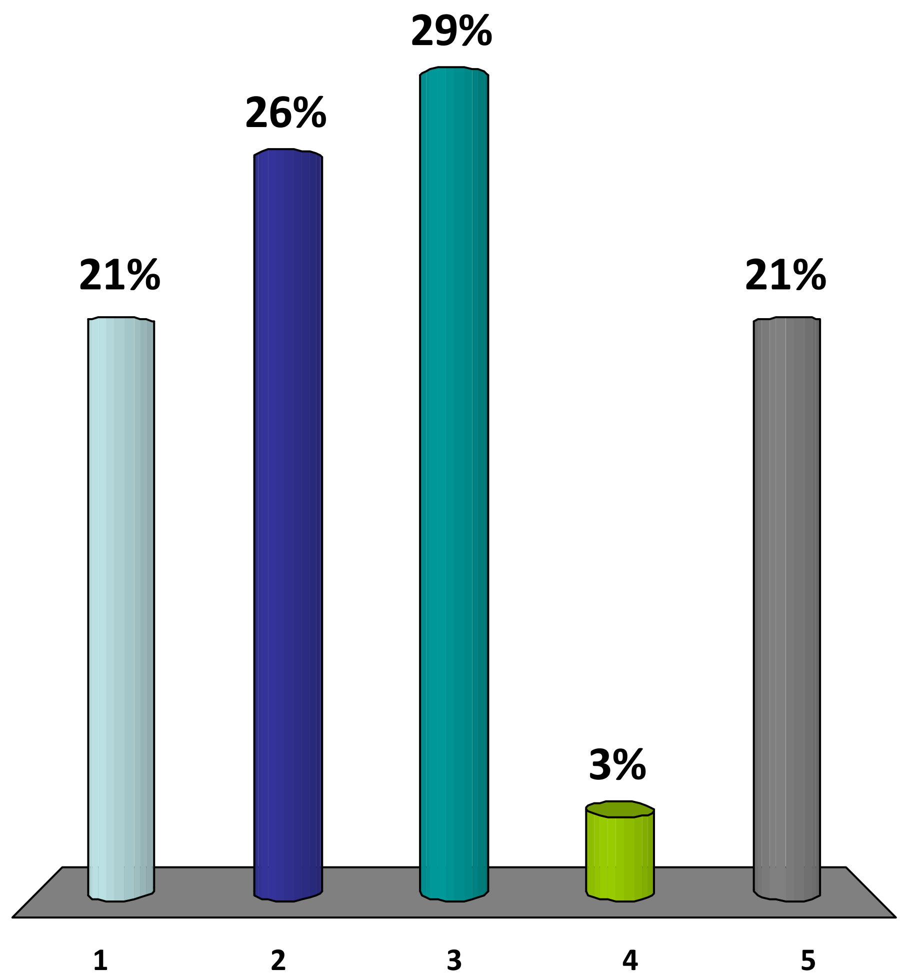
If the pivot is chosen at random, what is the expected number of times to partition before choosing a pivot that partitions the array into a: 1/4 : 3/4 split

1. 1.67
2. 2
3. 3
4. 4
5. 5



Which of the following helps to explain why QuickSort is faster than other sorting algorithms:

1. It is asymptotically faster.
2. It is randomized.
3. It is in-place.
4. It is easier to implement.
5. None of the above.



Summary

QuickSort:

- How to partition an array in O(n) times.
- How to choose a good pivot.
- Paranoid QuickSort.
- Randomized analysis.