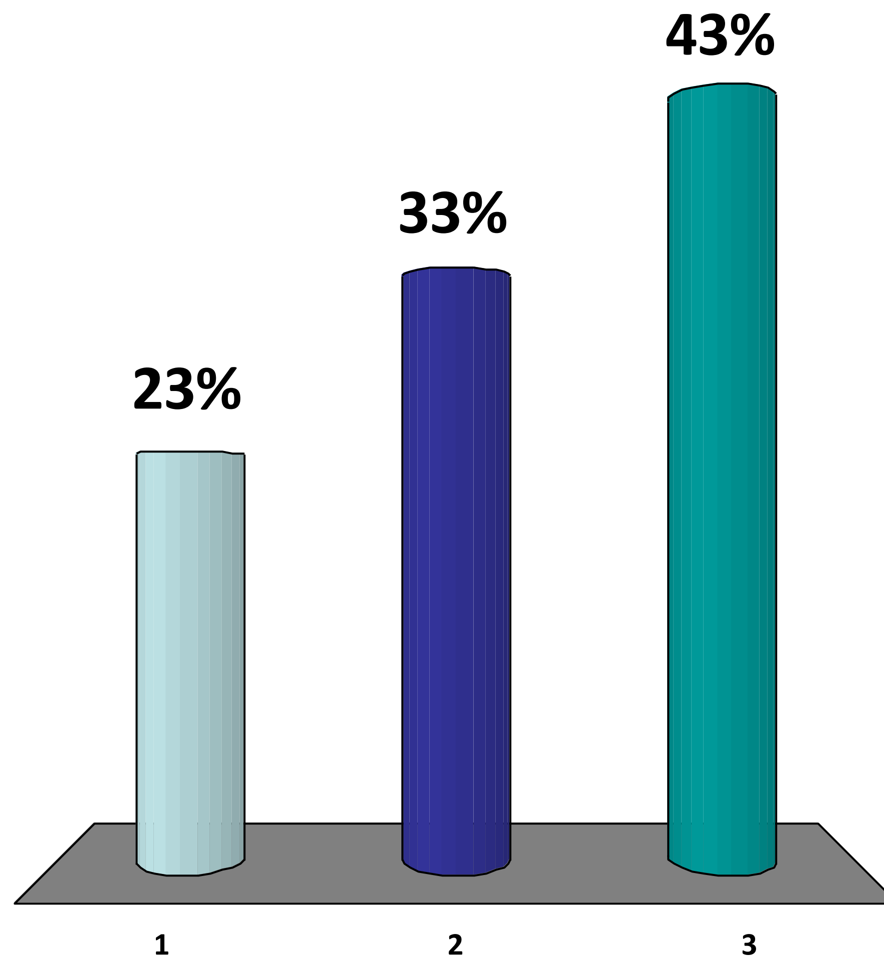
CS2020
Data Structures and Algorithms

Welcome!

Did you remember your clicker?

1. Of course, silly.
2. I remembered it yesterday.
3. Horses?

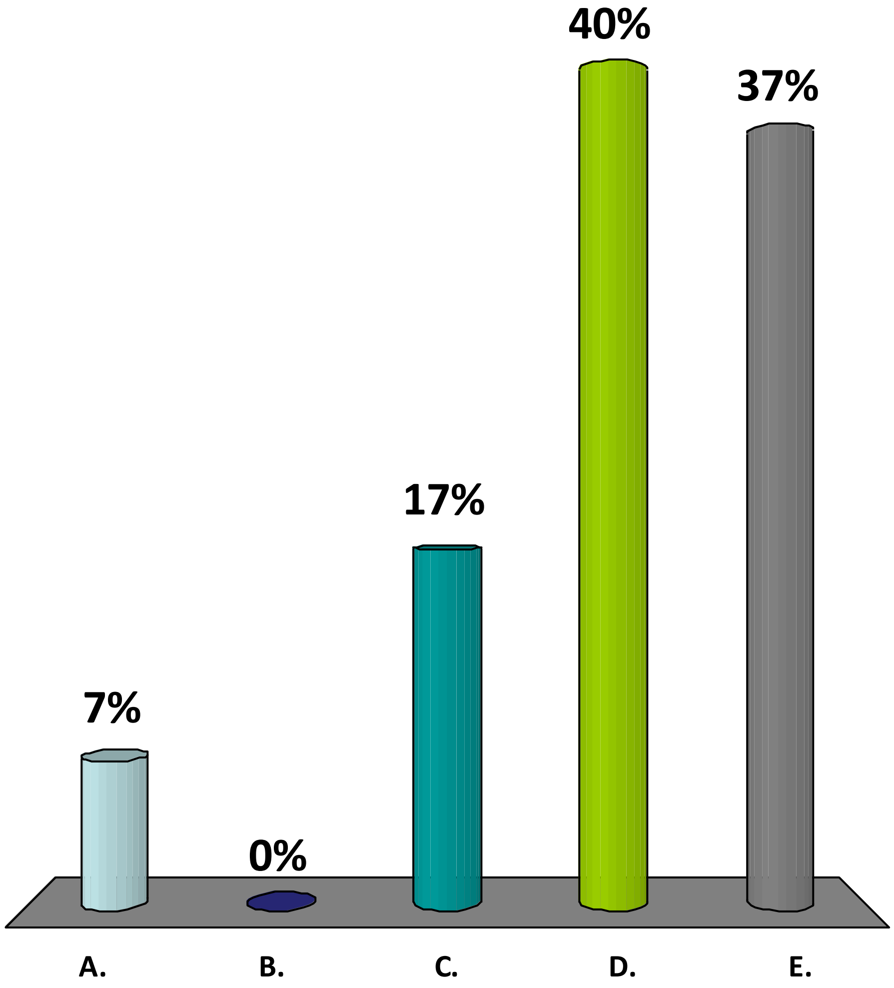


Last Time: Sorting, Part I

- Sorting algorithms
  - BubbleSort
  - SelectionSort
  - InsertionSort
  - MergeSort
- Properties
  - Running time
  - Space usage
  - Stability

What is your favorite character in Harry Potter?

A. Harry Potter
B. Ron Weasley
C. Hermione Granger
D. The Sorting Hat
E. Bilbo Baggins



My favorite Harry Potter character was the Sorting Hat.
His job was to learn people's secrets and then judge them.

Comparable Interface

x.compareTo(y) :

- -1: if (x<y)
- 0: if (x == y)
- 1: if (x>y)

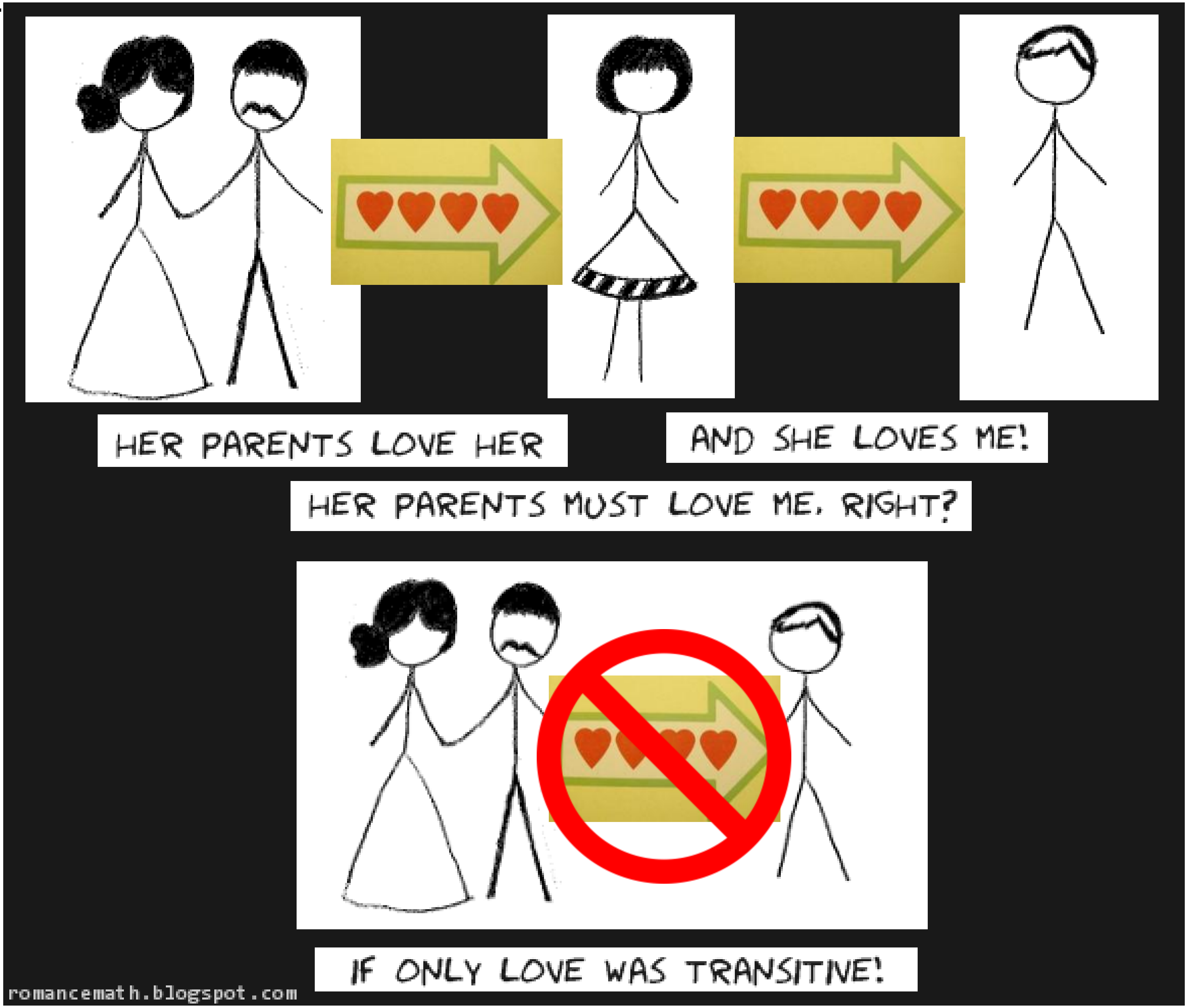
Must define a total ordering
Must be transitive.

```
interface Comparable<TypeA> {

    int compareTo(TypeA other);

}
```

Non-Transitive Relationship



Sorting Students

```
class Student implements Comparable<Student> {
...
...
}
```

Generic Sorting

```
public interface ISort{

    public <TypeA extends Comparable<TypeA>>
    void sort(TypeA[] dataArray);

}
```

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:

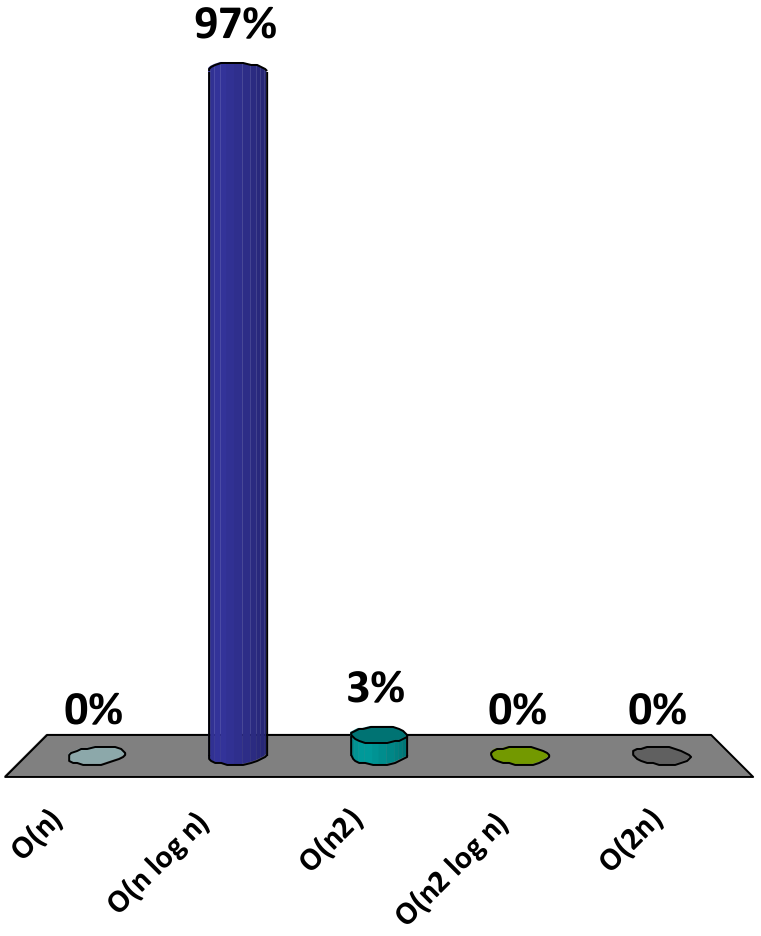
$B[1] \leq B[2] \leq \ldots \leq B[n]$

Example:

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

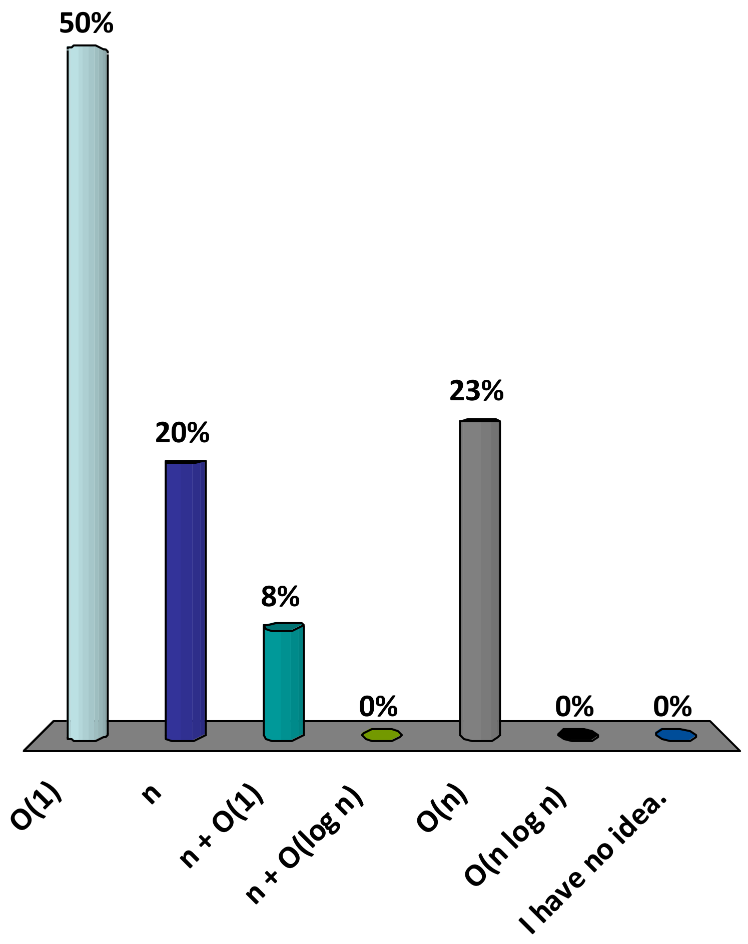
What is the running time of MergeSort on a sorted list?

a. O(n)
b. O(n log n)
c. $O(n^2)$
d. $O(n^2 \log n)$
e. $O(2^n)$



How much space does InsertionSort use to sort a list of n items?

a. O(1)
b. n
c. n + O(1)
d. n + O(log n)
e. O(n)
f. O(n log n)
g. I have no idea.



InsertionSort

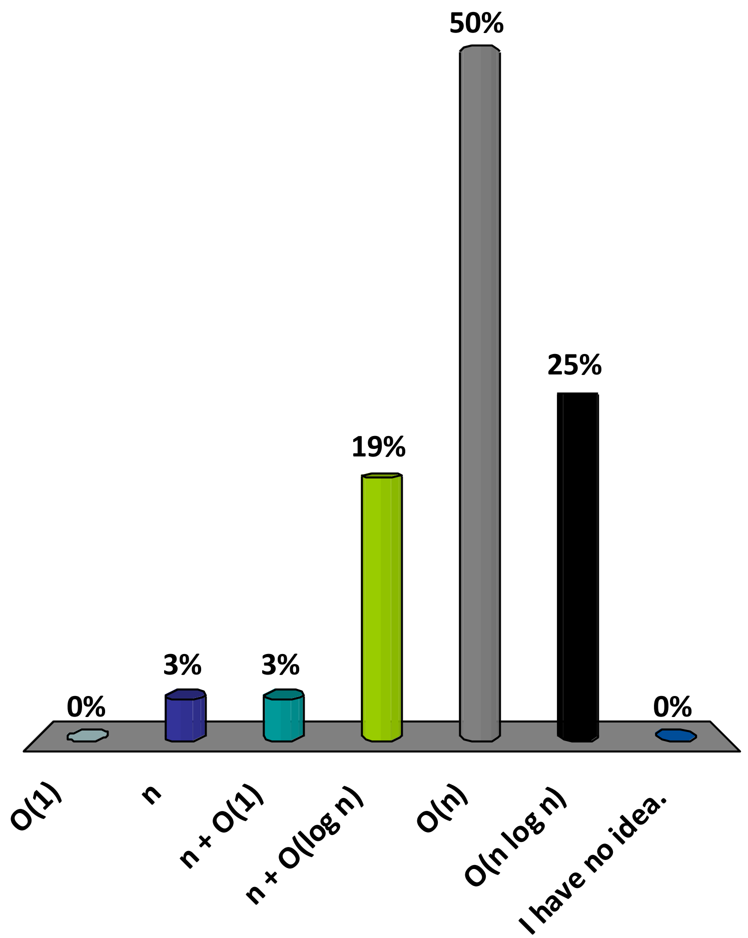
```
InsertionSort(A, n)
    for j ← 2 to n
        key ← A[j]
        i ← j-1
        while (i > 0) and (A[i] > key)
            A[i+1] ← A[i]
            i ← i-1
        A[i+1] ← key
```

array of size n

3 integers

How much space does MergeSort use to sort a list of n items?

a. O(1)
b. n
c. n + O(1)
d. n + O(log n)
e. O(n)
f. O(n log n)
g. I have no idea.



MergeSort Space Analysis

Let $S(n)$ be the worst-case space for sorting an array of n elements.

MergeSort(A, n)

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

else:

X ←Merge-Sort**(**…**);** $n/2 + S(n/2)$

Y ←Merge-Sort**(**…**);** $n/2 + S(n/2)$

return Merge **(**A, X ,Y, n/2**);**

MergeSort Space Analysis

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

MergeSort(A, n)

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

else:

X $\leftarrow$MergeSort**(**...**);**

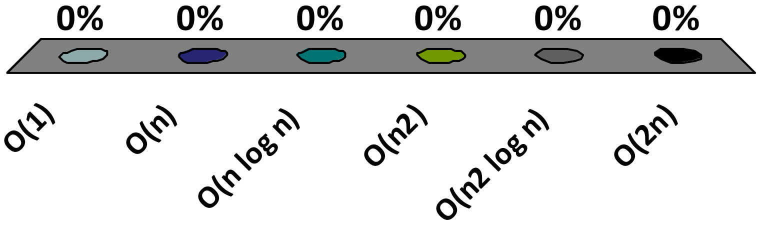
Y $\leftarrow$MergeSort**(**...**);**

return Merge **(**A, X, Y, n/2**);**

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

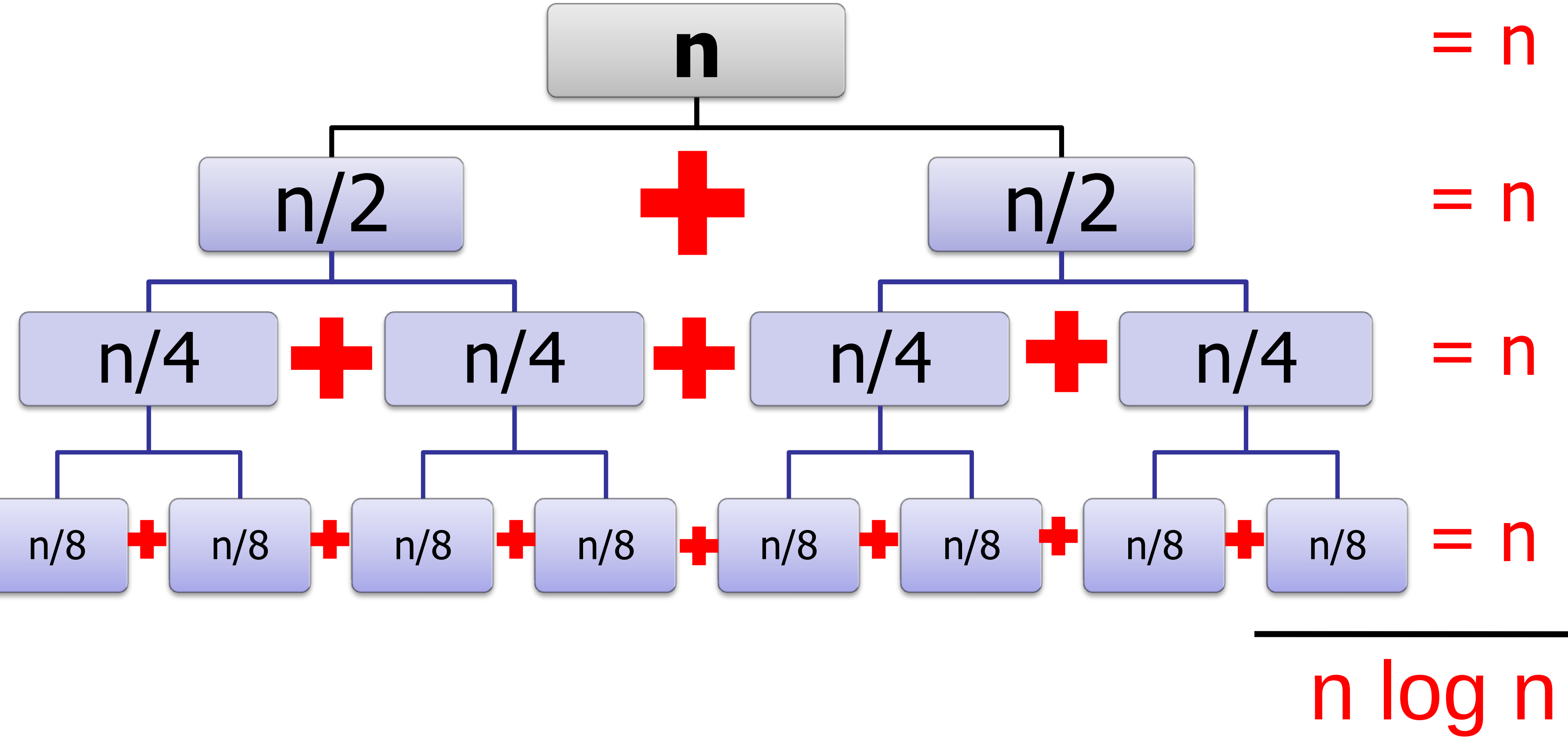
$S(n) = ?$

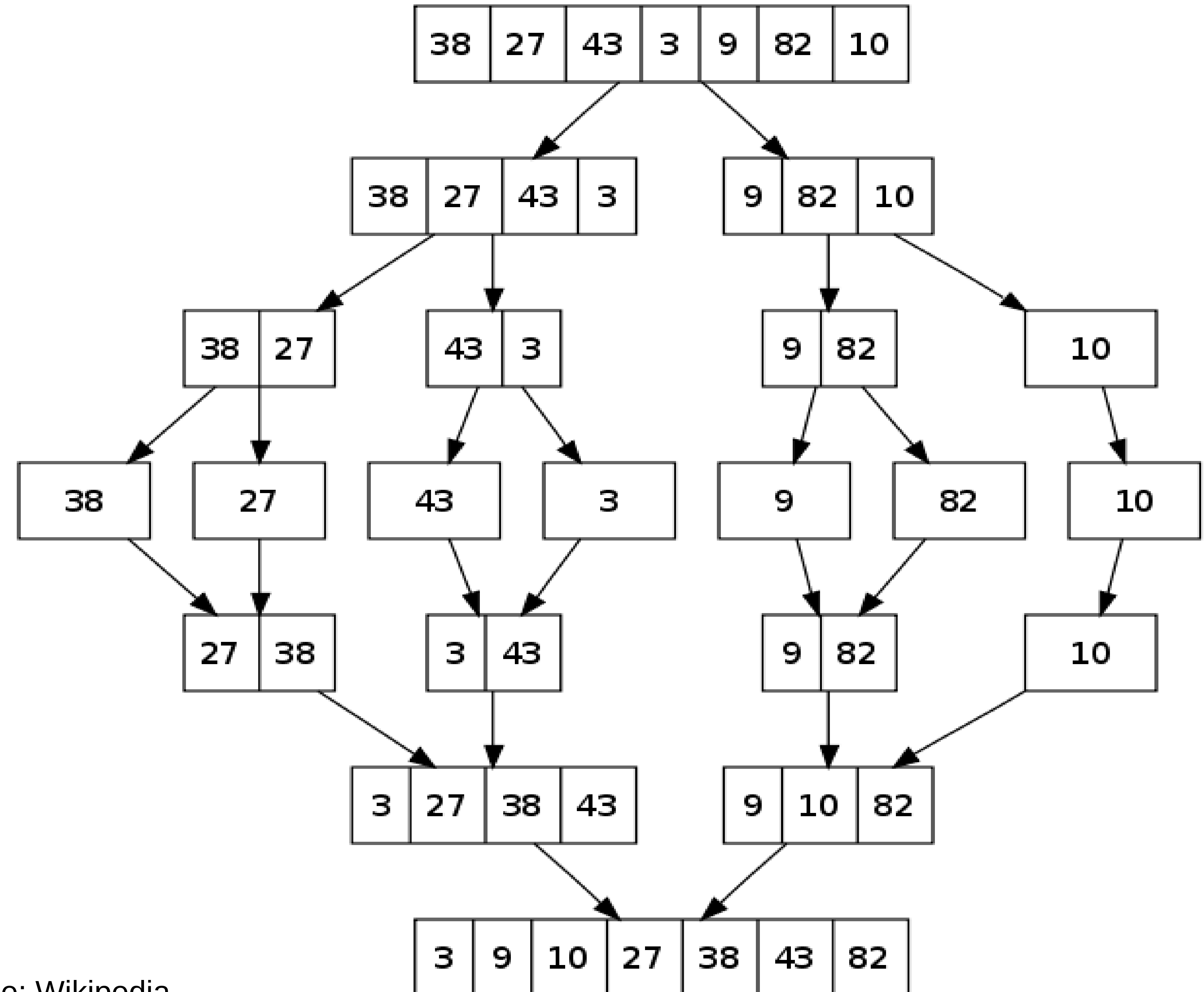
A. O(1)
B. O(n)
C. O(n log n)
D. O(n2)
E. O(n2 log n)
F. O(2n)



MergeSort Analysis

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



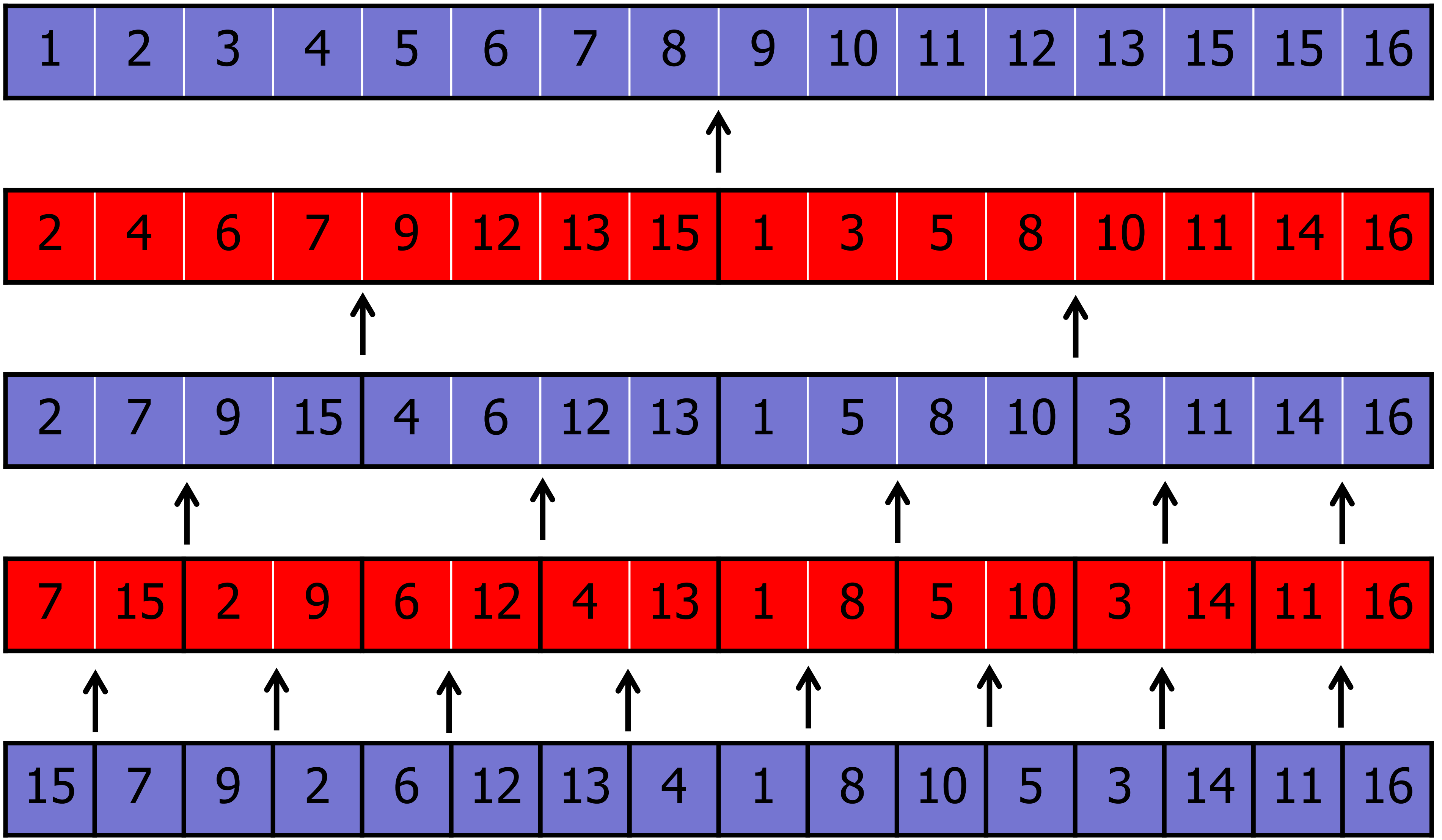


Source: Wikipedia

MergeSort, Bottom Up

15	7	9	2	6	12	13	4	1	8	10	5	3	14	11	16

MergeSort, Bottom Up



MergeSort Challenge

Implement MergeSort where:

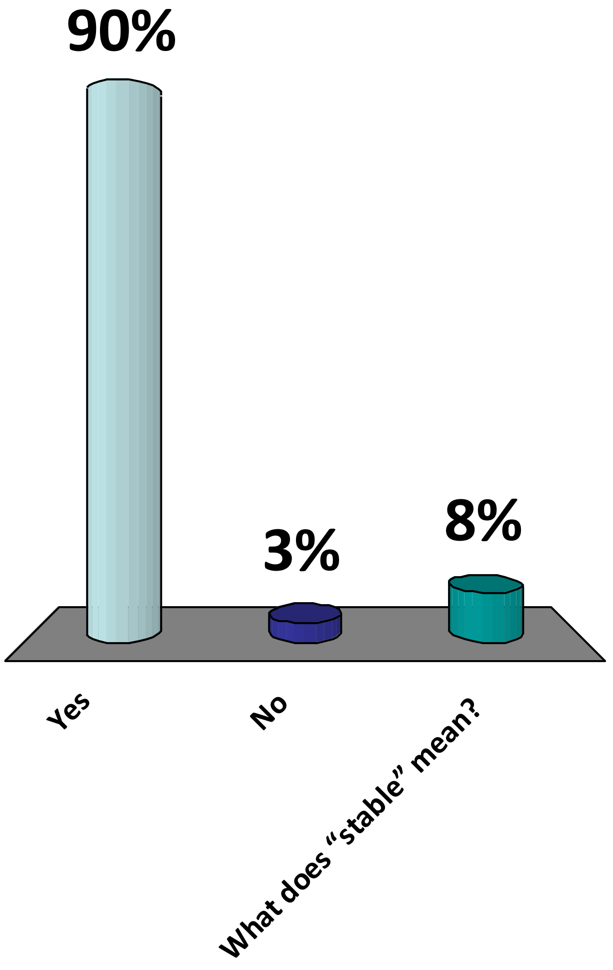
- It uses only 2n + O(log n) space.

  MergeSort (int[] inArray, int[] outArray)

- No new arrays are allocated during the sort.

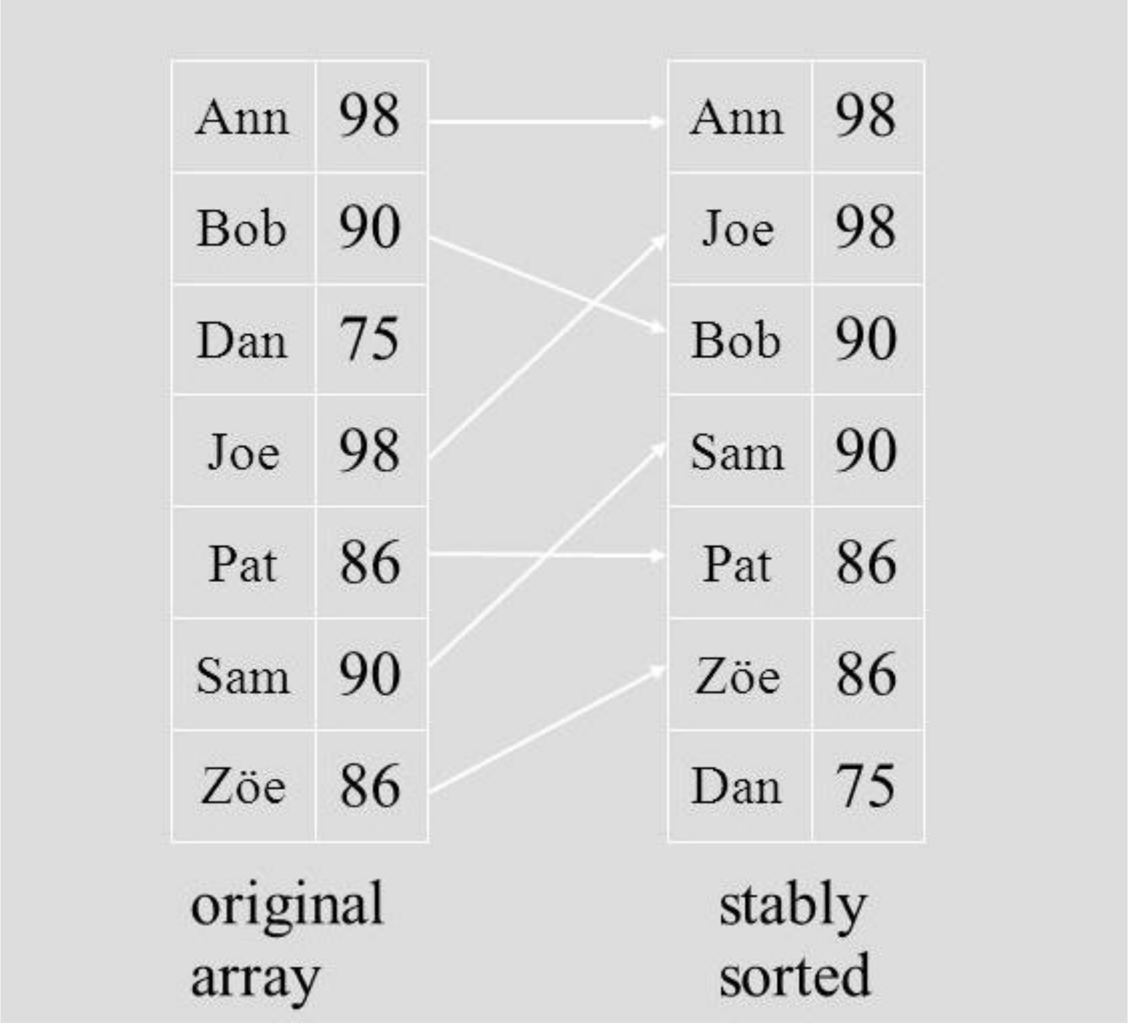
Is MergeSort stable?

A. Yes
B. No
C. What does “stable” mean?



Stable Sorting

E.g. sorting students' scores



Sorting Analysis

Summary:

- BubbleSort: $O(n^2)$
- SelectionSort: $O(n^2)$
- InsertionSort: $O(n^2)$
- MergeSort: $O(n \log n)$

Properties: time, space, stability

Summary

Name	Best Case	Average Case	Worst Case	Memory	Stable?
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes
Merge Sort	n log n	n log n	n log n	N	Yes

Today: Sorting, Part II

QuickSort

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

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
  - Turing Award: 1980
- Visiting student at Moscow State University
- Used for machine translation (English/Russian)

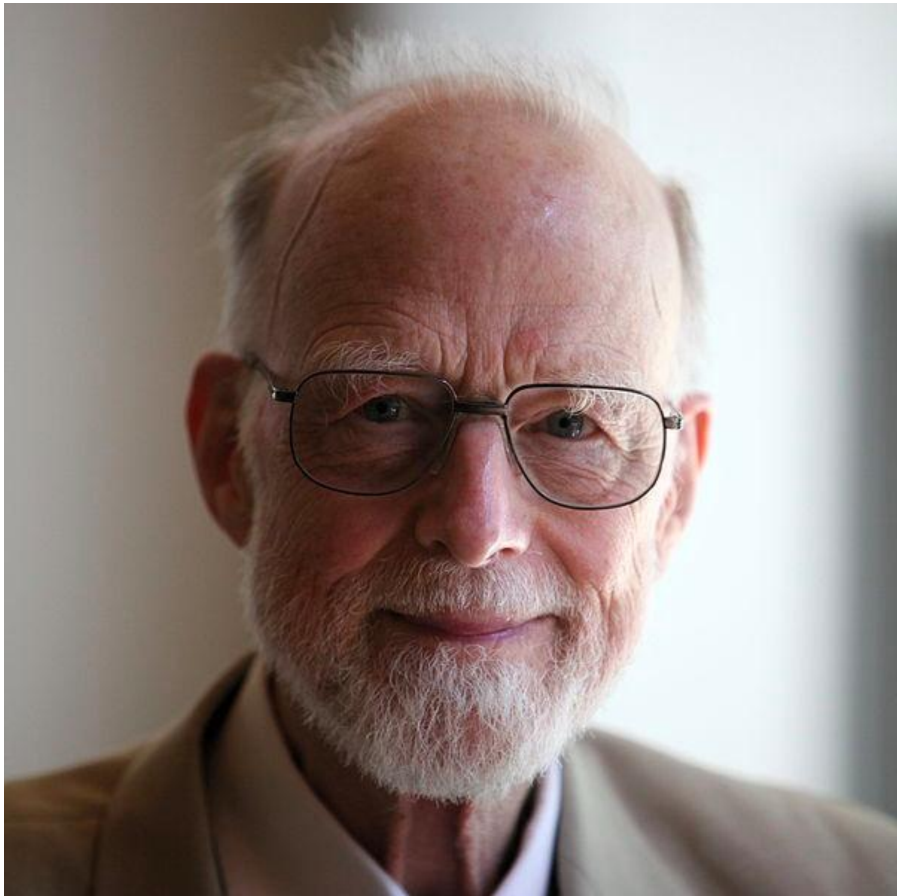


Photo: Wikimedia Commons (Rama)

QuickSort

History:

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

In practice:

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

QuickSort Today

1960: Invented by Hoare

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

1993: Bentley & McIlroy improvements

“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

2009: Vladimir Yaroslavskiy

- Dual-pivot Quicksort !!!
- Now standard in Java 7
- 10% faster!

2012: Sebastian Wild and Markus E. Nebel

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

QuickSort

In class:

- Easy to understand! (divide-and-conquer…)

- Moderately hard to implement correctly.

- Harder to analyze. (Randomization…)

- Challenging to optimize.

QuickSort

For starter, let’s assume the world is beautiful….

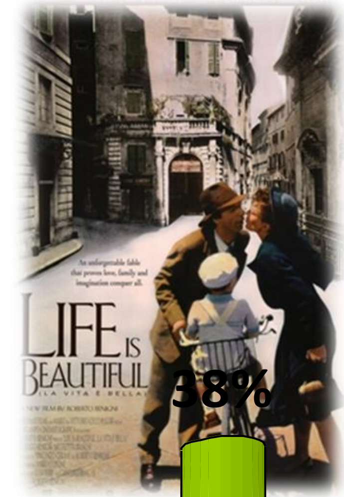
Quick Sort Assumption. Let’s assume…

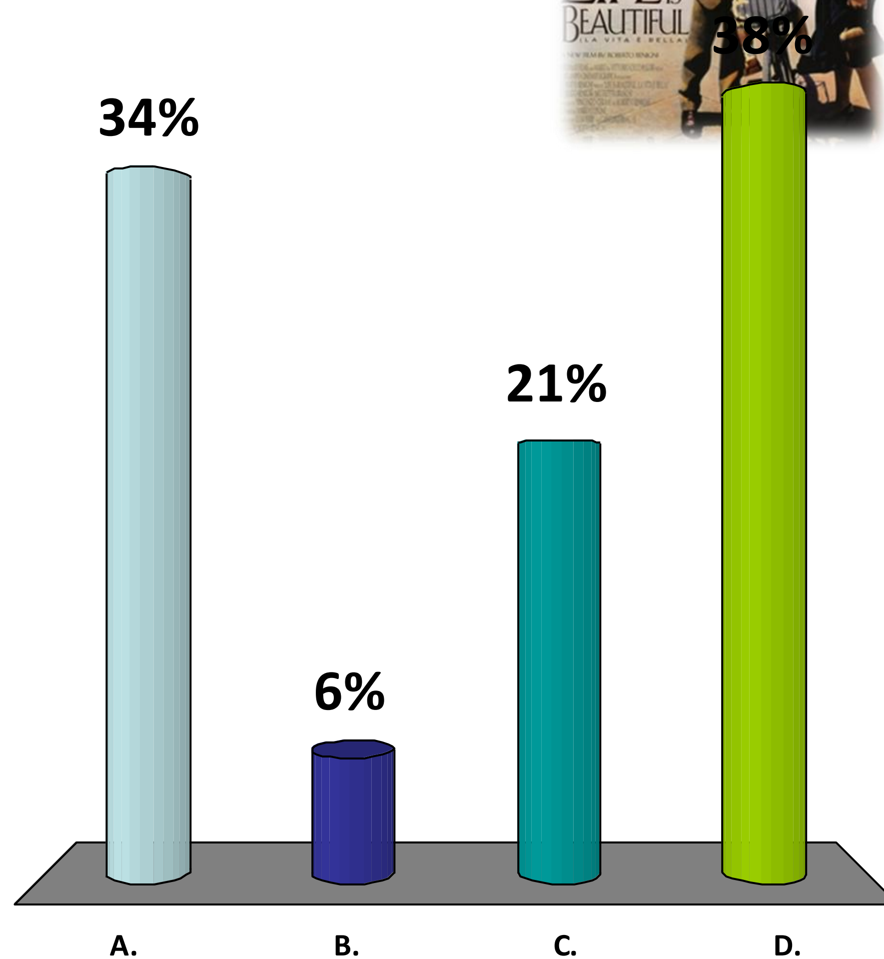
A. Elements are sorted

B. Elements are randomized

C. Elements have no duplicates

D. Let’s watch the movie “Life is beautiful”

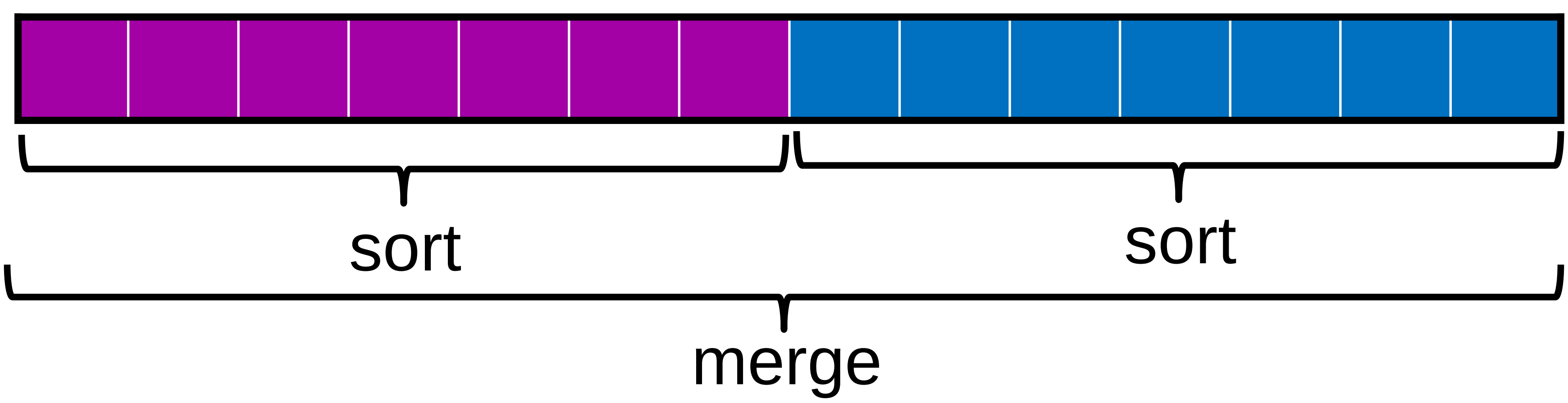




Let's assume each element is unique in the array

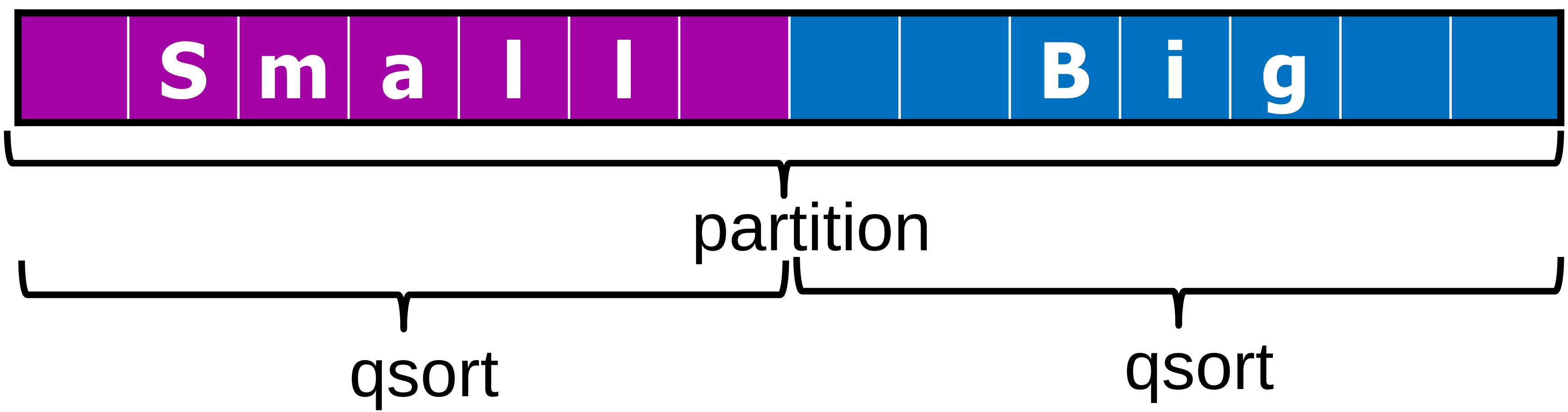
Recall: MergeSort

```
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)
```



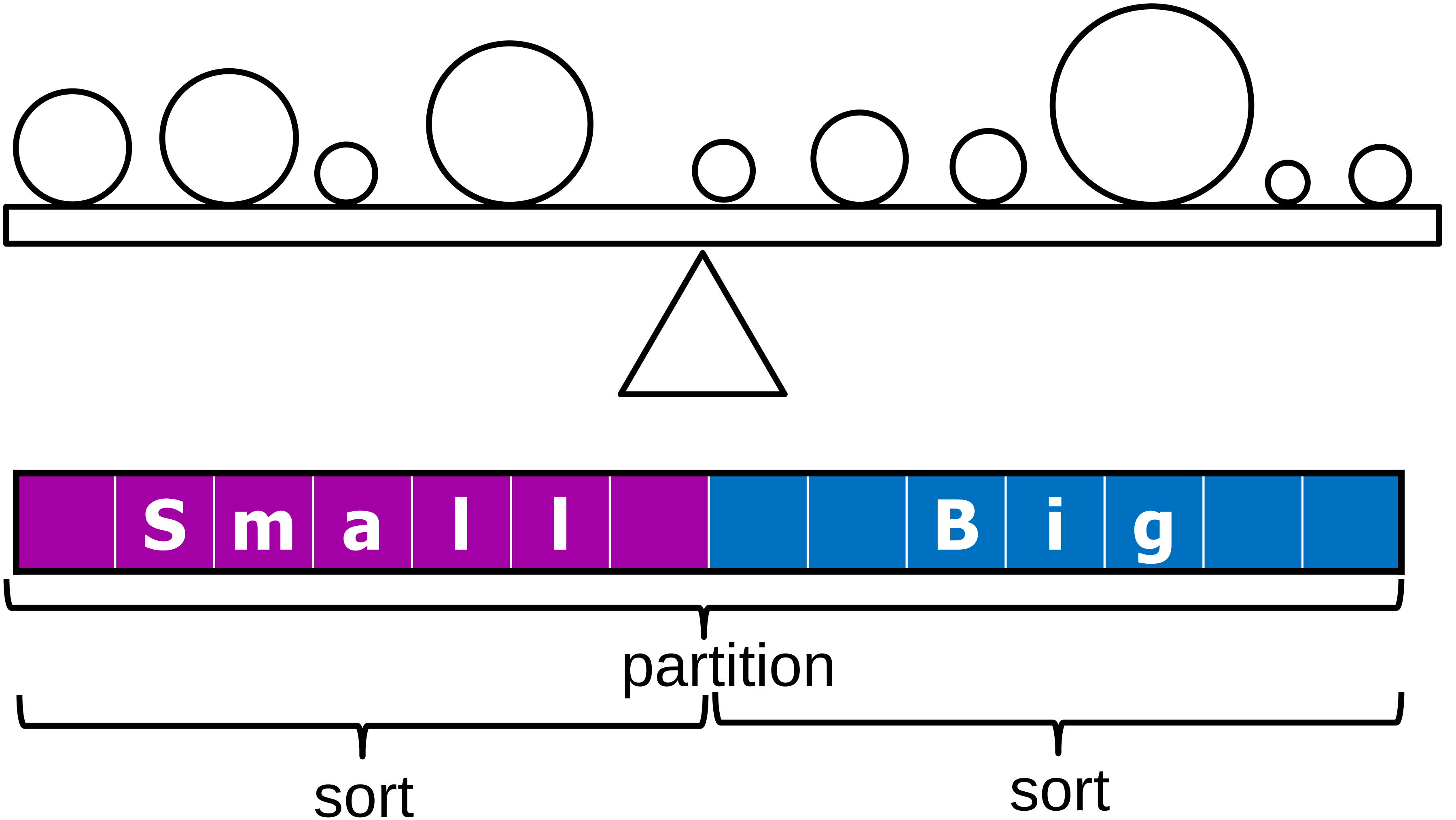
QuickSort

```
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)
```



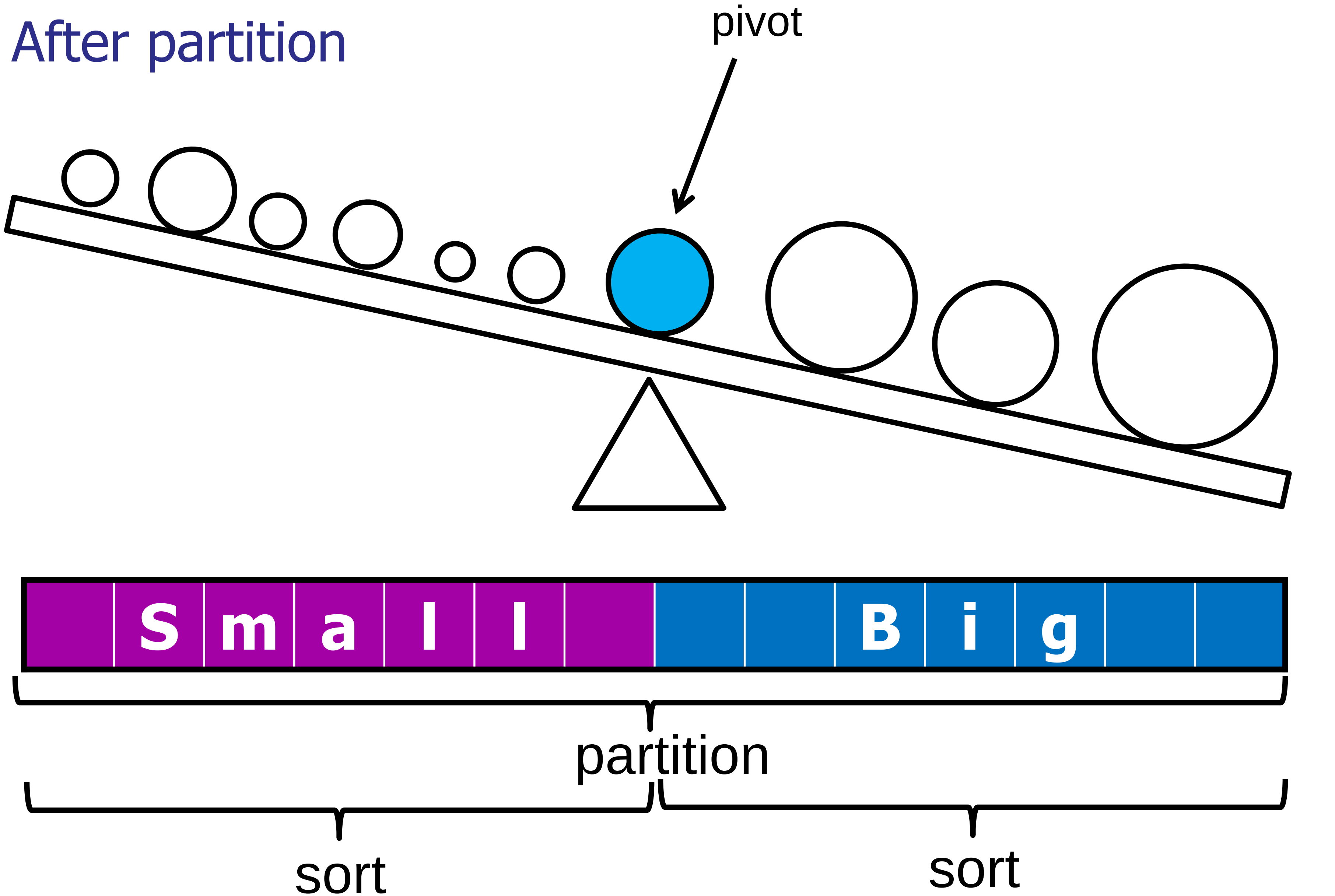
QuickSort

Before partition



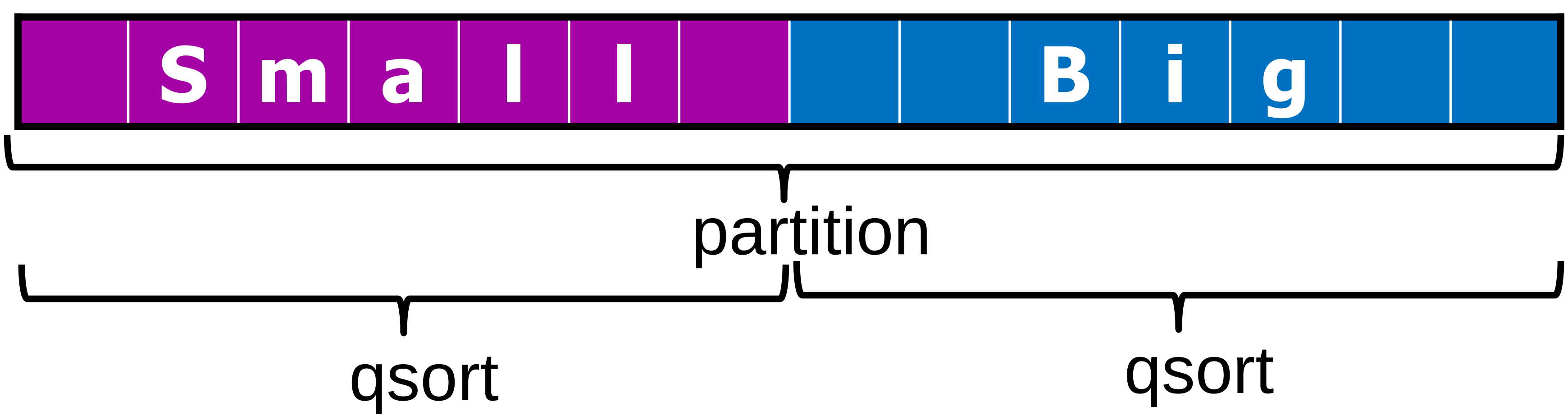
QuickSort

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	$> x$

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, e.g. the first element.*
2. Find all elements smaller than the pivot.
3. Find all elements larger than the pivot.

Quicksort

Example:

6 3 9 8 4 2

Choose
this as
the pivot

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			

Choose
this as
the pivot

Quicksort

Example:

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

Choose
this as
the pivot

Quicksort

Example:

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

Quicksort

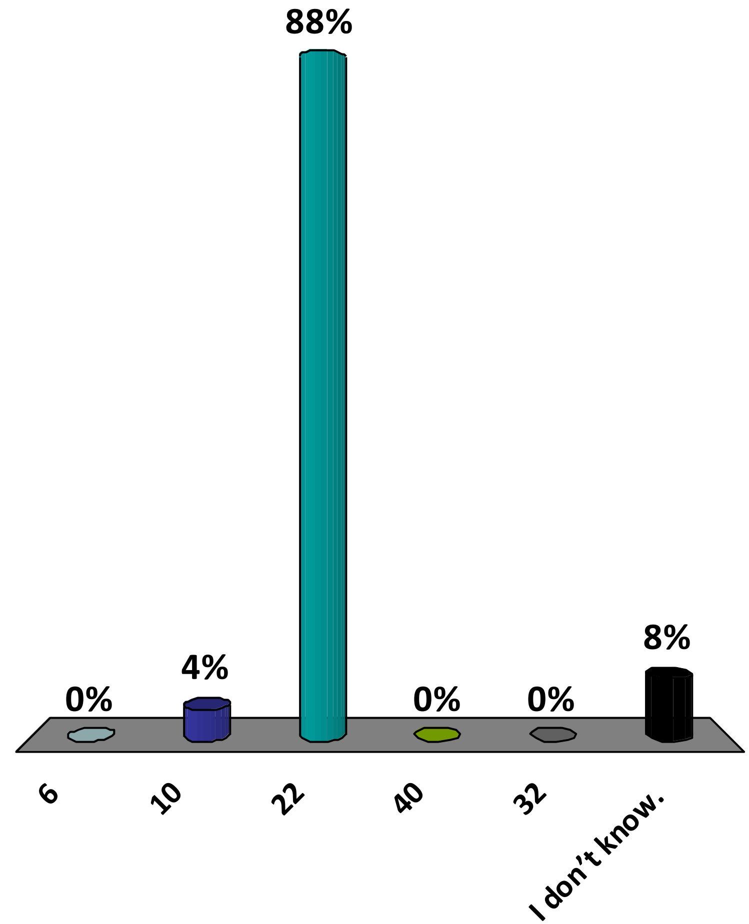
Example:

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

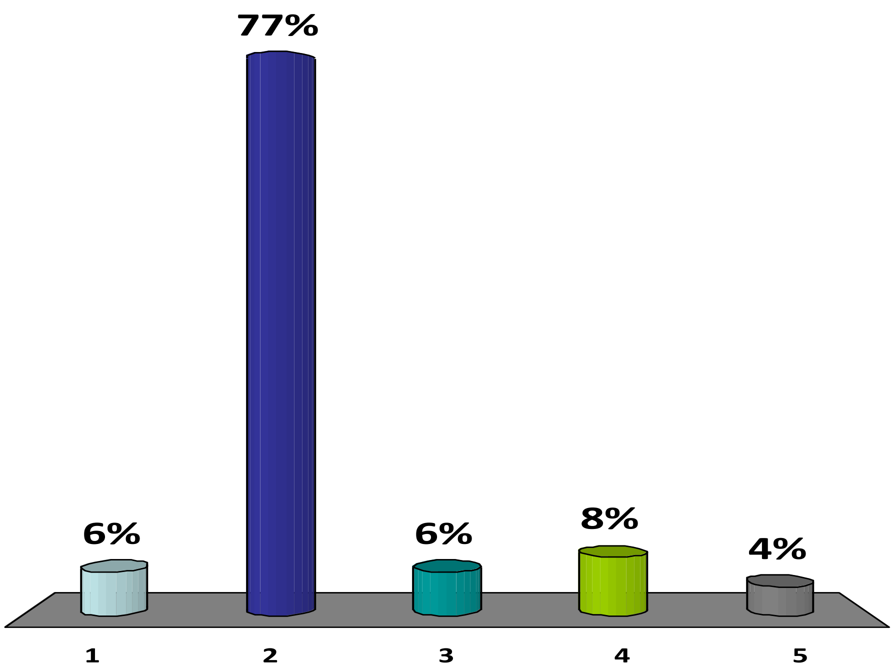
Example:

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.

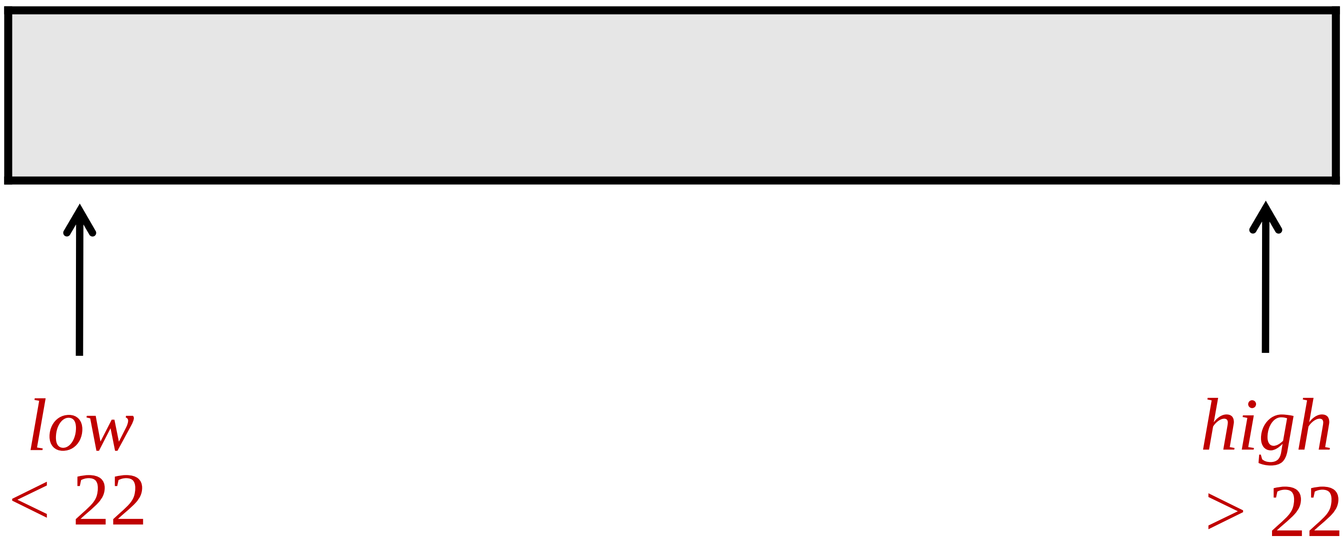


Partitioning an Array

Example: partition around 22

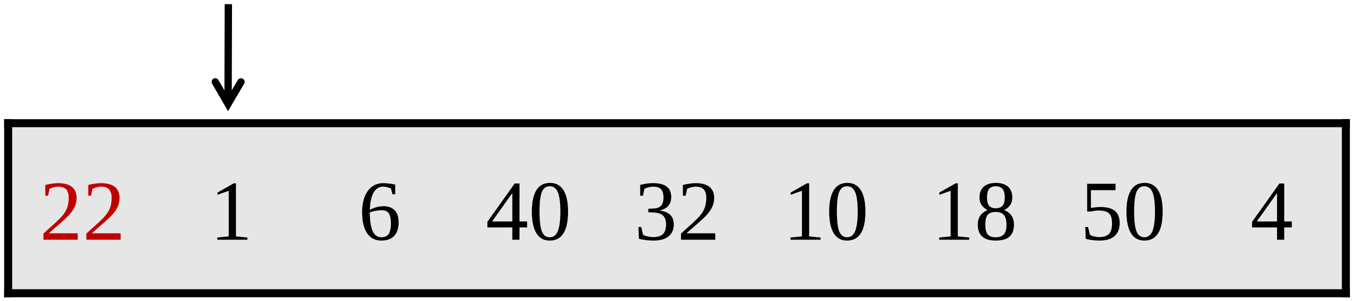
22	1	6	40	32	10	18	50	4

Output array:

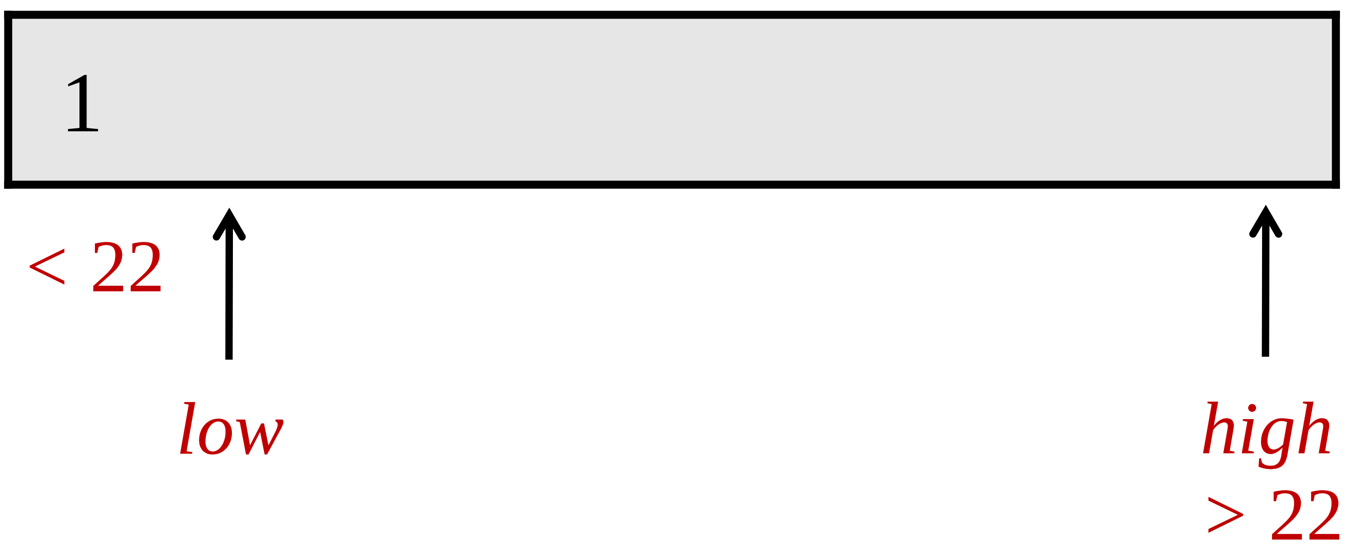


Partitioning an Array

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

< 22

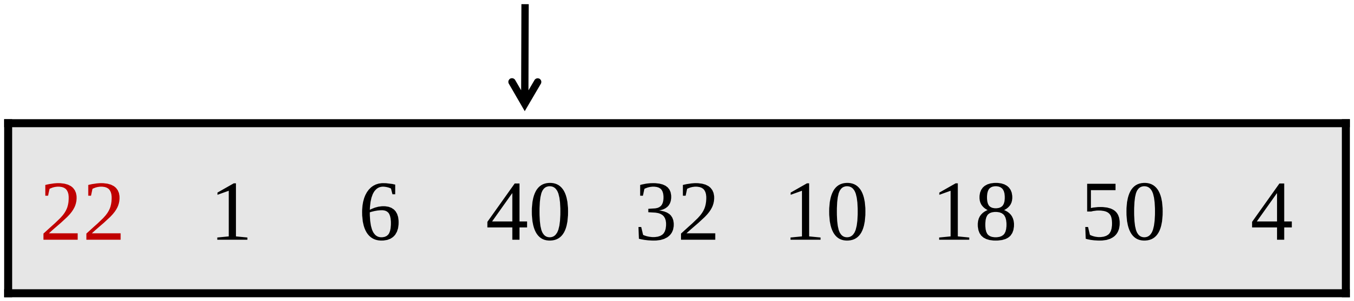
low

high

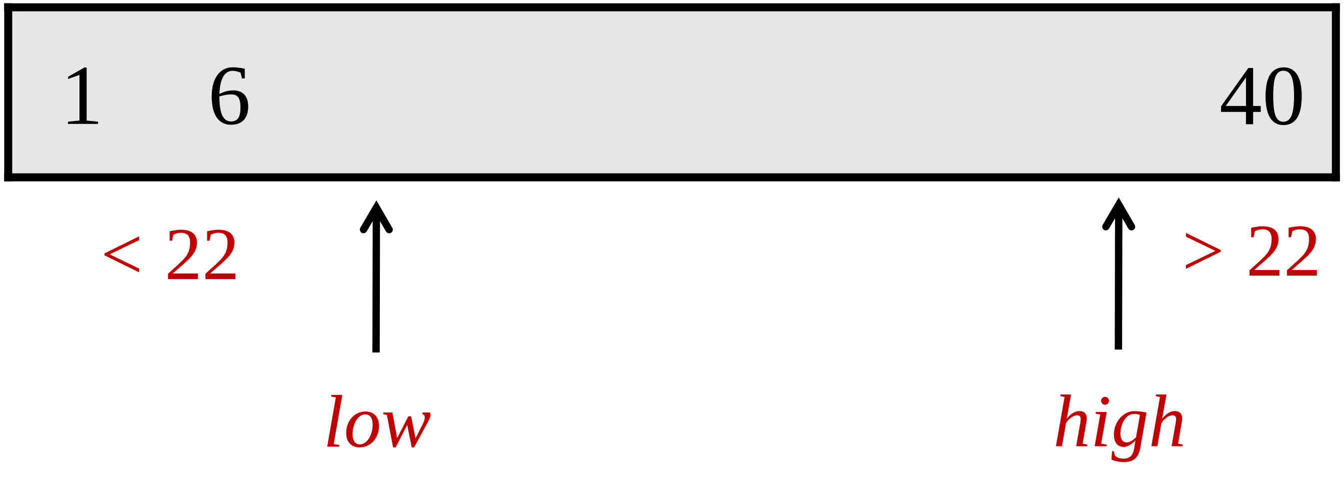
> 22

Partitioning an Array

Example: partition around 22

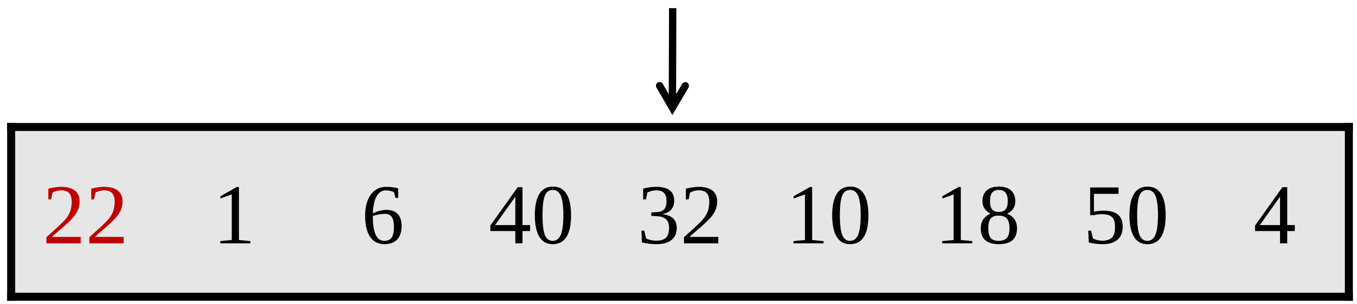


Output array:

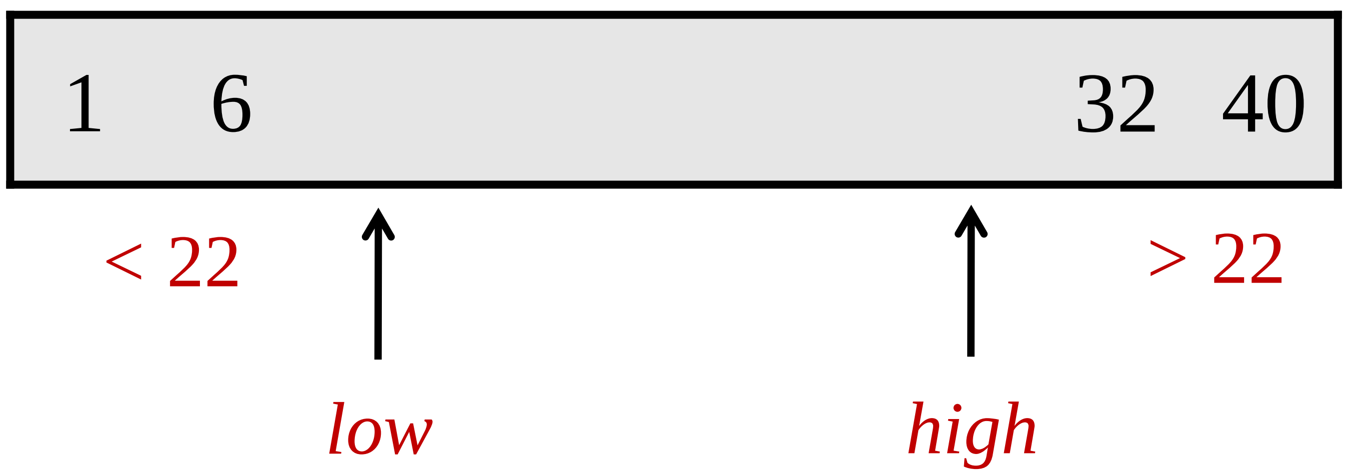


Partitioning an Array

Example: partition around 22

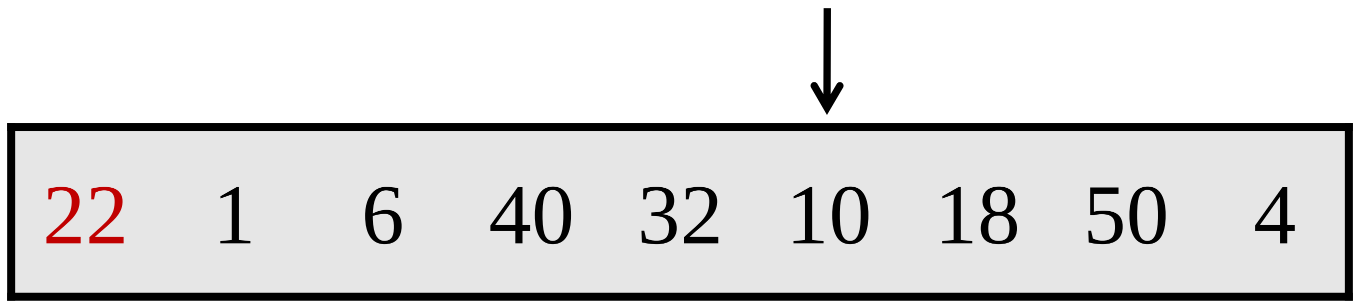


Output array:

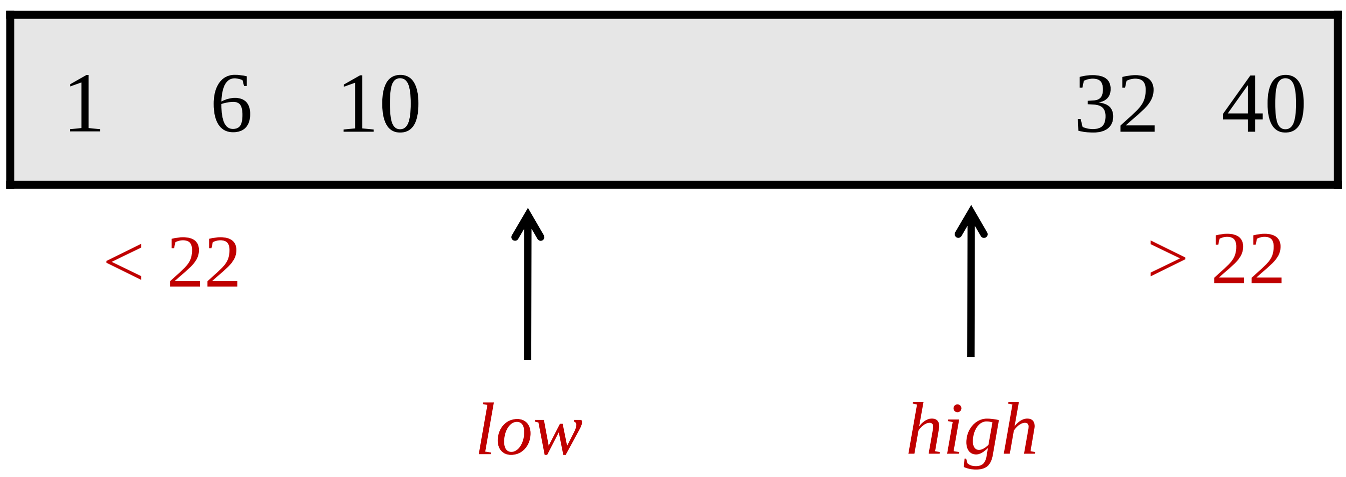


Partitioning an Array

Example: partition around 22

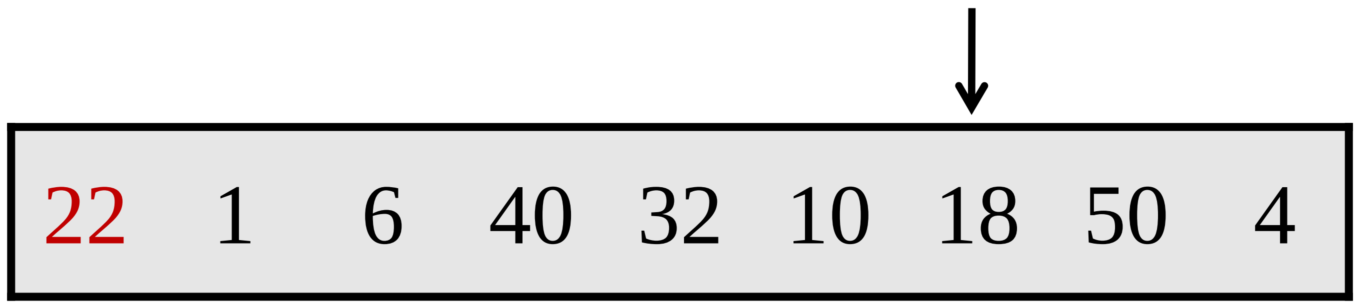


Output array:

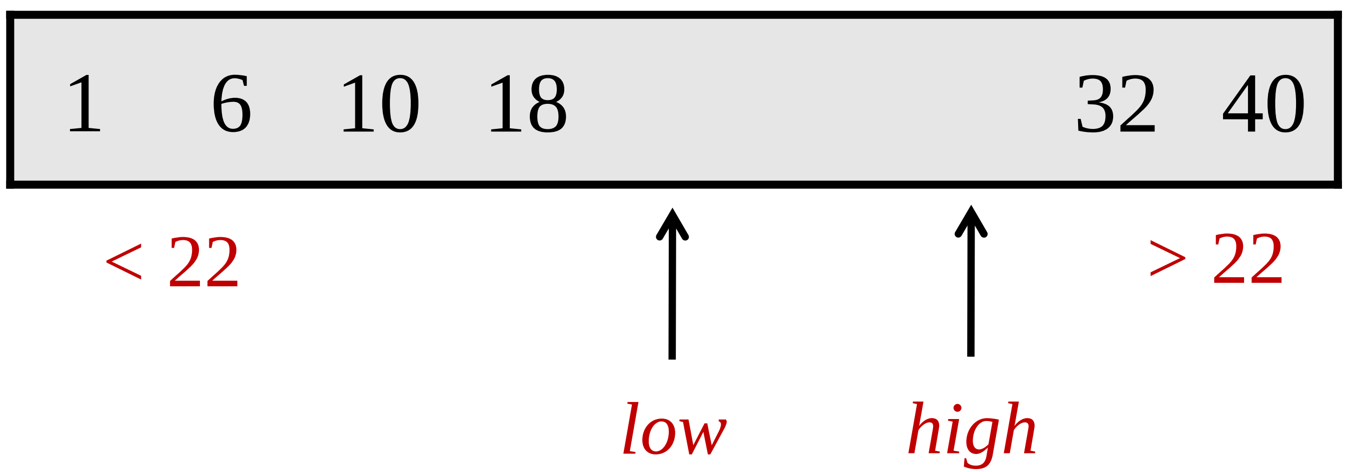


Partitioning an Array

Example: partition around 22

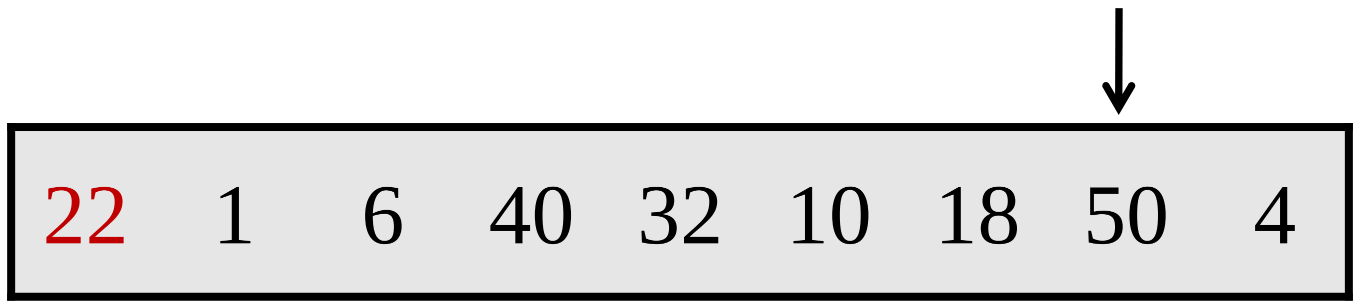


Output array:

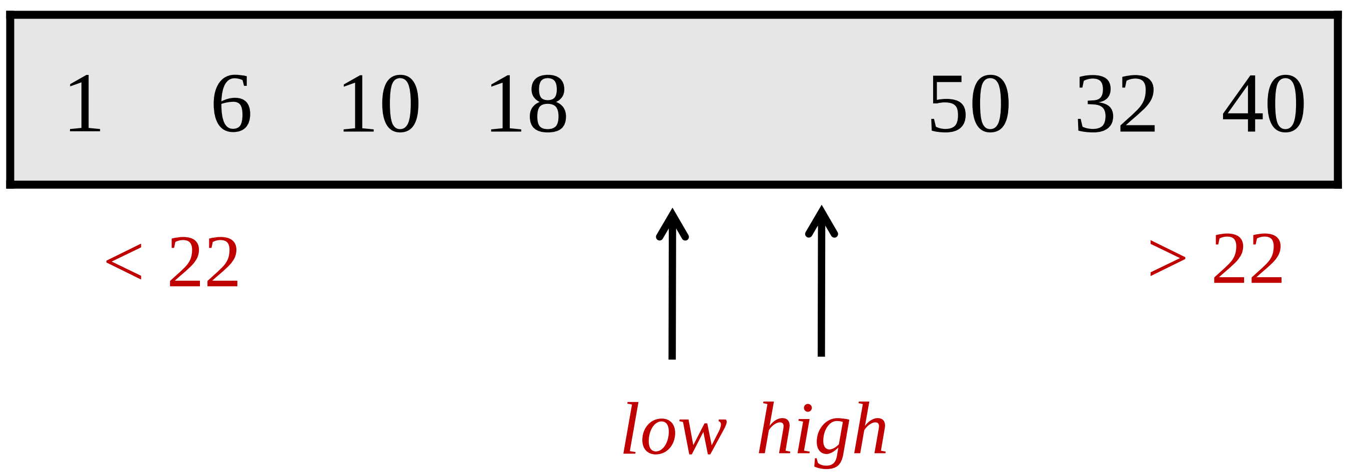


Partitioning an Array

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	4		50	32	40

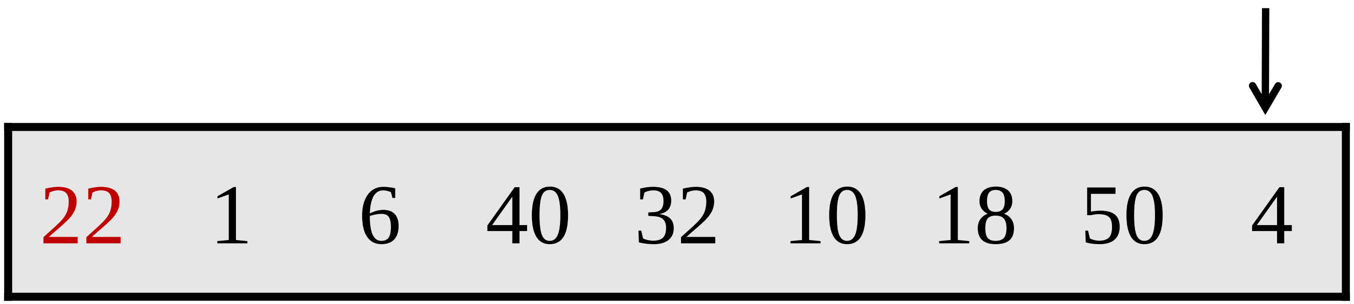
< 22

> 22

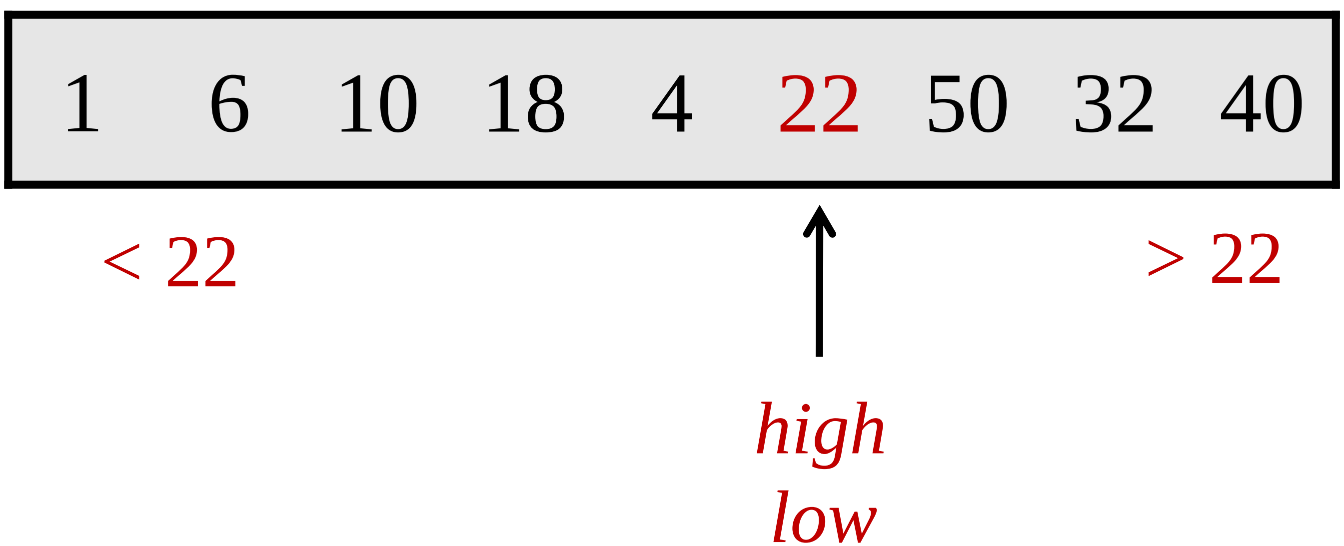
high
low

Partitioning an Array

Example: partition around 22



Output array:

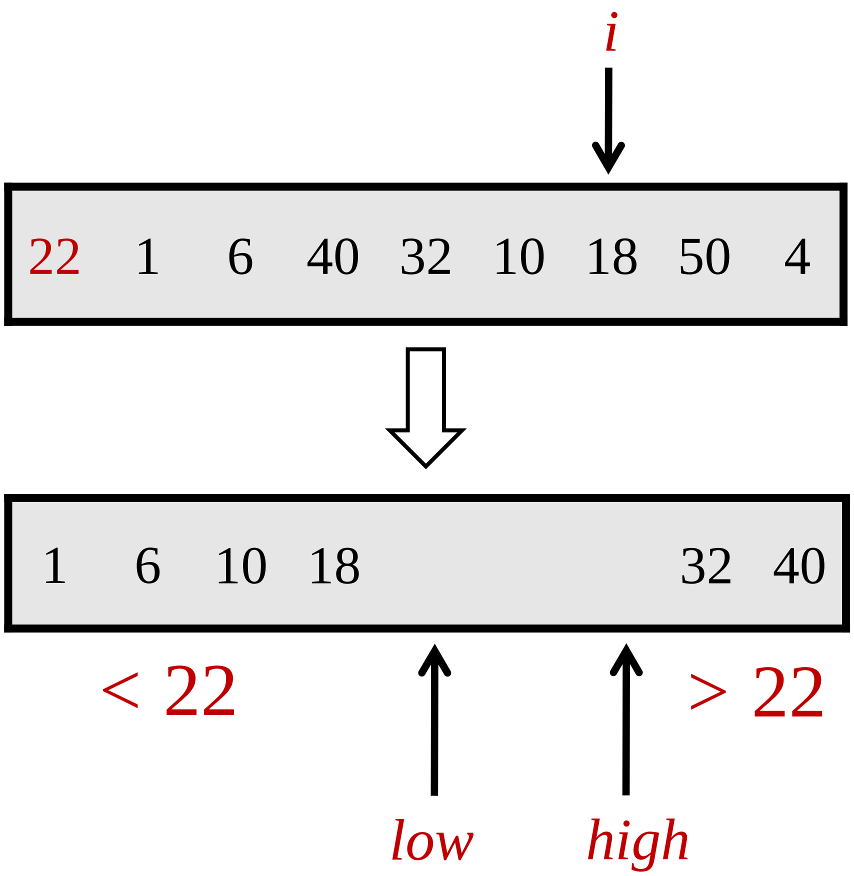


Partition

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: $B[i] = pivot$

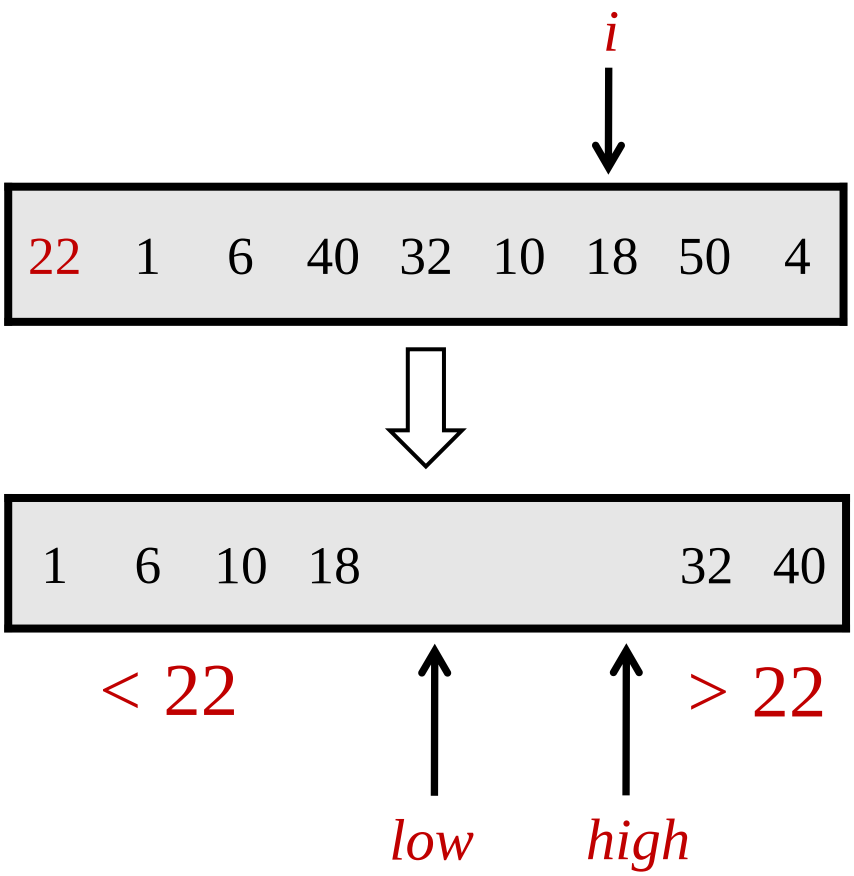
By invariants, B is partitioned around the pivot.

Partition

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



What is wrong with the partition procedure?

24% 1. There is a bug. It doesn't work.

38% ✓ 2. It uses too much memory.

18% 3. It is too slow.

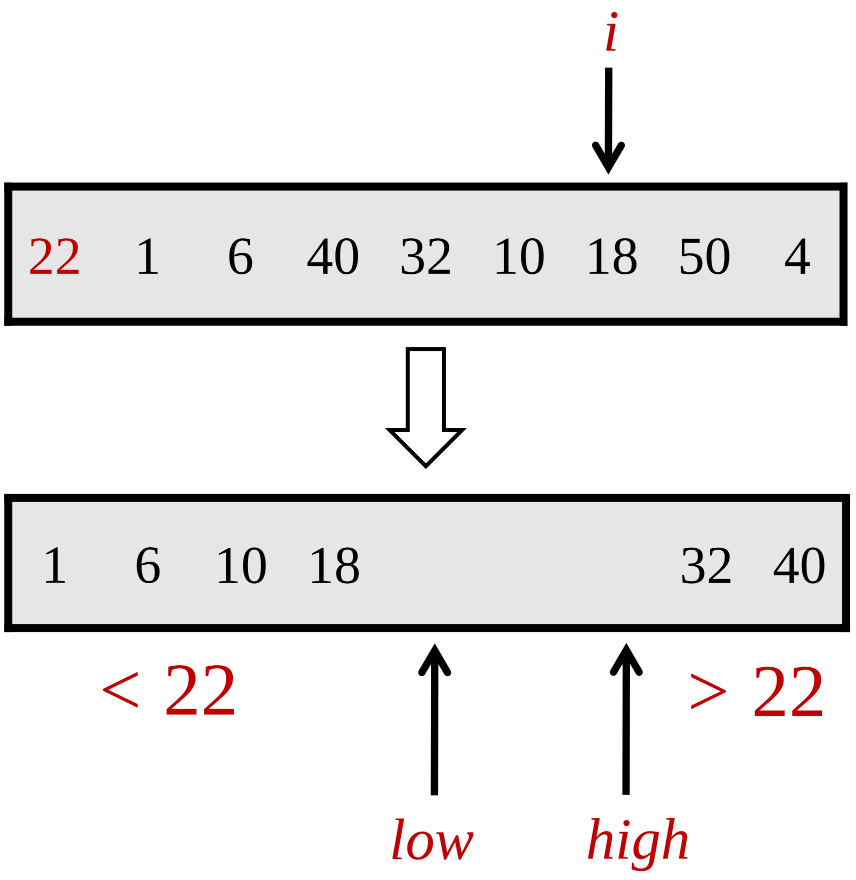
4% 4. It only works for integers.

7% 5. It has poor caching performance.

[redacted] works perfectly.

Partition

```
partition(A[2..n], n, pivot)                    // Assume no duplicates
    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 >
```



Partitioning an Array "in-place"

Example: partition around 22

22	1	6	40	32	10	18	4	50

low < 22

high > 22

Partitioning an Array "in-place"

Example: partition around 22

22	1	6	40	32	10	18	4	50

low
< 22

high
> 22

Move until it's bigger than the pivot

Move until it's less than the pivot

Partitioning an Array

Example: partition around 22

22	1	6	40	32	10	18	4	50

< 22 

low

high

> 22

Partitioning an Array

Example: partition around 22

22	1	6	40	32	10	18	4	50

< 22

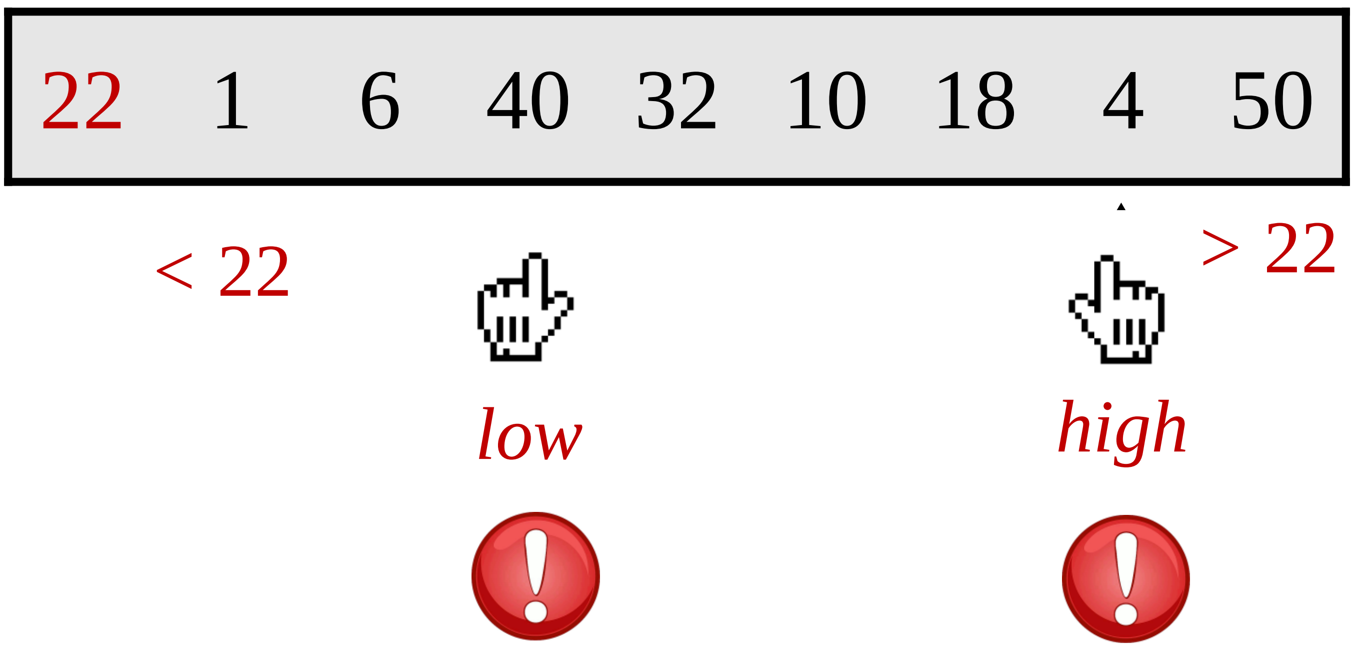
low

high

> 22

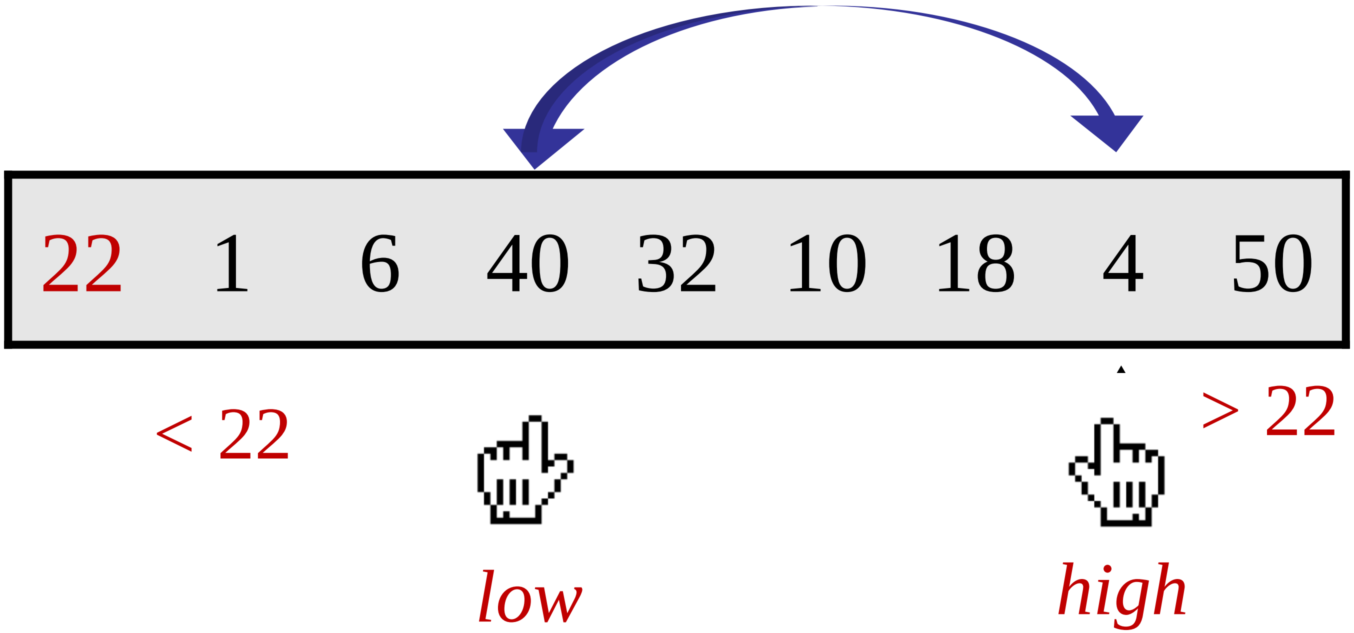
Partitioning an Array

Example: partition around 22



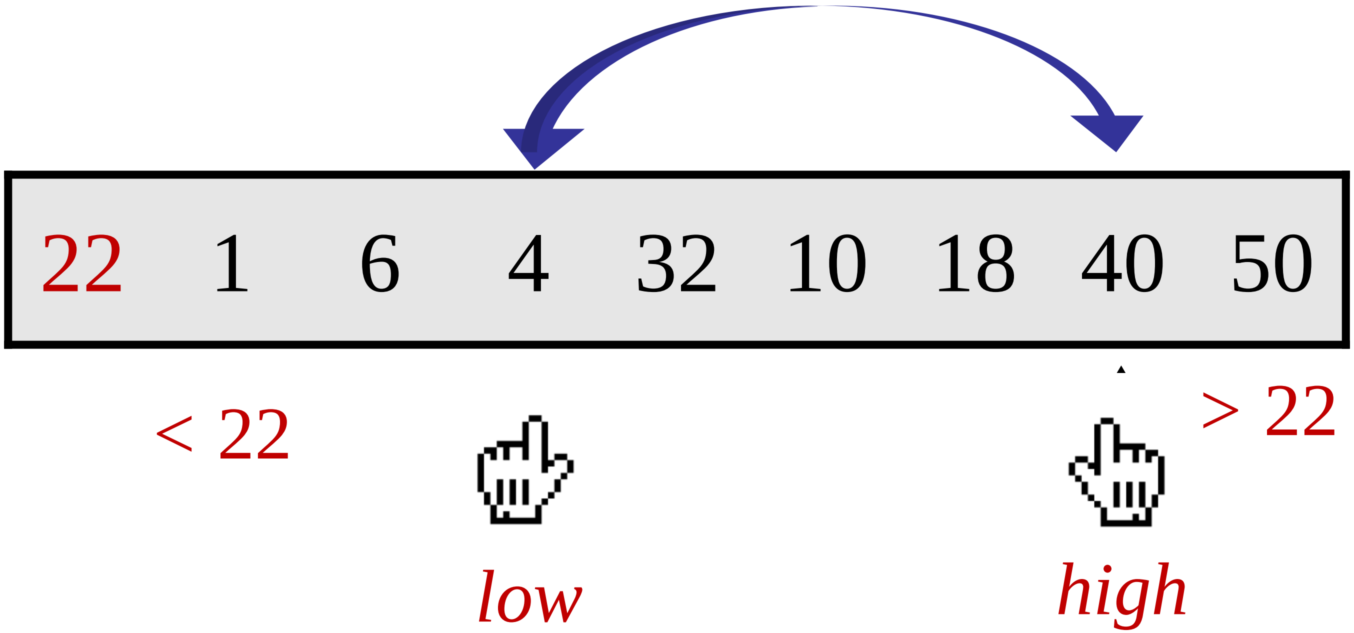
Partitioning an Array

Example: partition around 22



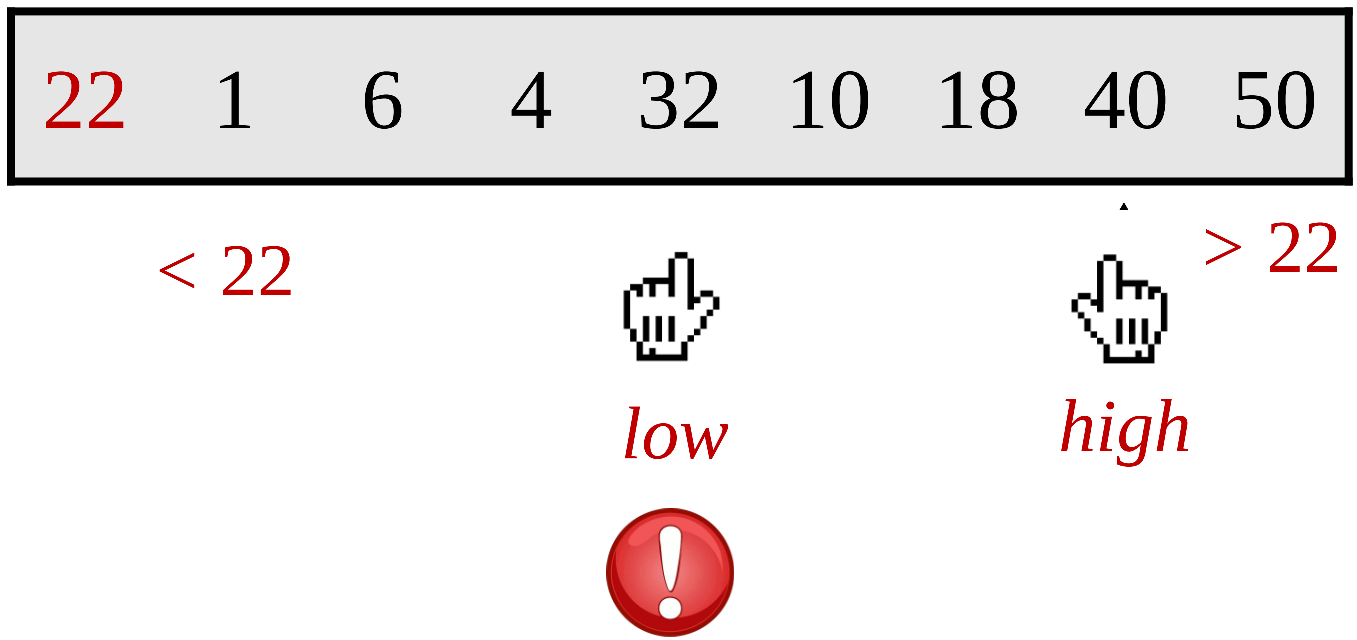
Partitioning an Array

Example: partition around 22



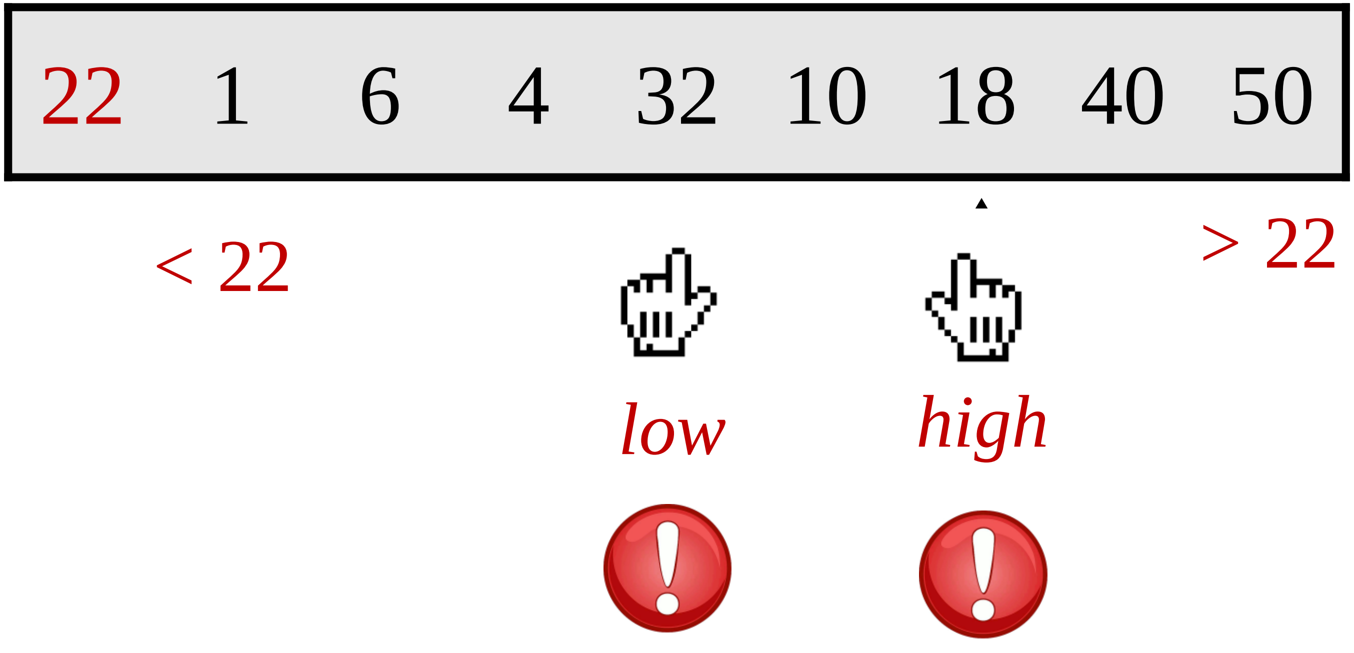
Partitioning an Array

Example: partition around 22



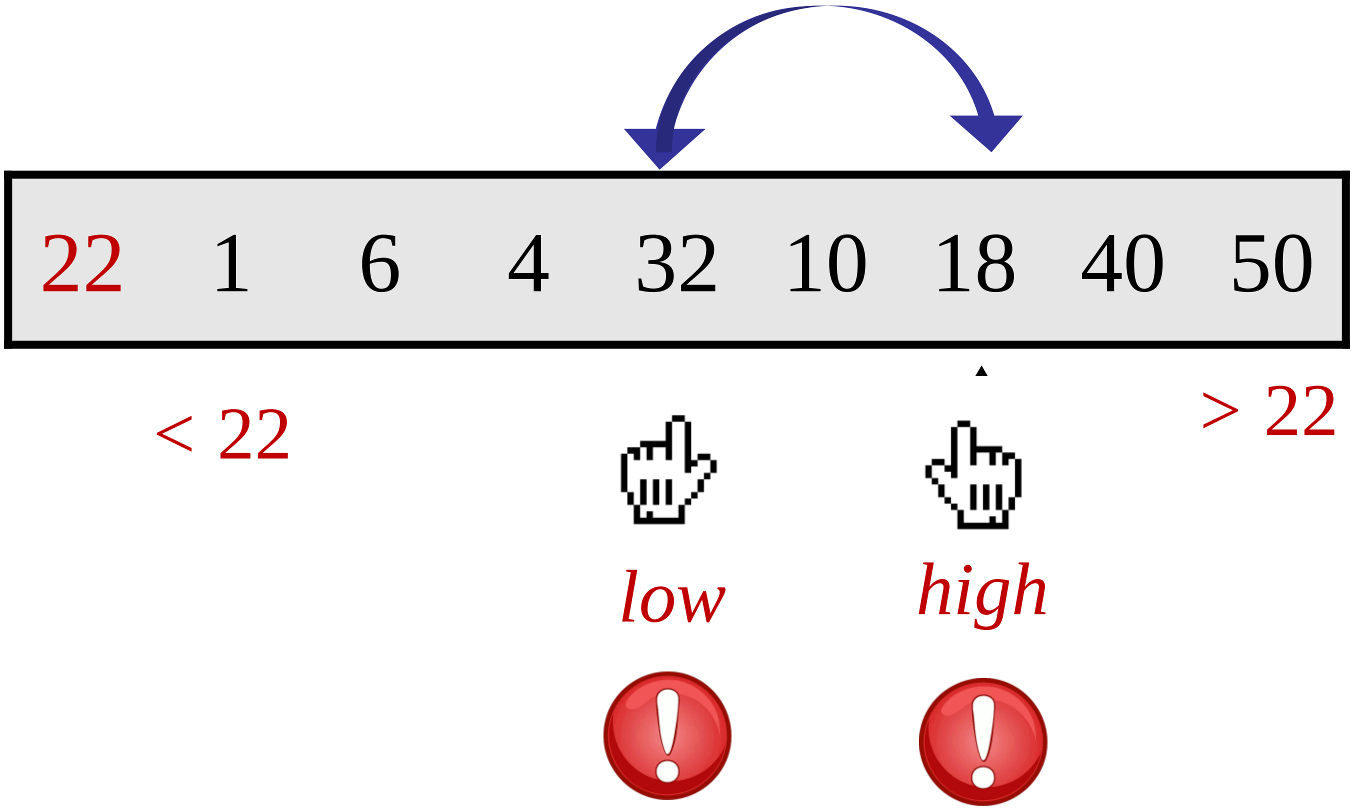
Partitioning an Array

Example: partition around 22



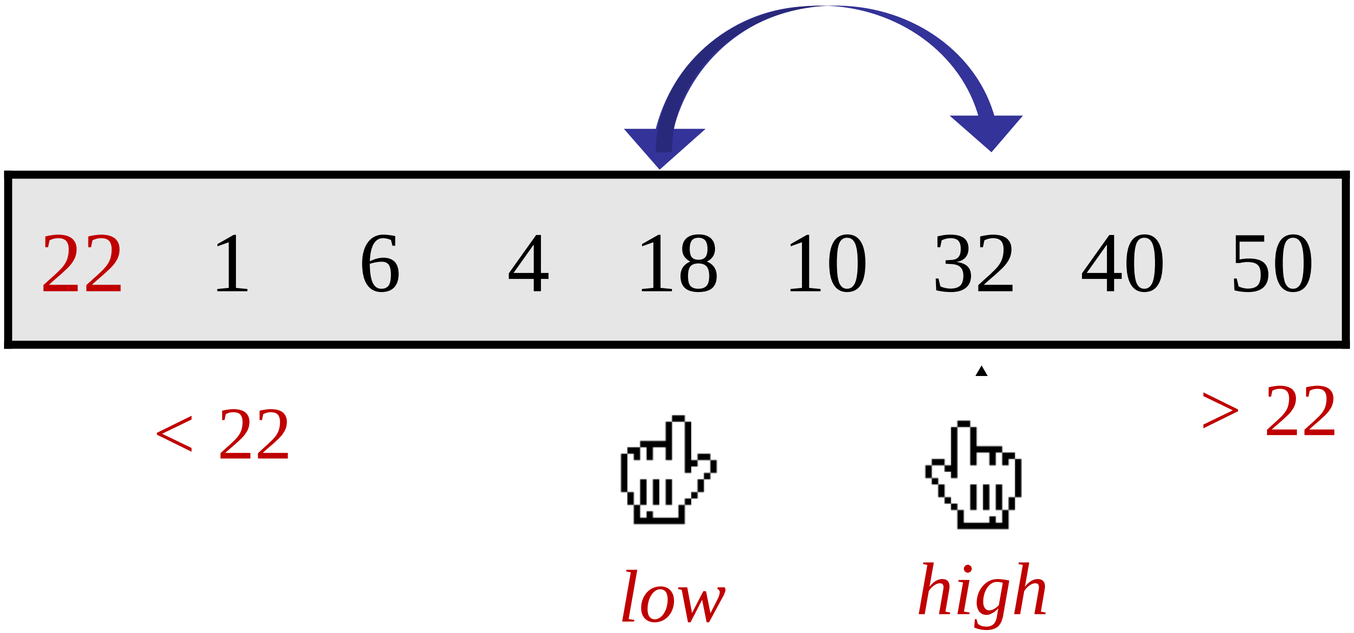
Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22

22	1	6	4	18	10	32	40	50

< 22

> 22

low *high*

Partitioning an Array

Example: partition around 22

22	1	6	4	18	10	32	40	50

< 22

> 22

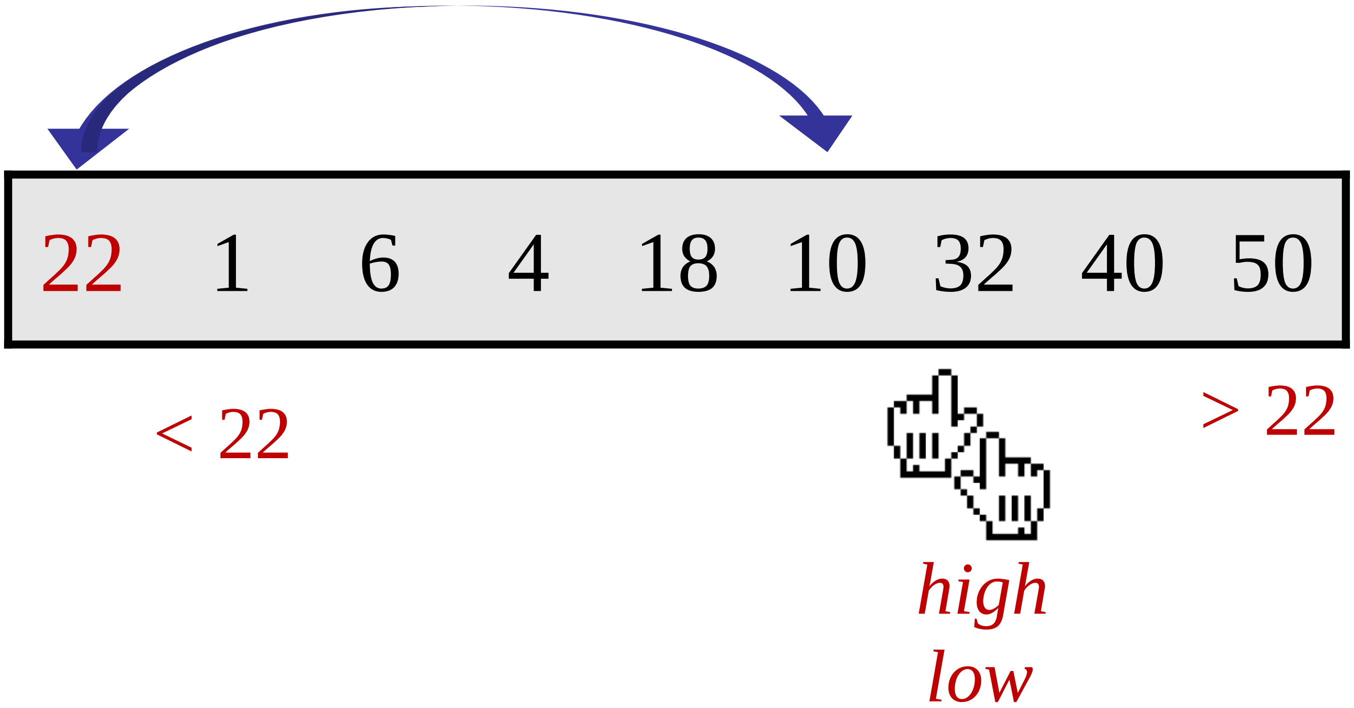
high

low

`high == low`

Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22

10	1	6	4	18	22	32	40	50

< 22

> 22

high

low

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;
```

We actually have freedom to choose a different pivot other than the first element. But let's assume pIndex = 1 for the moment

Pseudocode

vs.

Real Code

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 $A[low]>\text{pivot}$.

Partition

$partition(A[1..n], n, pIndex)$ **// Assume no duplicates, $n>1$**

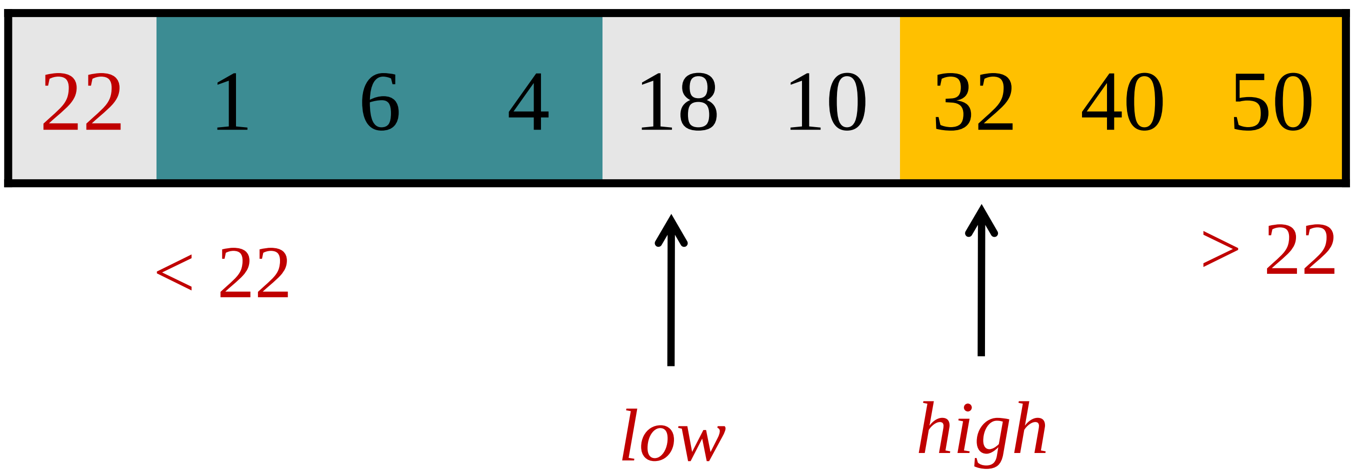
```
partition(A[1..n], n, pIndex)             // Assume no duplicates, n>1
    pivot = A[pIndex];                    // pIndex is the index of the pivot
    swap(A[1], A[pIndex]);                // store pivot in A[1]
    low = 2;                              // start after pivot in A[1]
    high = n+1;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

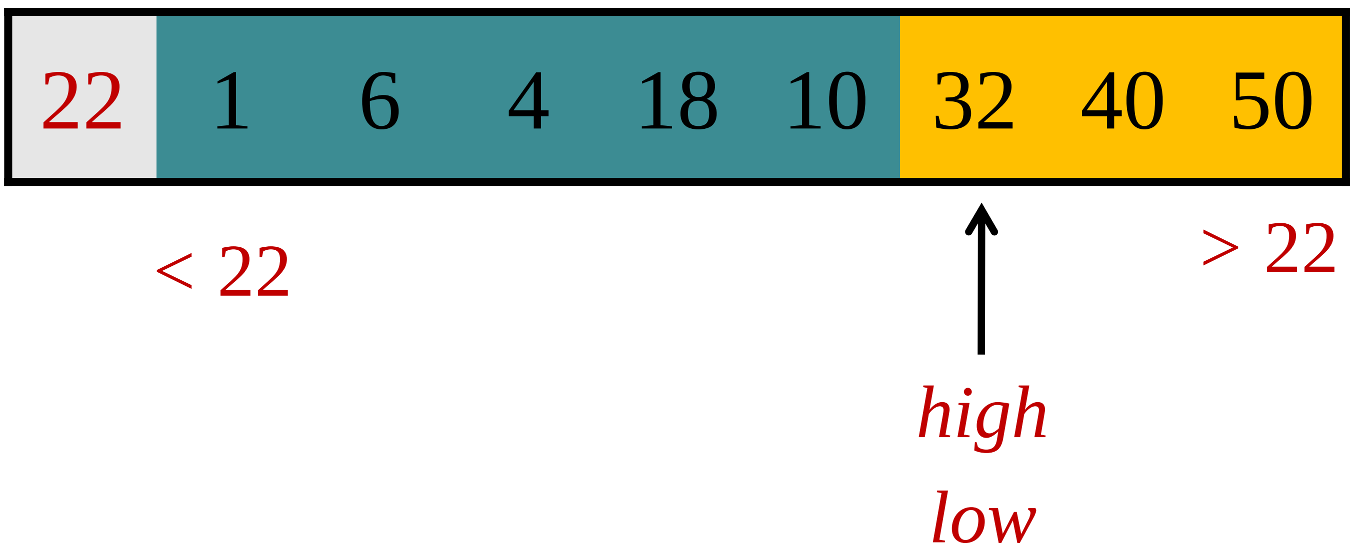


Partition

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

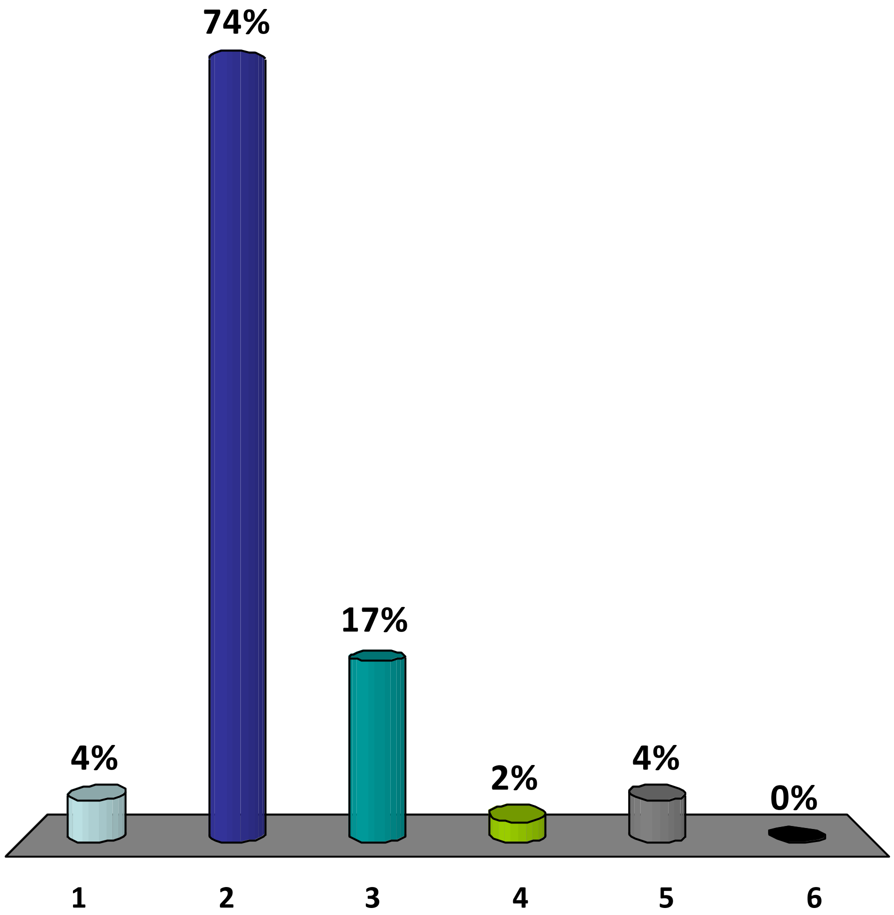
Claim: Array A is partitioned around the pivot

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

Today: Sorting, Part II

QuickSort

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

QuickSort

What happens if there are duplicates?

Partition

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

```
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;
```

low/high never change!

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;
```

Duplicates

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

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

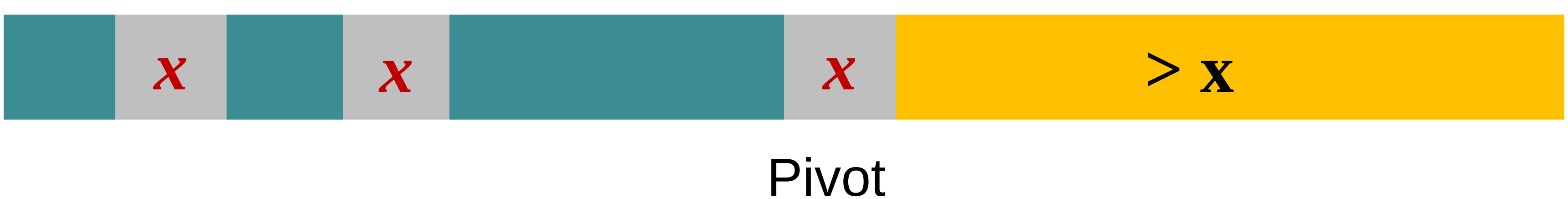
else

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;

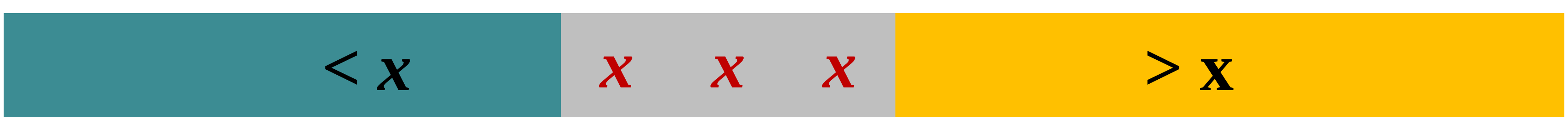
else

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)



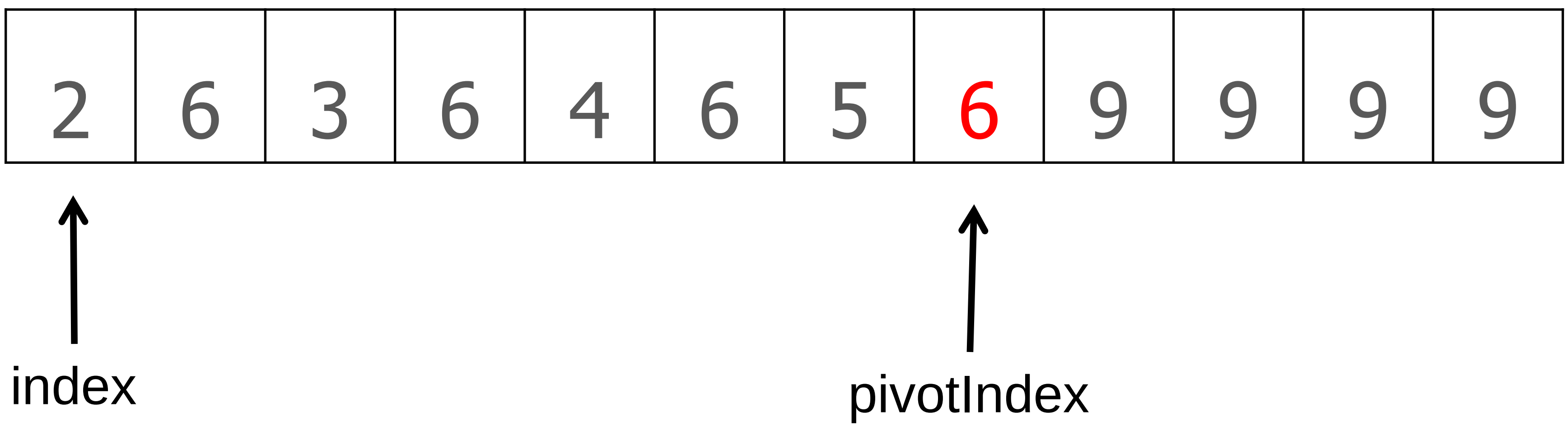
Pivot

Partition

```
packDuplicates(A[1..n], n, pivotIndex)
    pivot = A[pivotIndex];
    index = 1;
    while (index < pivotIndex)
        if (A[index] == pivot) {
            pivotIndex--;
            swap(A[index], A[pivotIndex]);
        }
        else {
            index ++;
        }
}
```

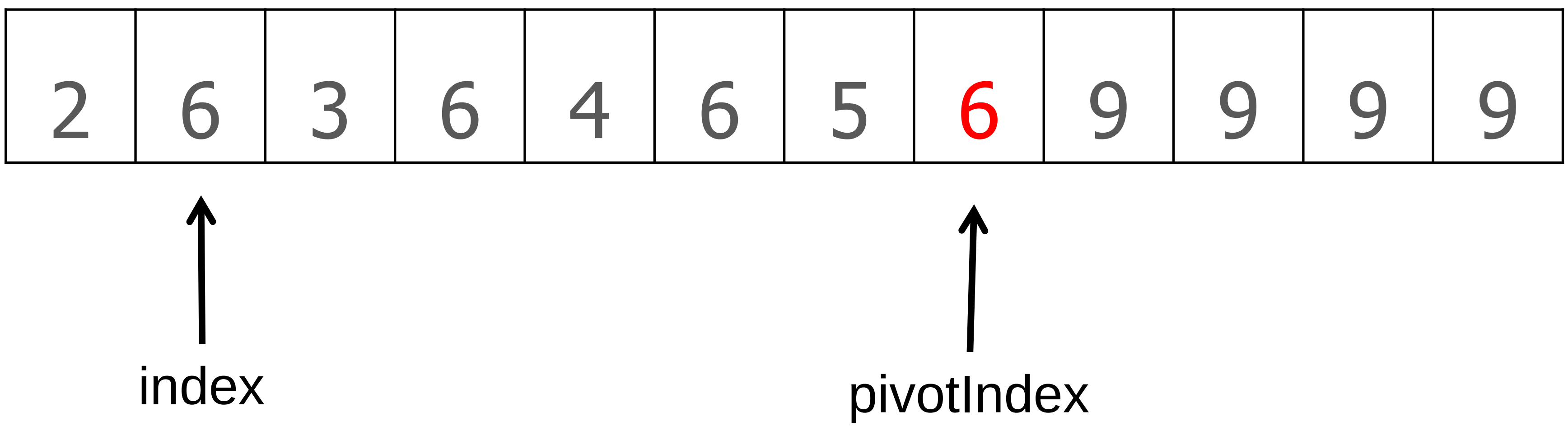
Duplicates

Example:



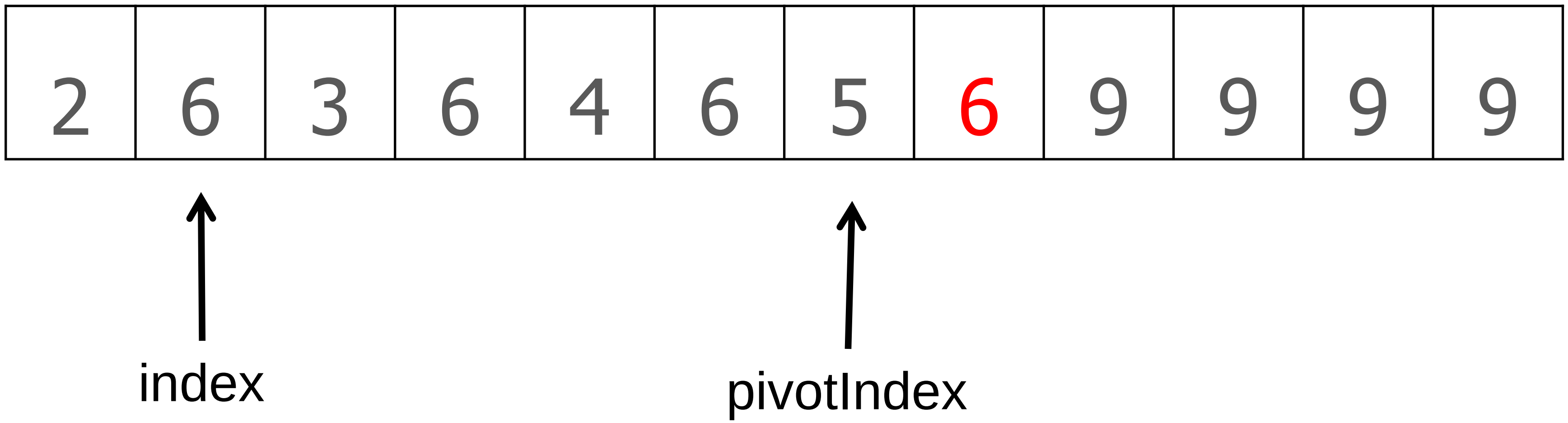
Duplicates

Example:



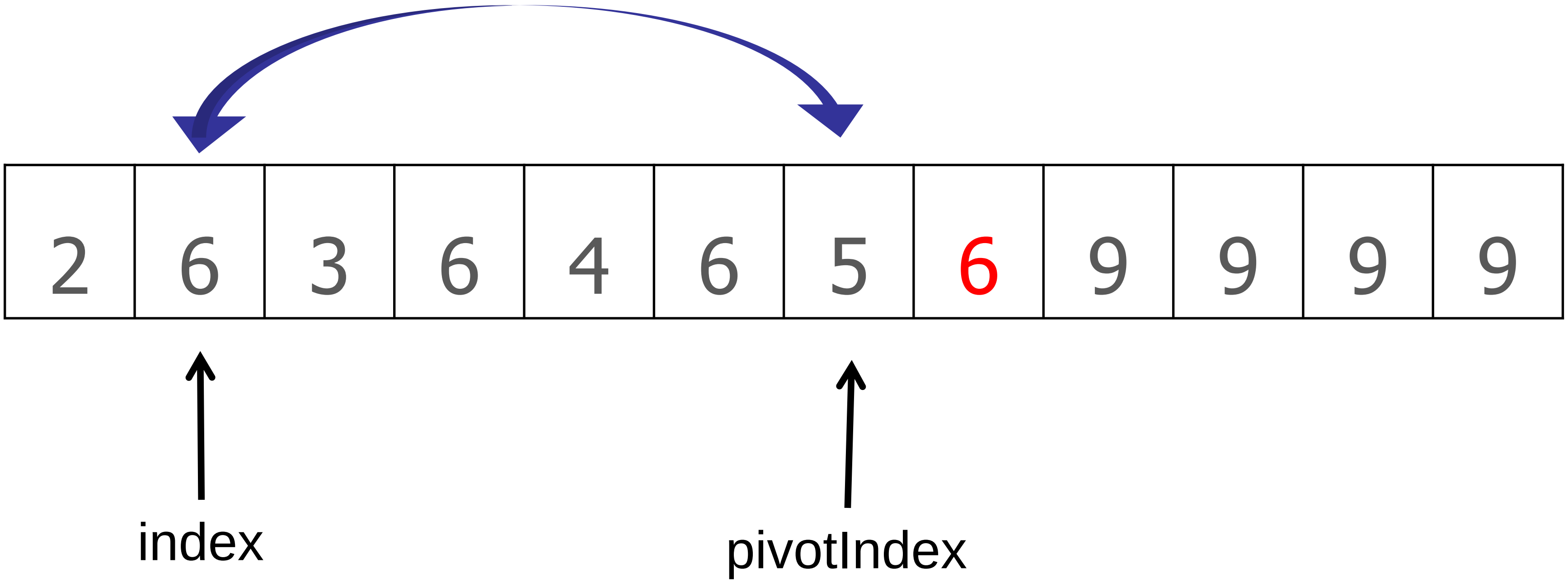
Duplicates

Example:



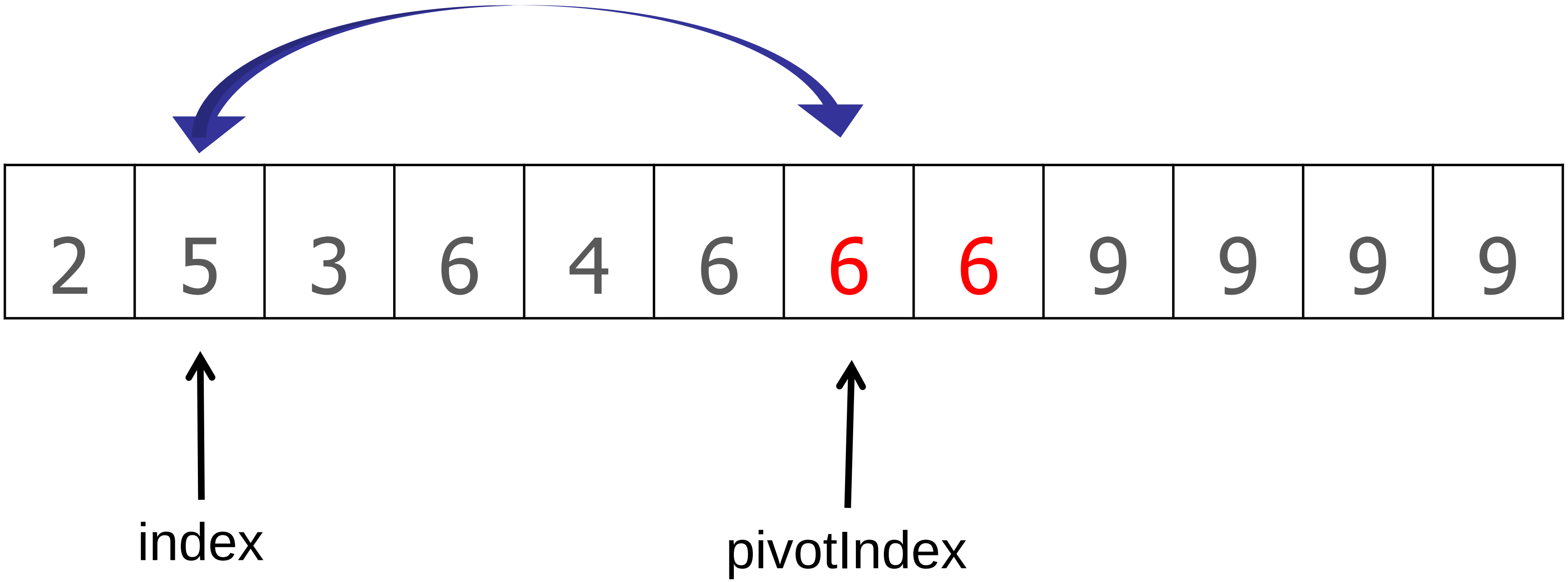
Duplicates

Example:



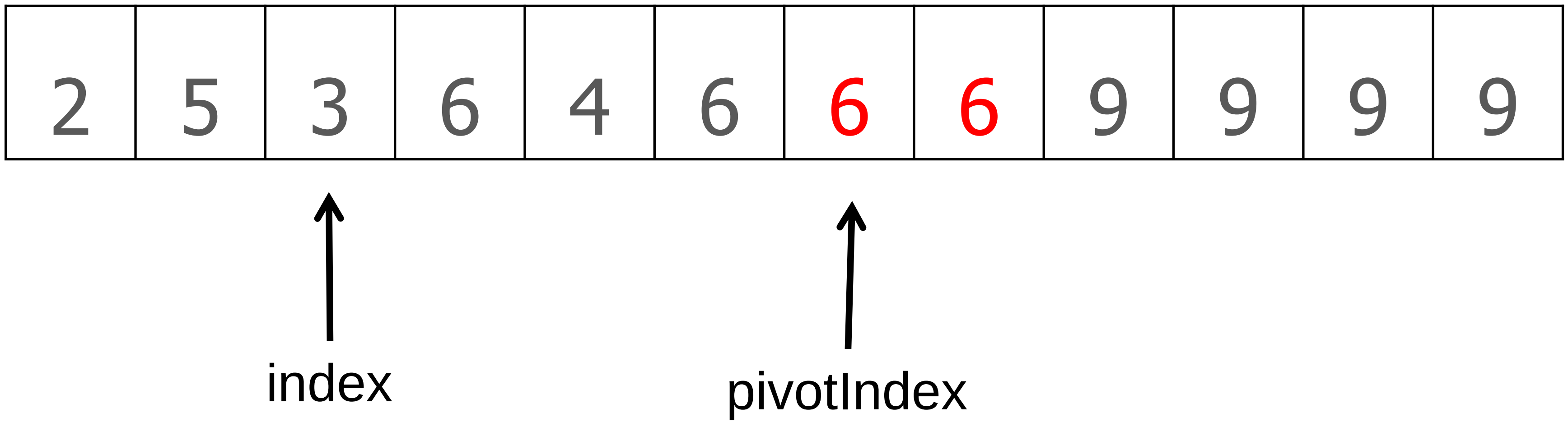
Duplicates

Example:



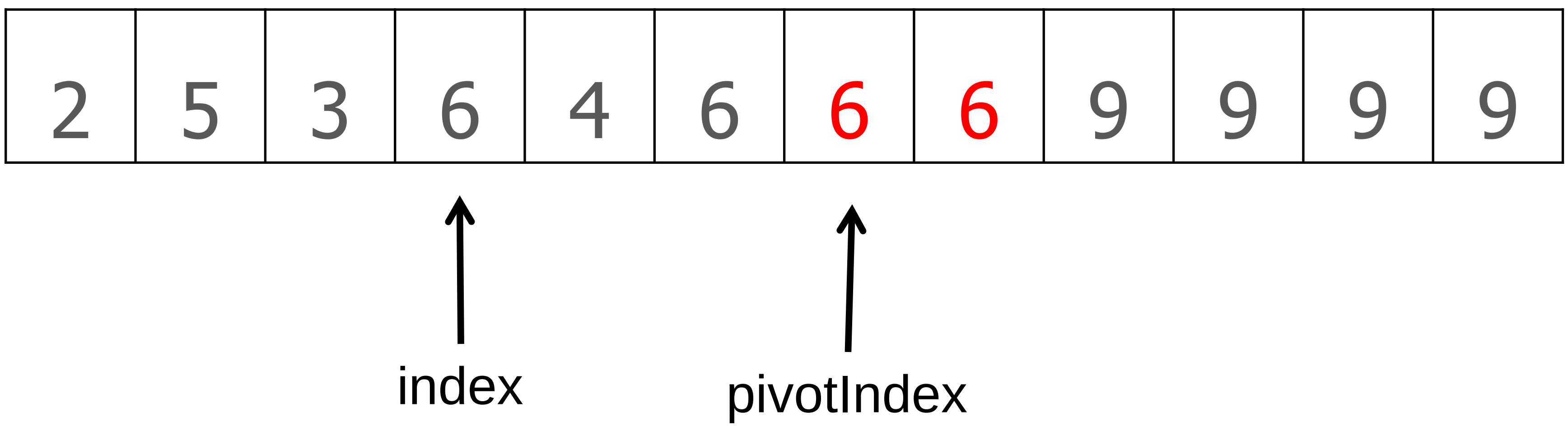
Duplicates

Example:



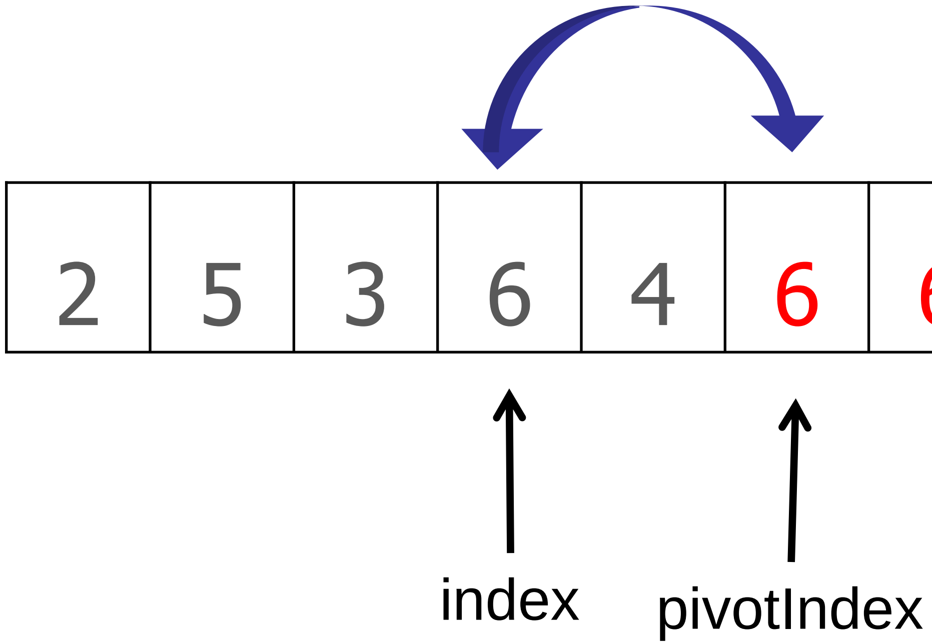
Duplicates

Example:



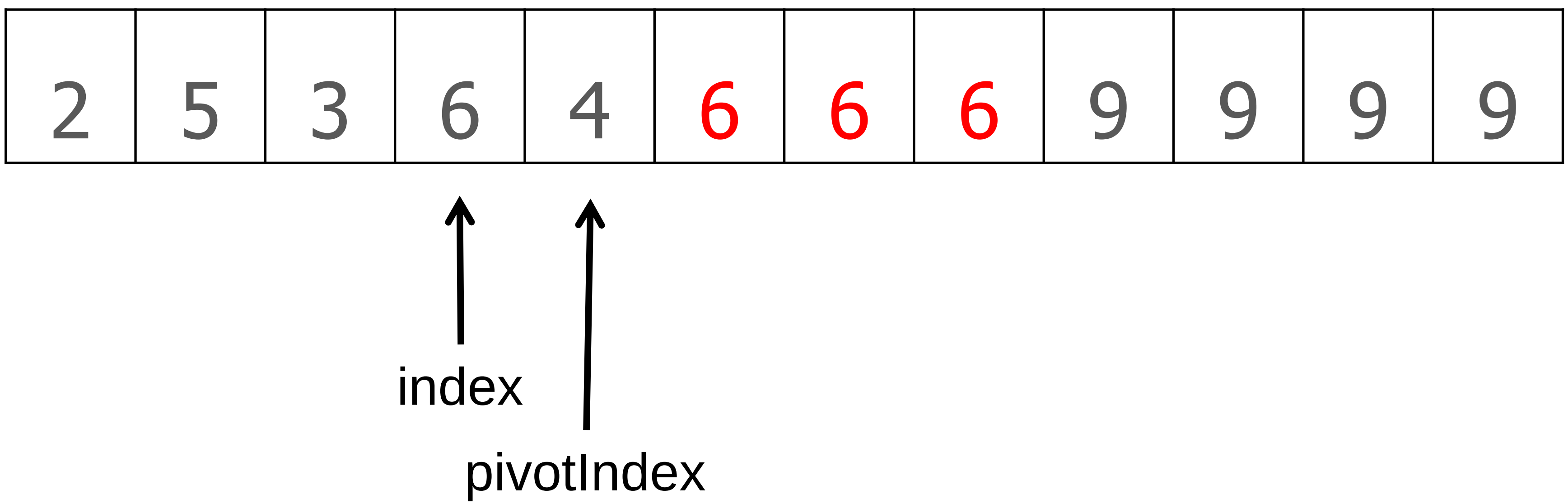
Duplicates

Example:



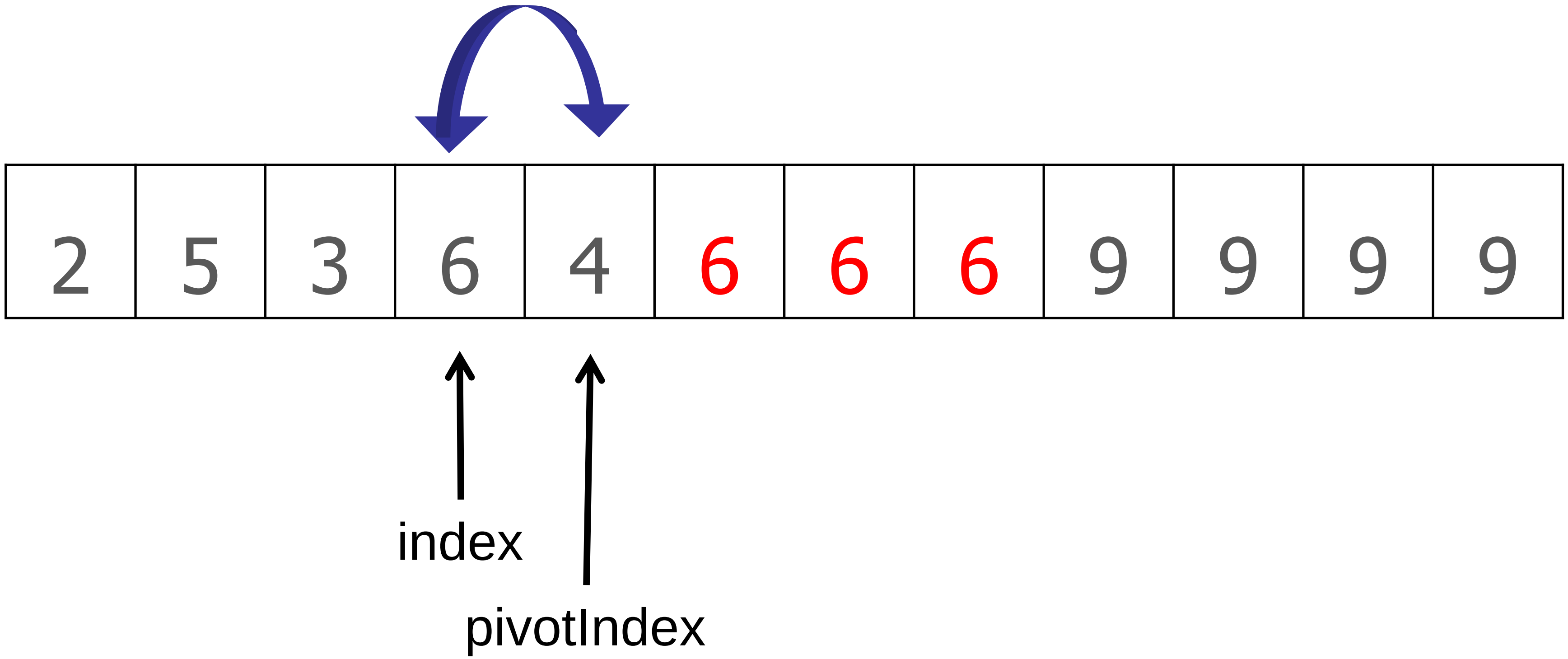
Duplicates

Example:



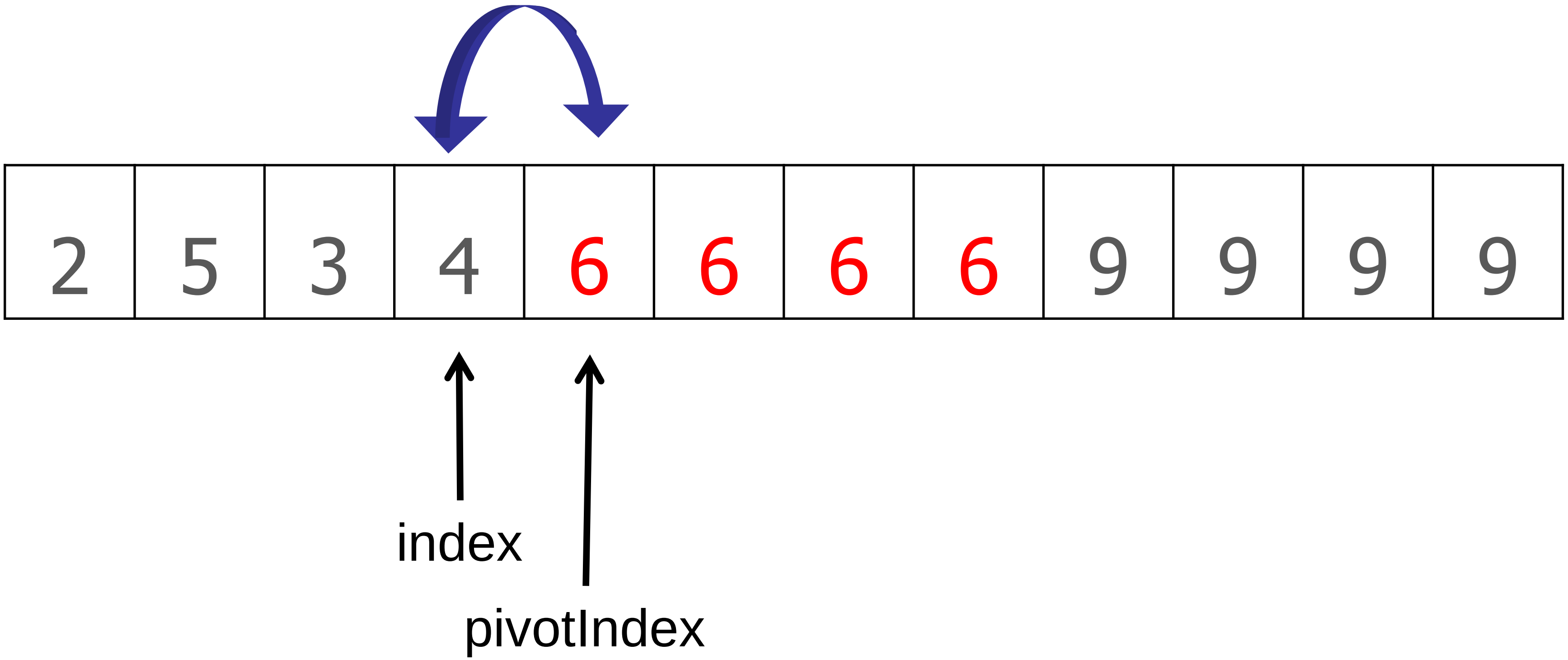
Duplicates

Example:



Duplicates

Example:



Duplicates

Example:

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

index

pivotIndex

Partition

```
packDuplicates(A[1..n], n, pivotIndex)
    pivot = A[pivotIndex];
    index = 1;
    while (index < pivotIndex)
        if (A[index] == pivot) {
            pivotIndex--;
            swap(A[index], A[pivotIndex]);
        }
        else {
            index ++;
        }
    }
```

Duplicates

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

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

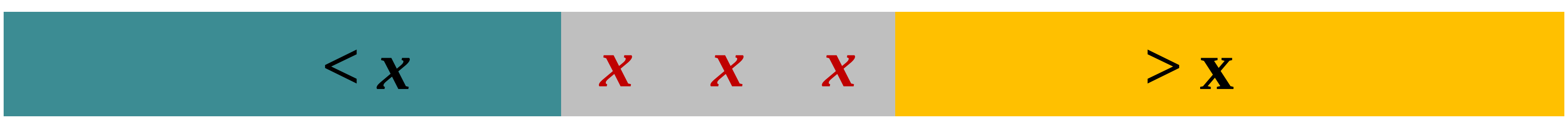
    else

        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)

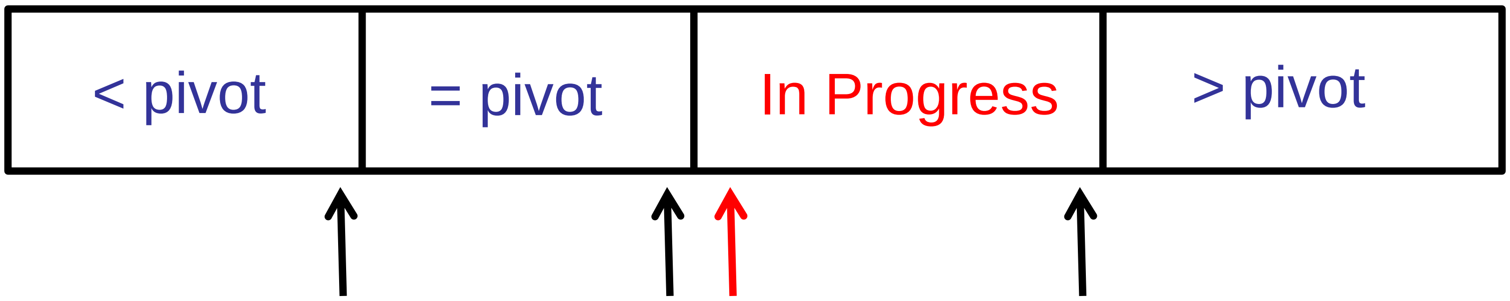


Duplicates

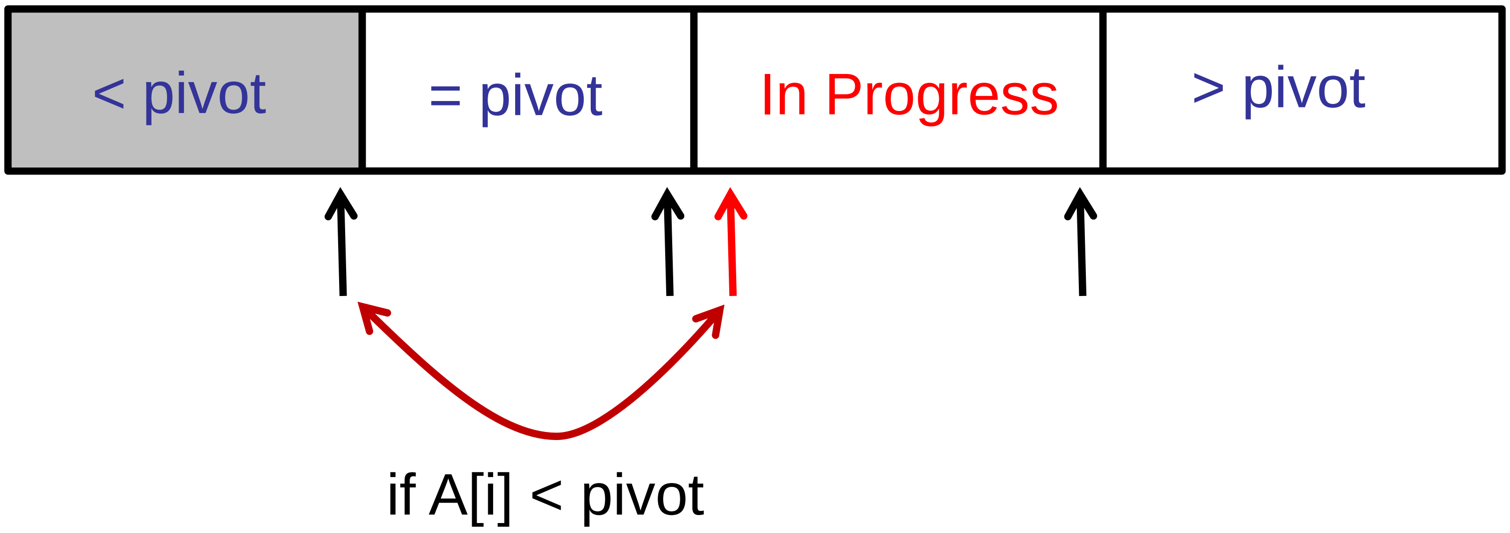
3-Way Partitioning

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

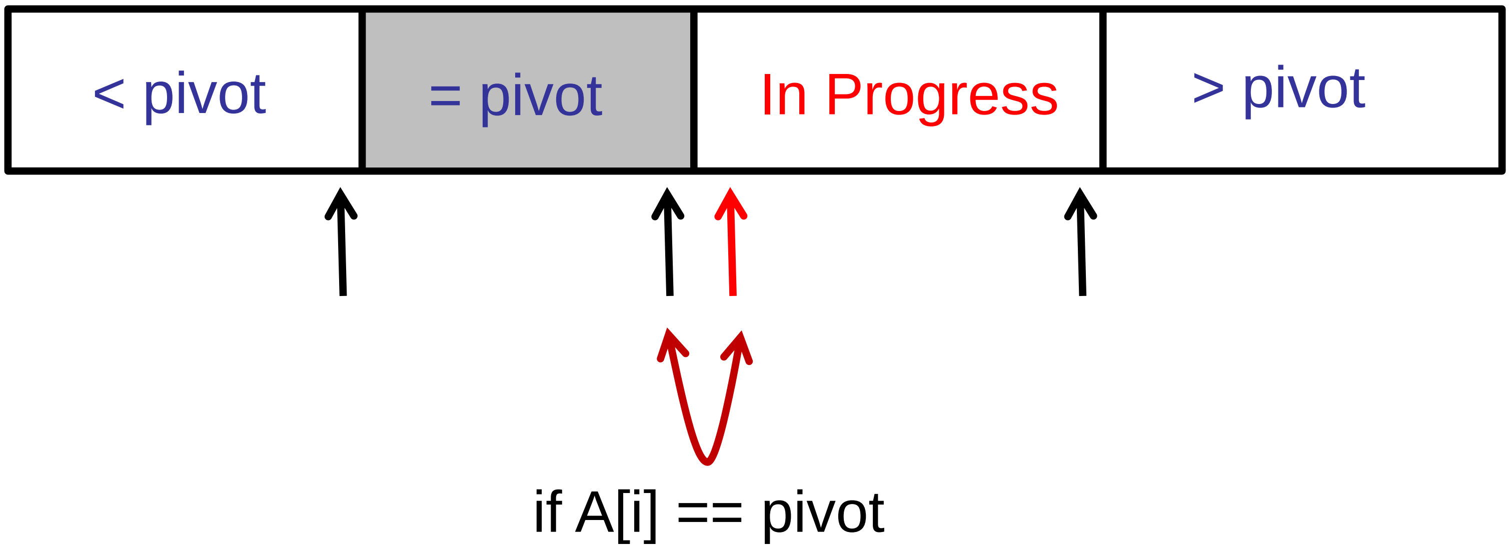
3-Way Partitioning



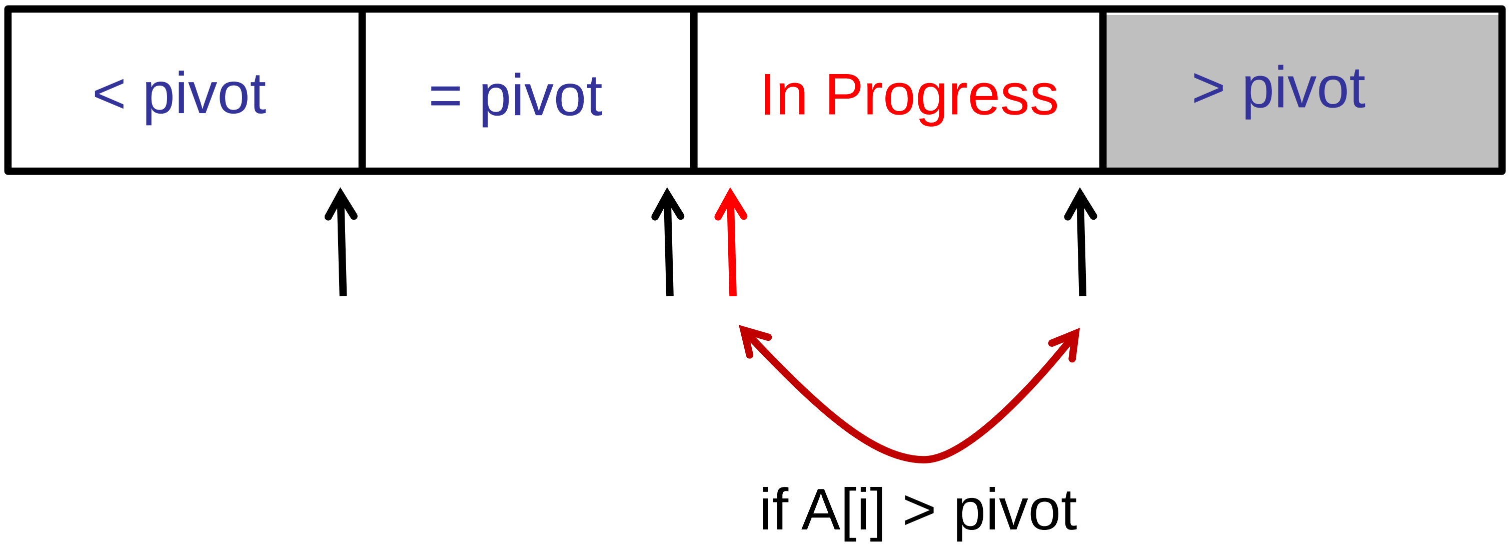
3-Way Partitioning



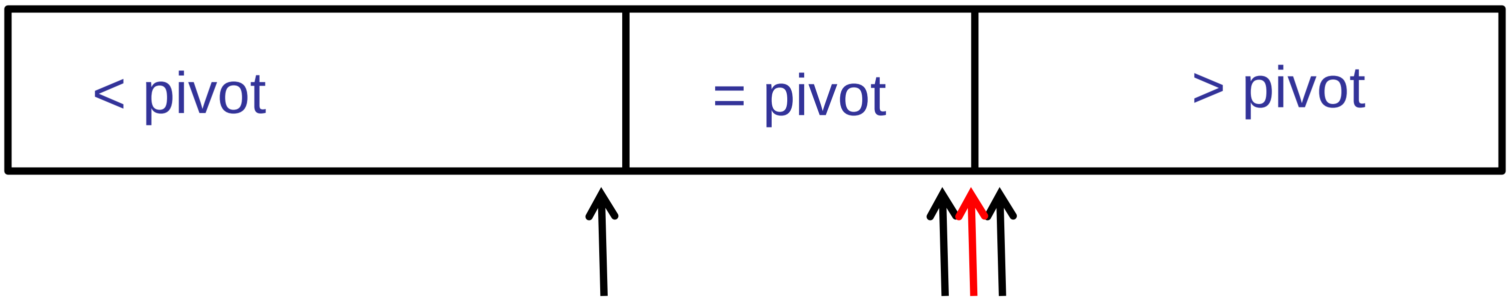
3-Way Partitioning



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*)

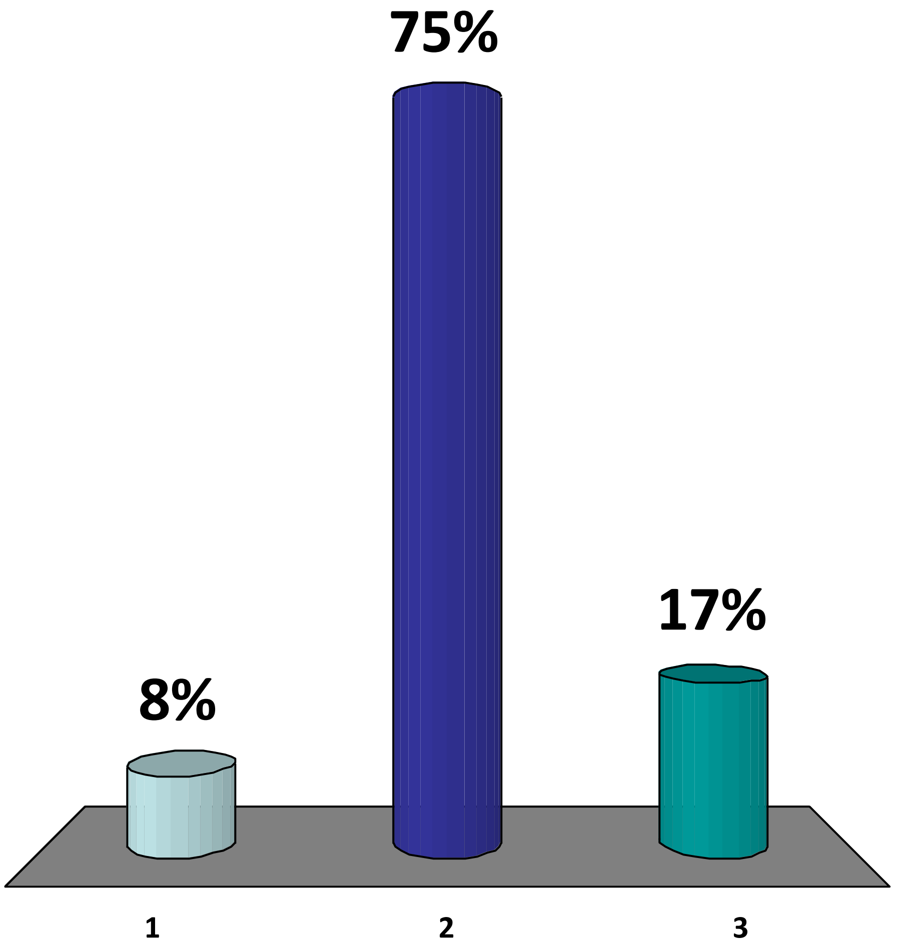
x = QuickSort($A[1..p–1]$, p–1)

y = QuickSort($A[p+1..n]$, n–p)

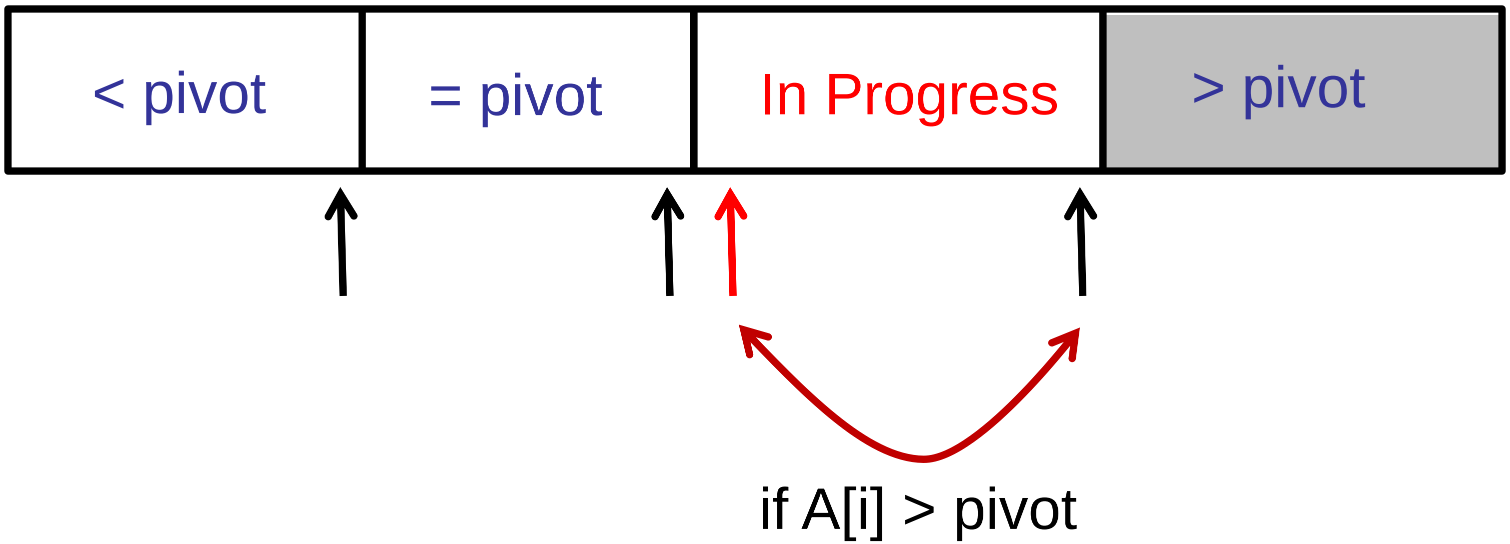
$< x$	x	x	x	$> x$

Is QuickSort stable?

1. Yes
2. No
3. Dragons?



QuickSort is not stable



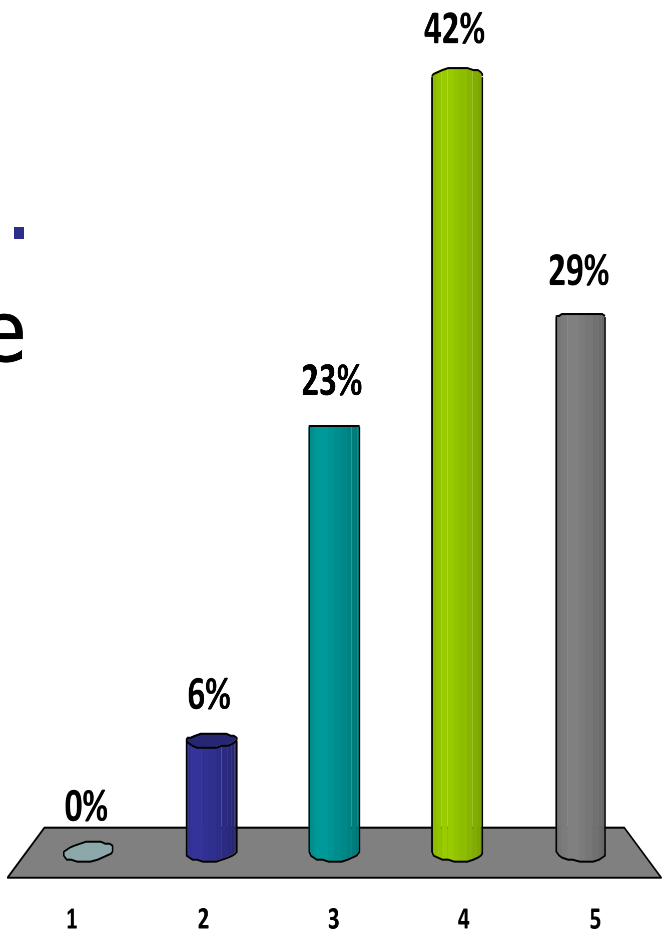
Choice of Pivot

Options:

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

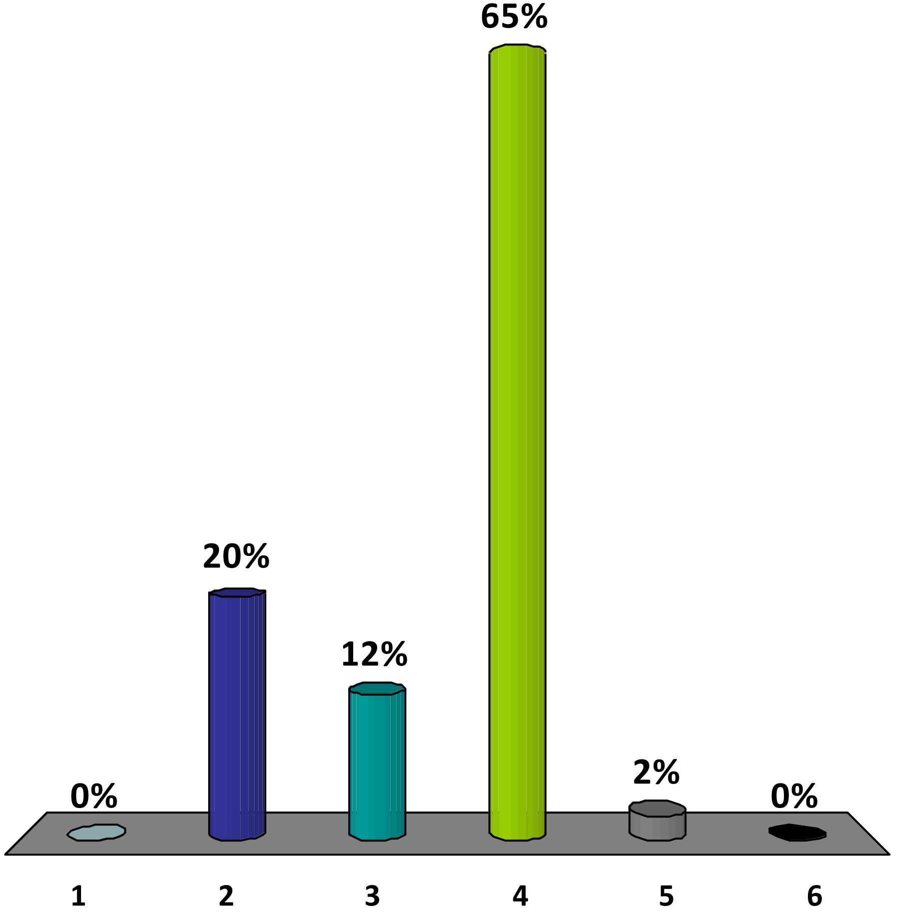
What is a good (deterministic) choice for the pivot?

1. The first element A[1].
2. The last element A[n].
3. The middle element A[n/2].
4. The median of the first, the last, and the middle element.
5. It does not matter.



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

Choose $A[1]$ for pivot:

100	99	98	97	96	95	94	93	92

99	98	97	96	95	94	93	92	100

Choice of Pivot

Choose $A[1]$ for pivot:

99	98	97	96	95	94	93	92	100

98	97	96	95	94	93	92	99	100

Choice of Pivot

Choose $A[1]$ for pivot:

98	97	96	95	94	93	92	99	100

$\Downarrow$

97	96	95	94	93	92	98	99	100

Choice of Pivot

Choose $A[1]$ for pivot:

98	97	96	95	94	93	92	99	100

$\Downarrow$

97	96	95	94	93	92	98	99	100

Choice of Pivot

Sorting the array takes *n* executions of partition.

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

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

98	97	96	95	94	93	92	99	100

97	96	95	94	93	92	98	99	100

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

50% 1. $T(n) = 2T(n/2) + cn$

19% 2. $T(n) = 2T(n/2) + c$

5% 3. $T(n) = T(n/2) + cn$

17% ✓ 4. $T(n) = T(n-1) + T(1) + cn$

7% 5. $T(n) = T(n-1) + T(1) + c$

2% 6. $T(n) = T(n/4) + T(3n/4) + cn$

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

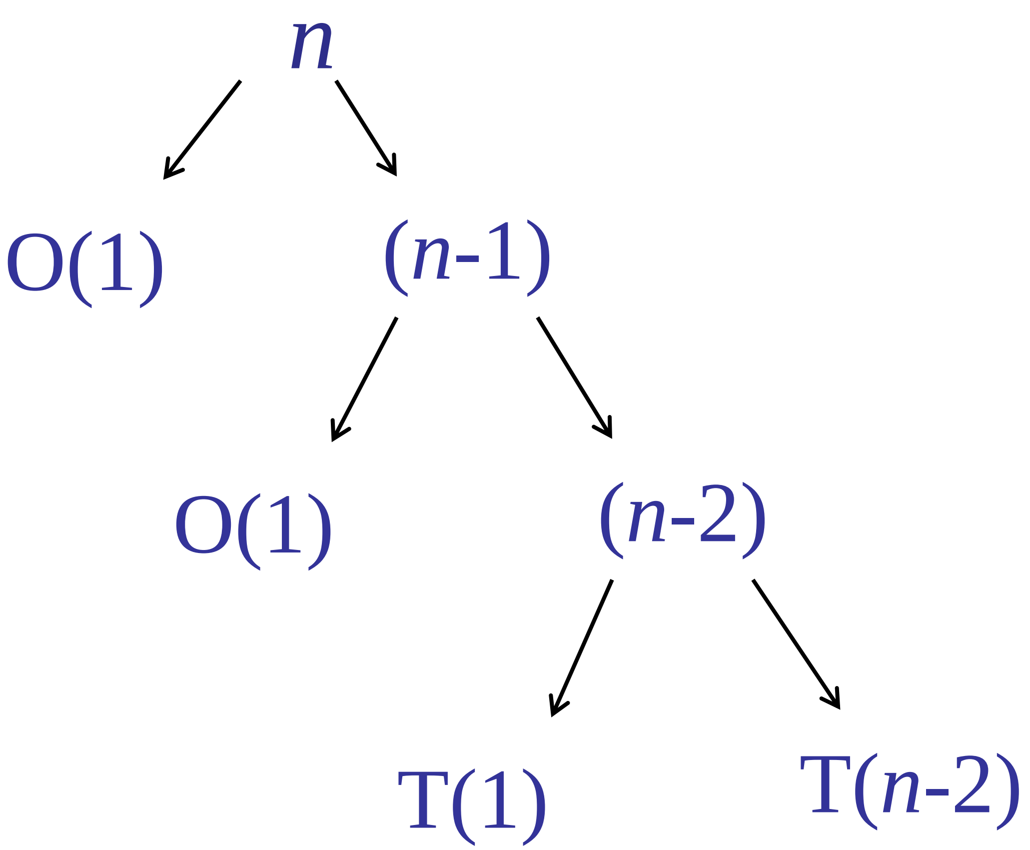
Deterministic QuickSort

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

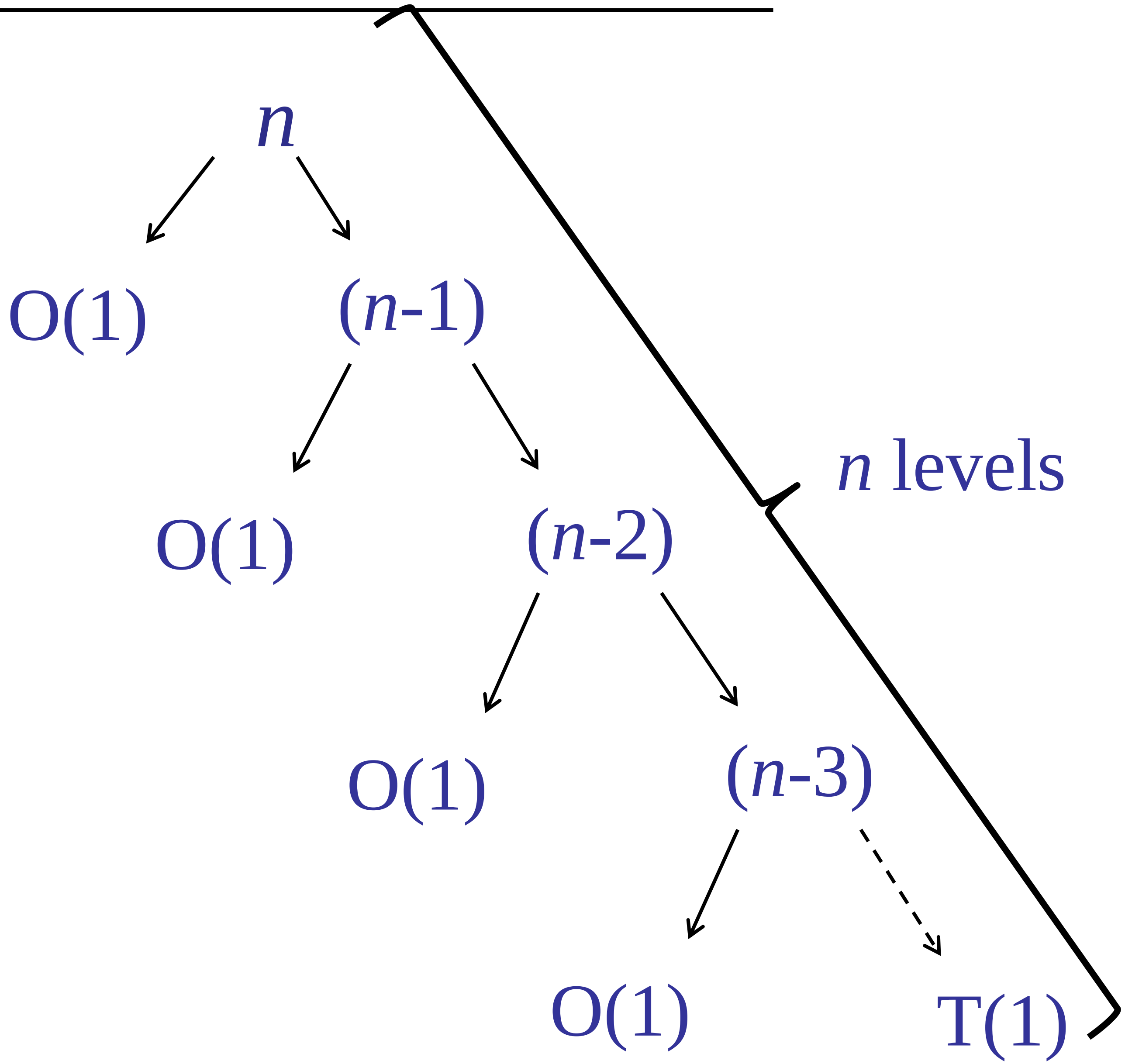
Deterministic QuickSort

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

Deterministic QuickSort

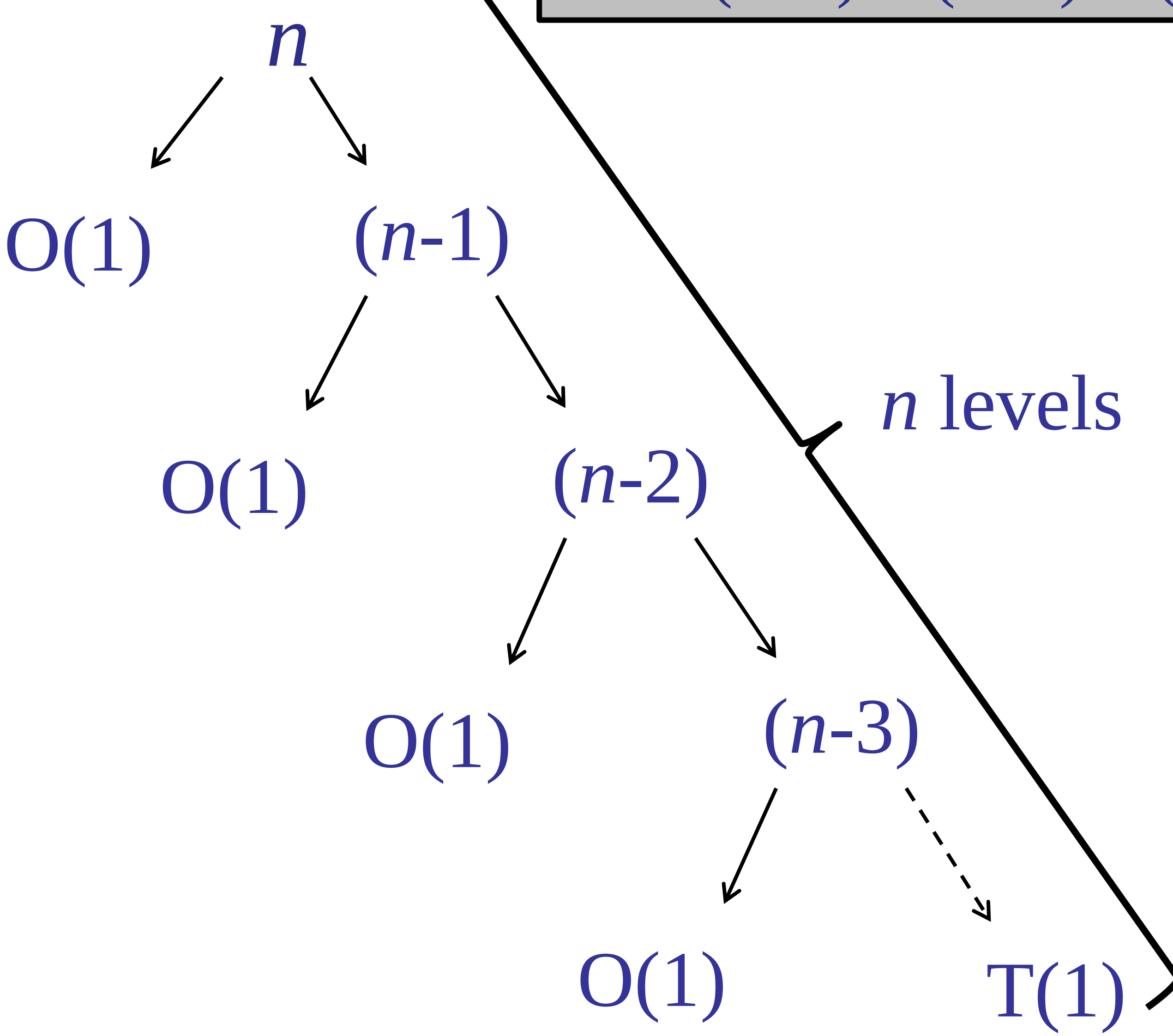


Deterministic QuickSort



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

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

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

8% 1. $O(\log n)$

10% 2. $O(n)$

78% ✓ 3. $O(n \log n)$

2% 4. $O(n^2)$

0% 5. $O(n^3)$

% 6. None of the above.

Lucky QuickSort

If we split the array evenly:

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

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

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

QuickSort Summary

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

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

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 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*)

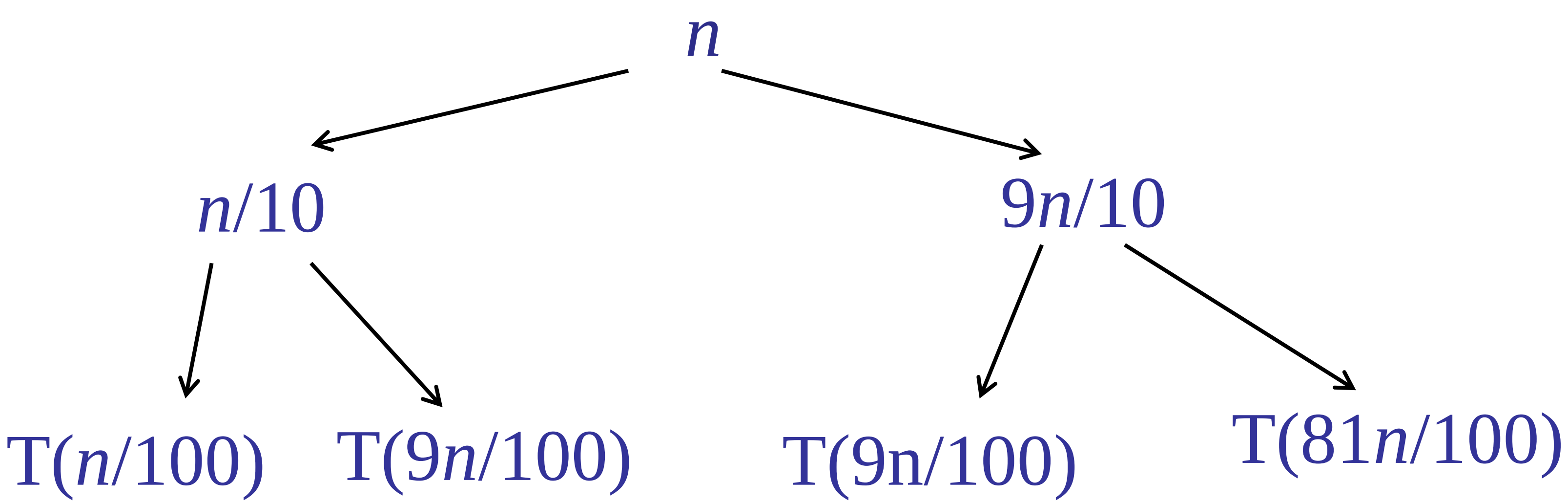
QuickSort Analysis

$k = n/10$

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

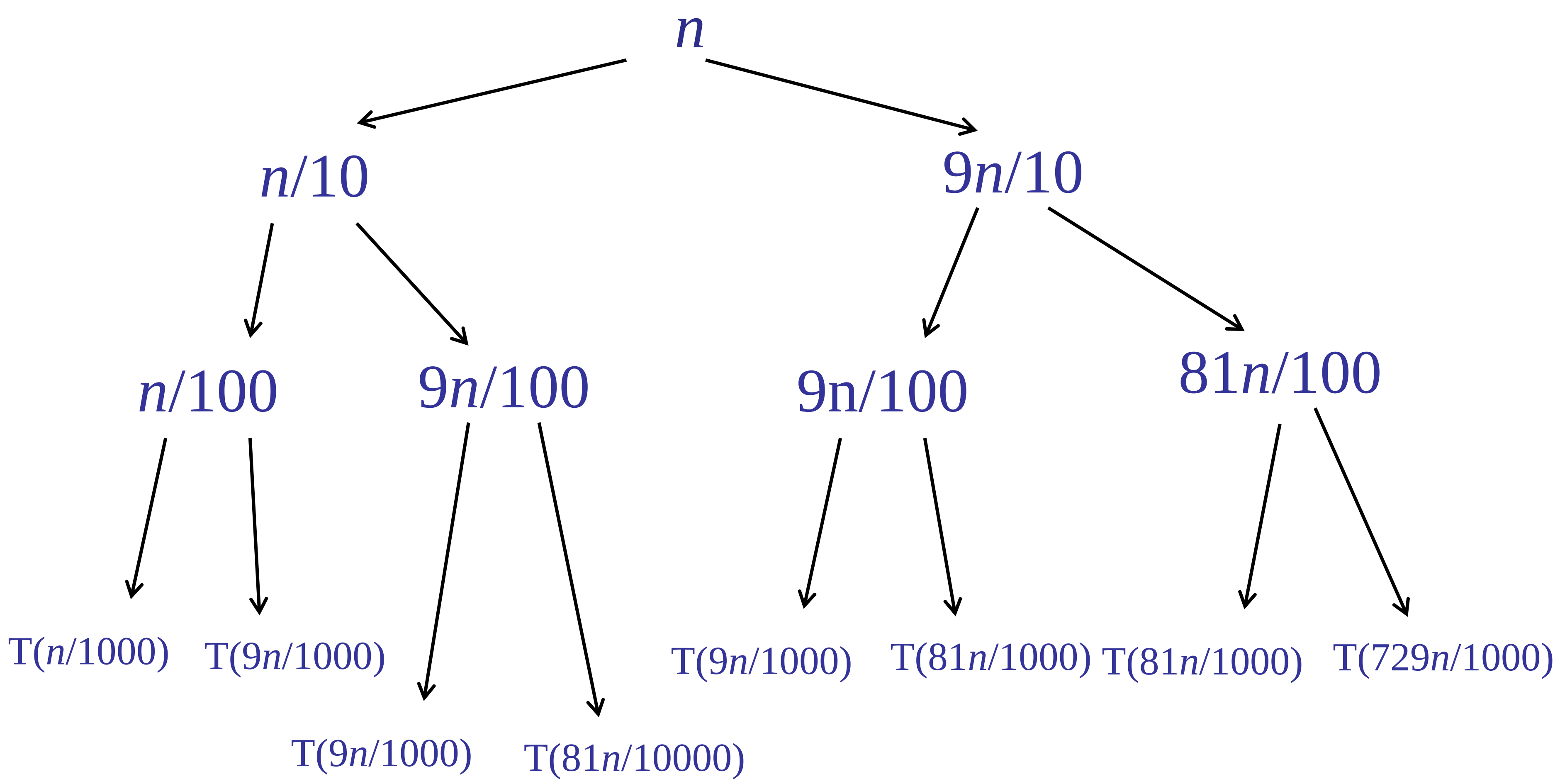
QuickSort Analysis

$k = n/10$



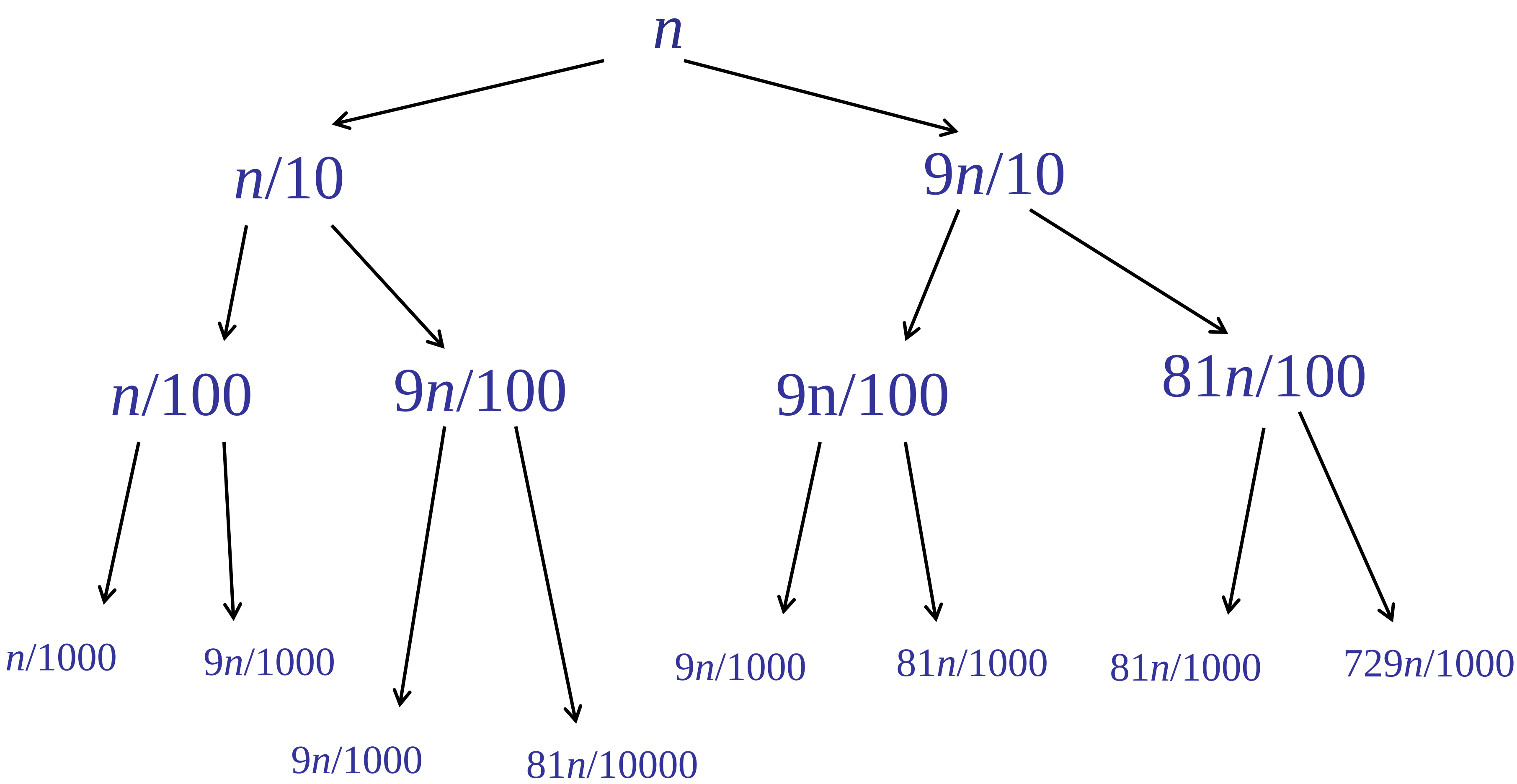
QuickSort Analysis

$k = n/10$

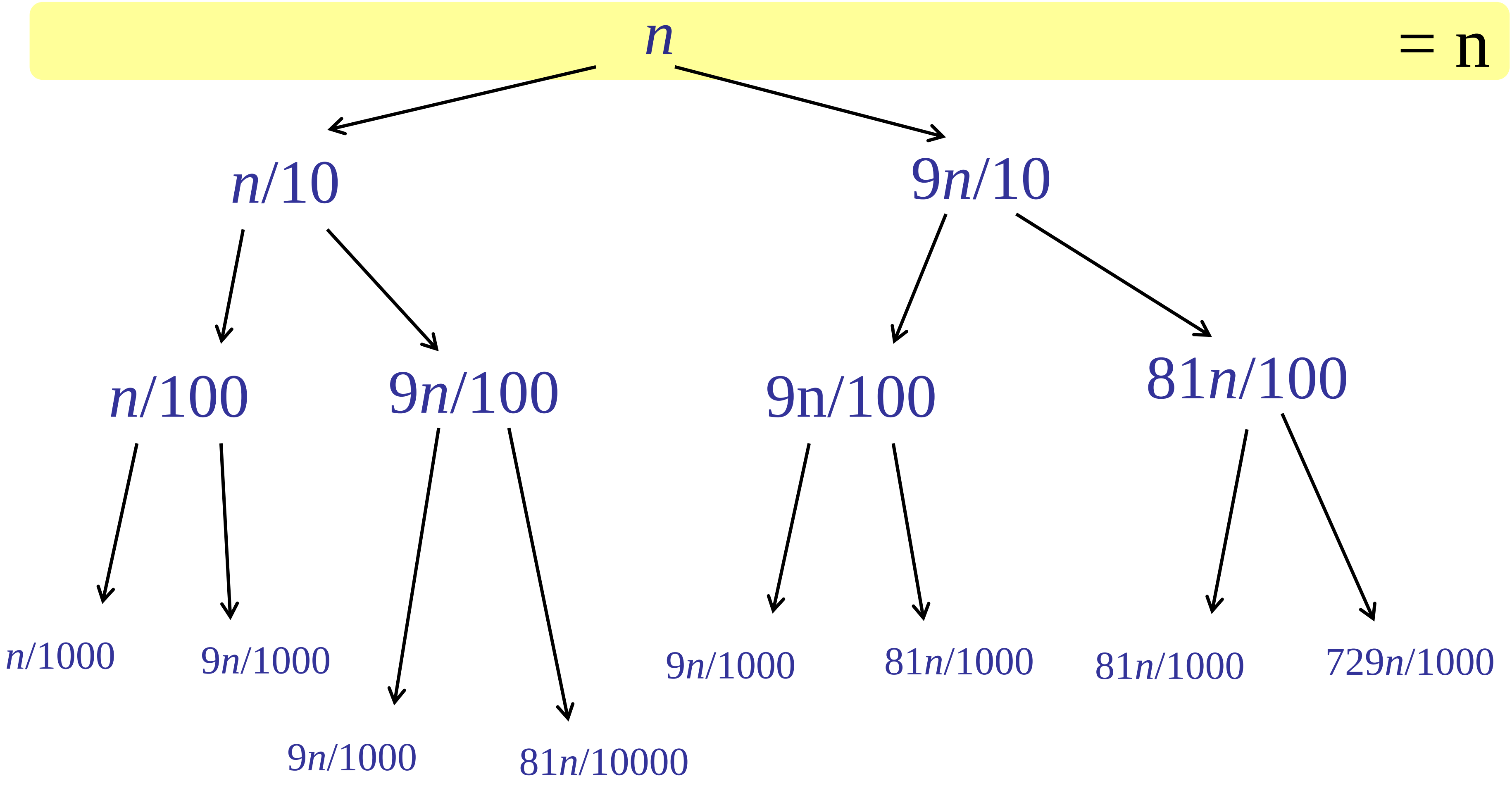


QuickSort Analysis

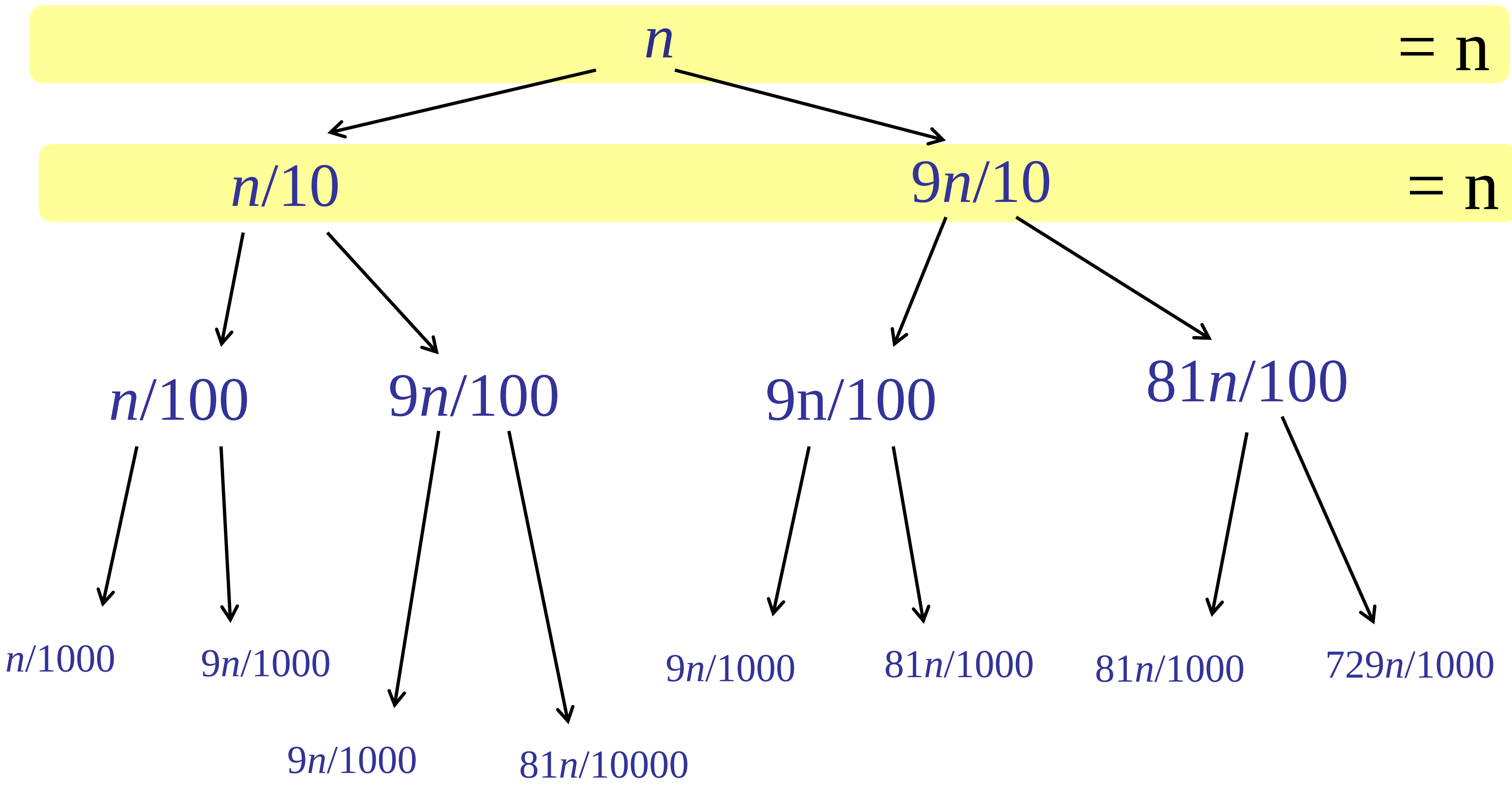
$k = n/10$



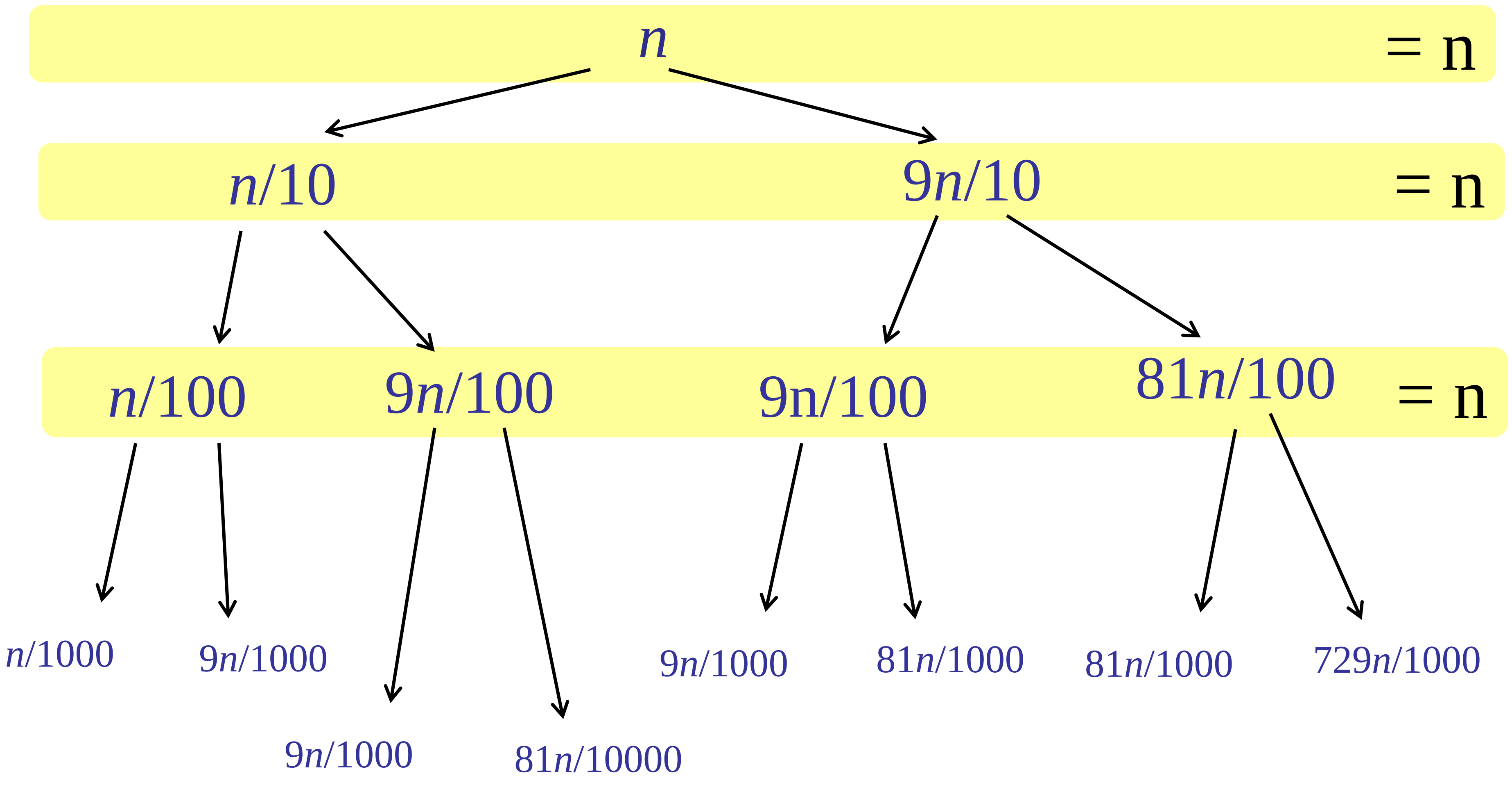
QuickSort Analysis



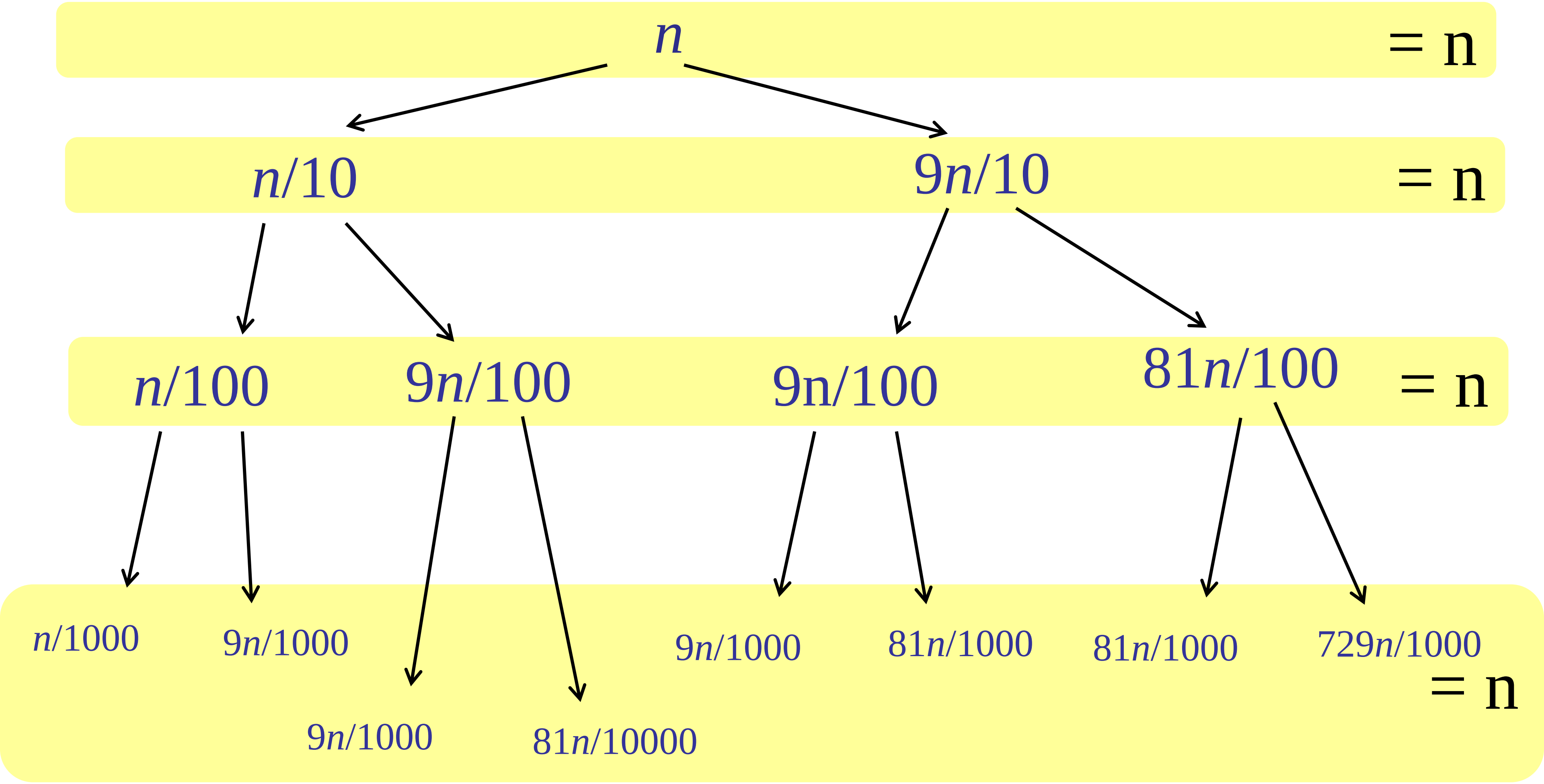
QuickSort Analysis



QuickSort Analysis

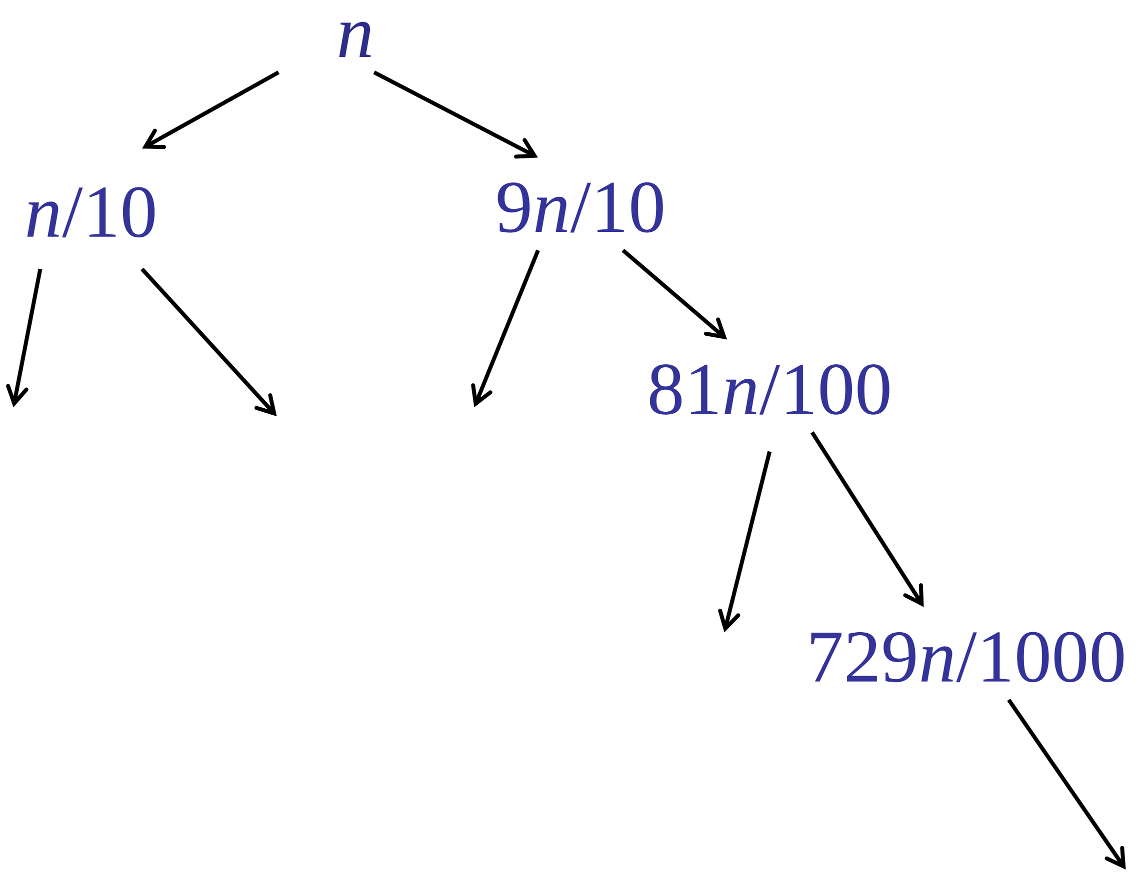


QuickSort Analysis



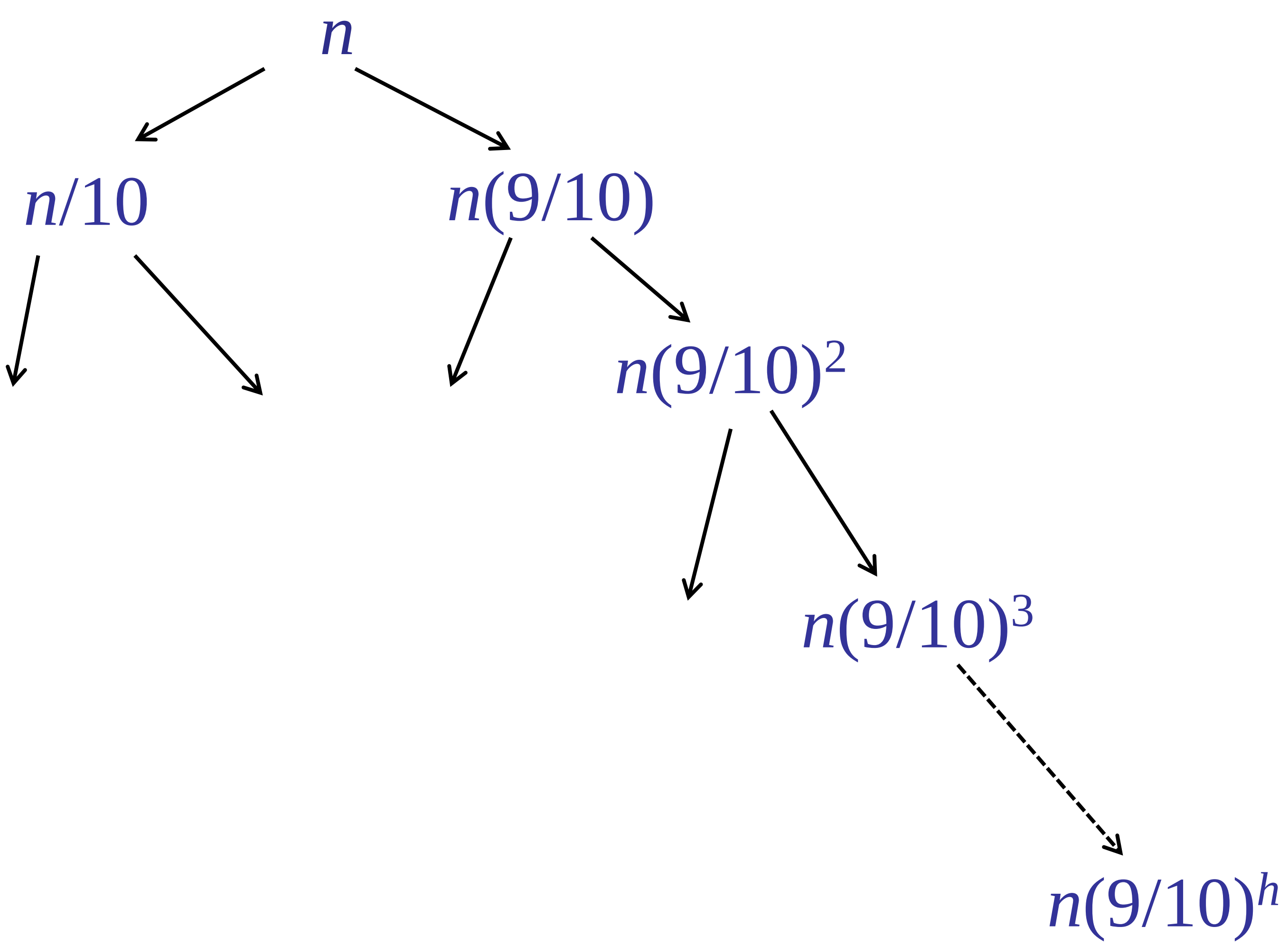
How many levels??

QuickSort Analysis



How many levels??

QuickSort Analysis



How many levels??

QuickSort Analysis

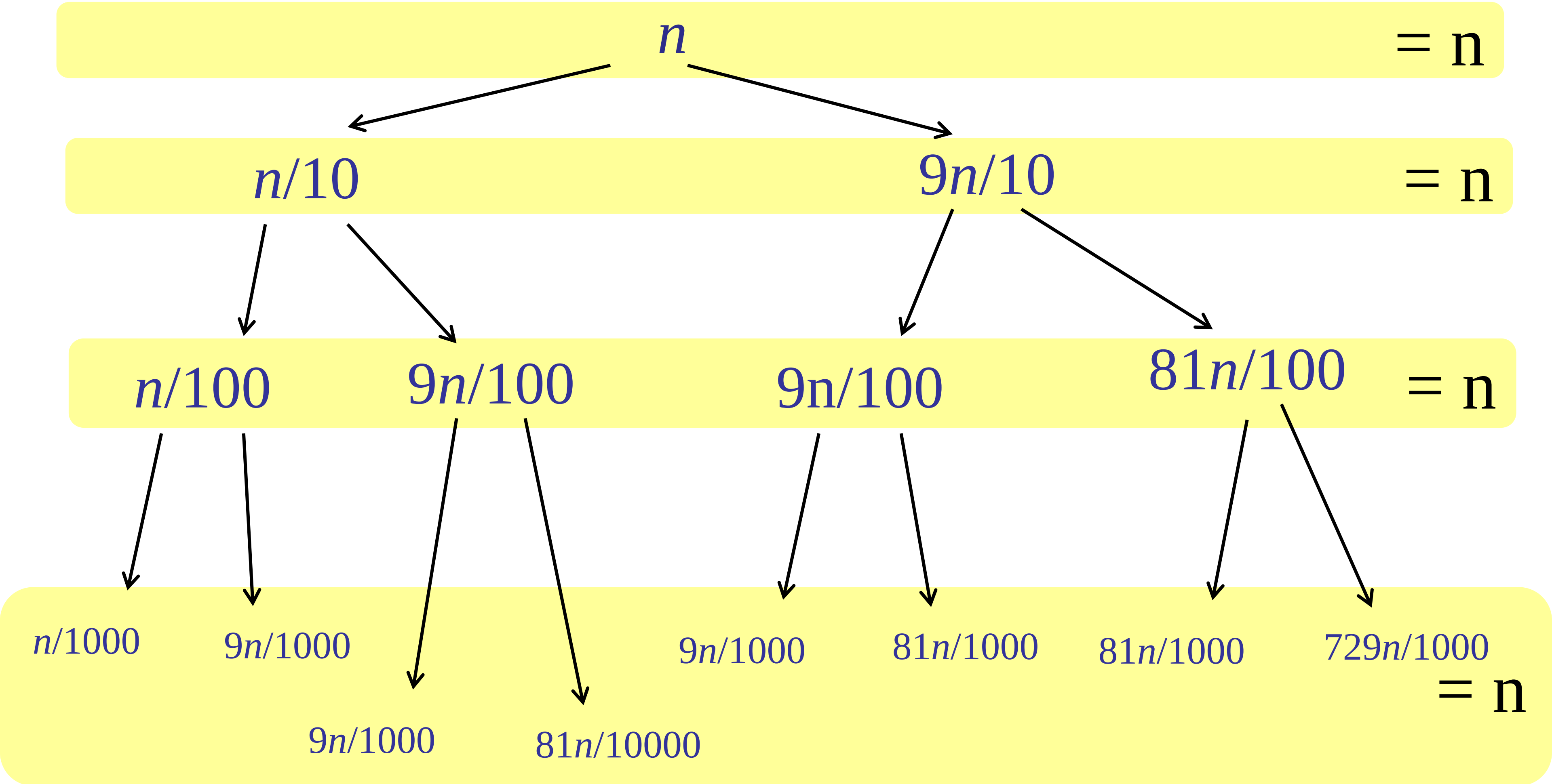
Maximum number of levels:

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

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

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

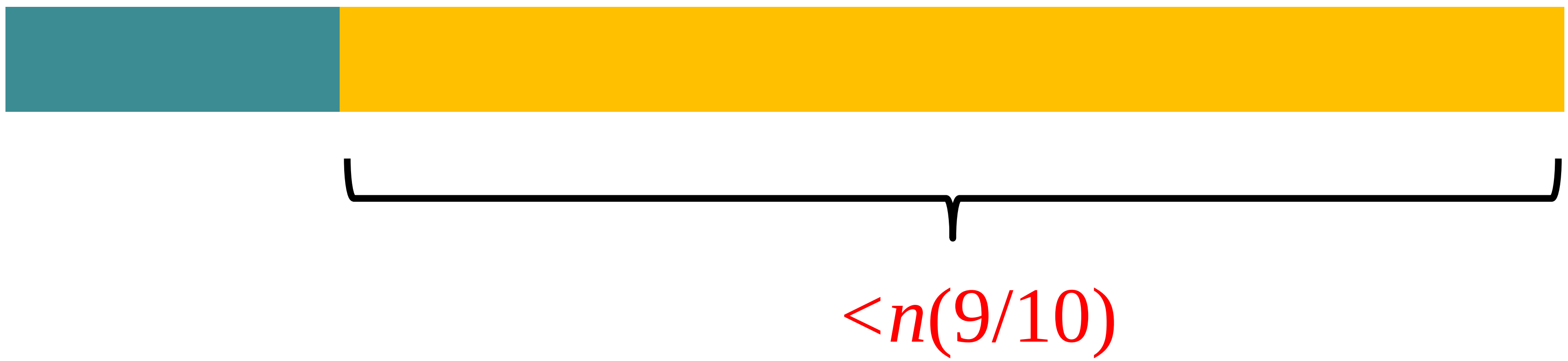
QuickSort Analysis



How many levels??

QuickSort Analysis

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



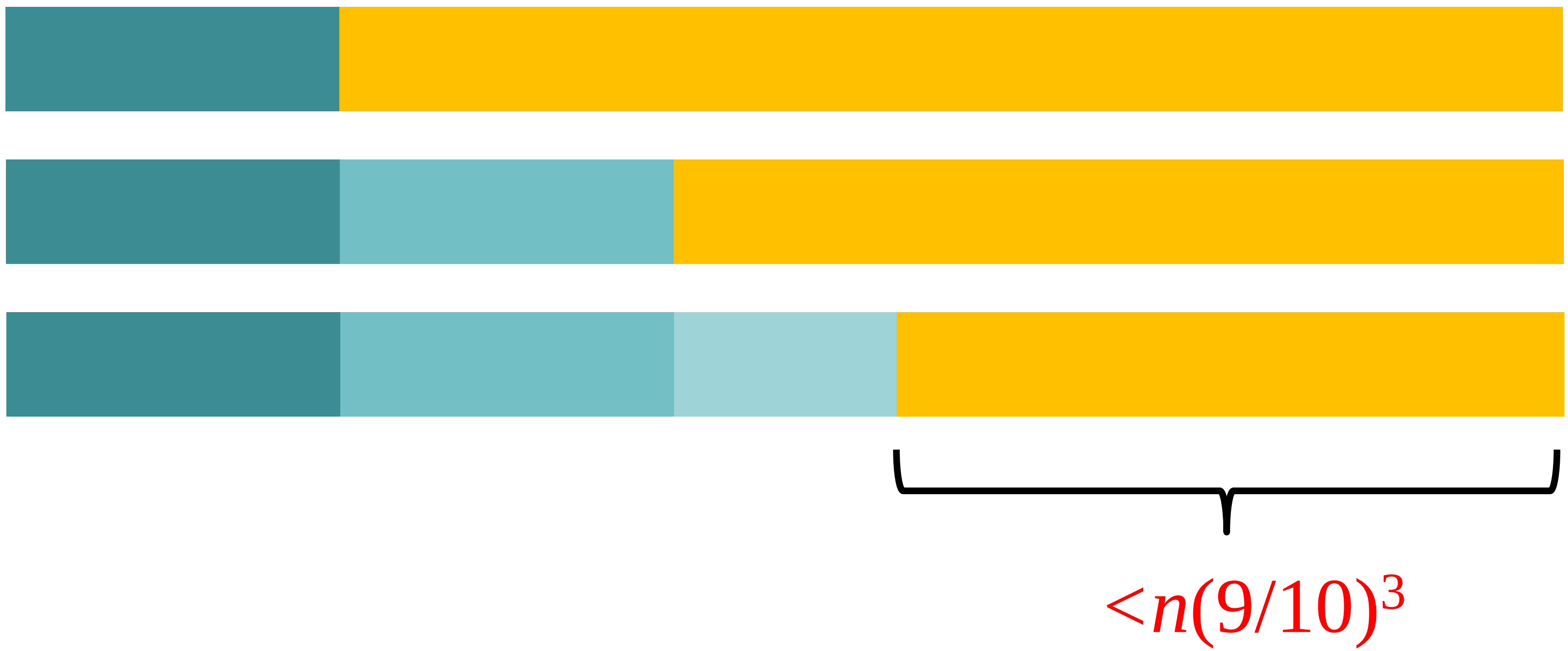
QuickSort Analysis

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

$< n(9/10)^2$

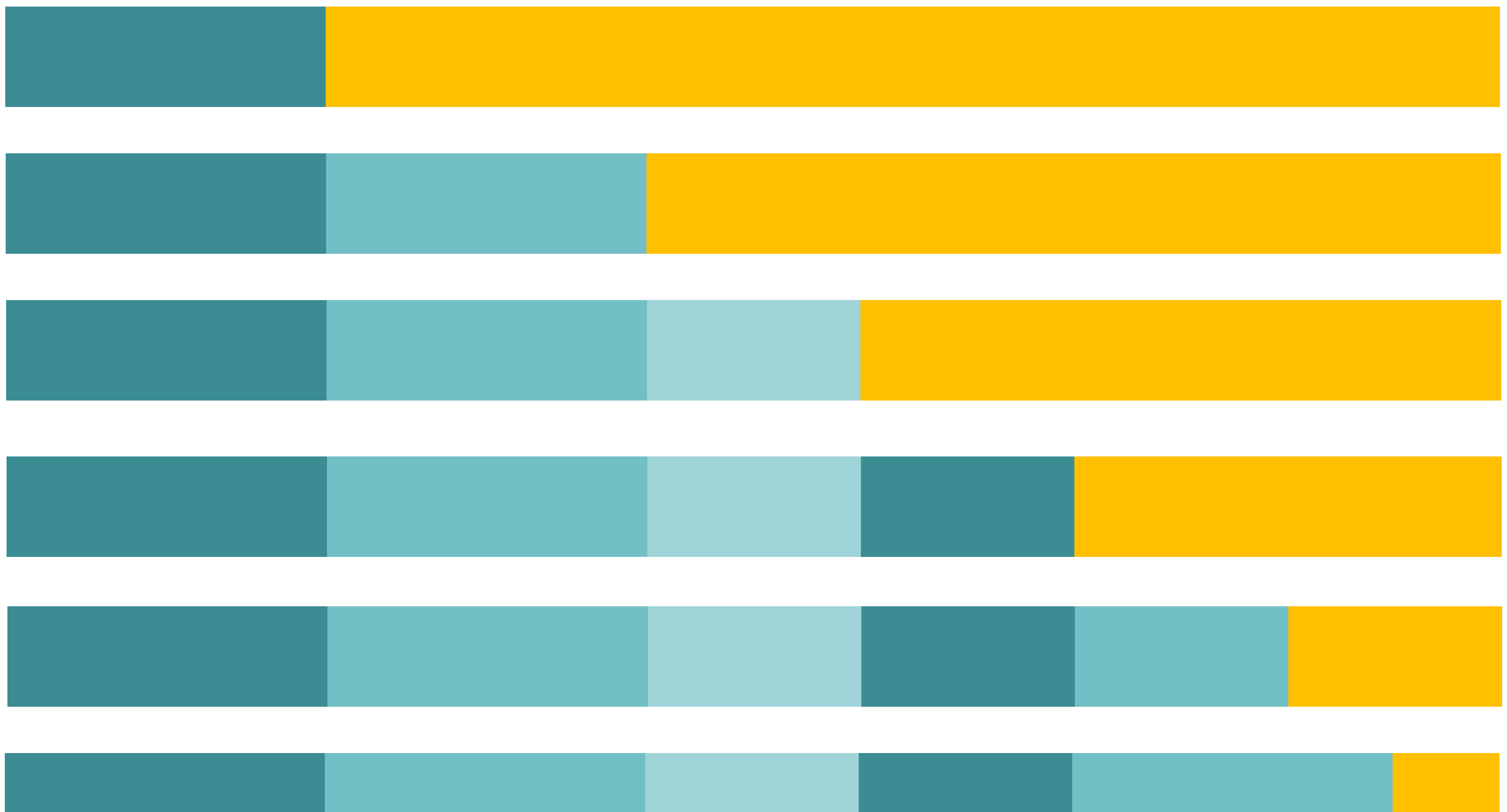
QuickSort Analysis

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



QuickSort Analysis

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



QuickSort Analysis

Maximum number of levels:

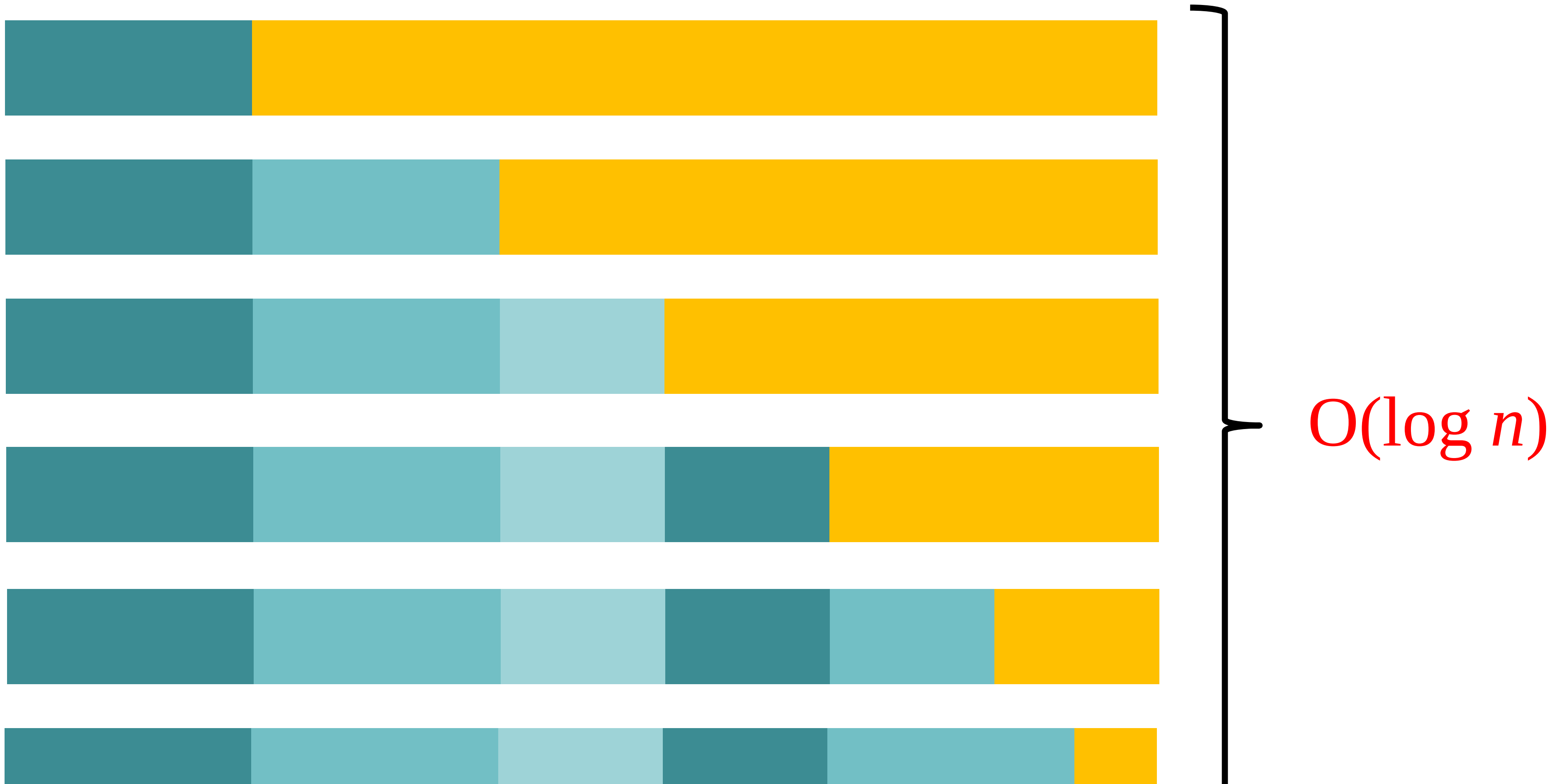
$$1 = n(9/10)^h$$

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

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

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

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

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

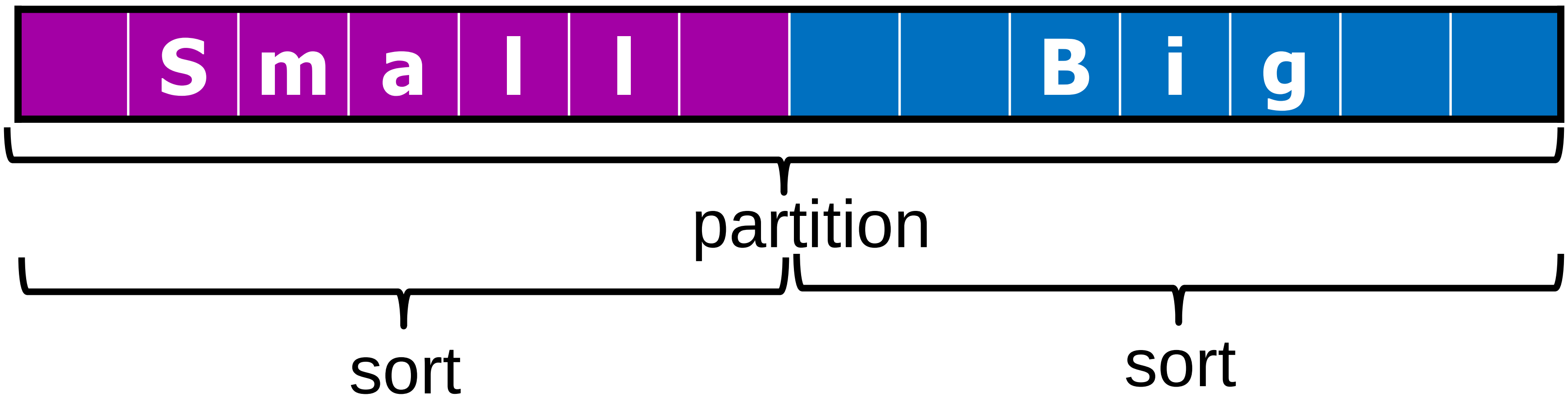
Randomized algorithm:

- 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)
```



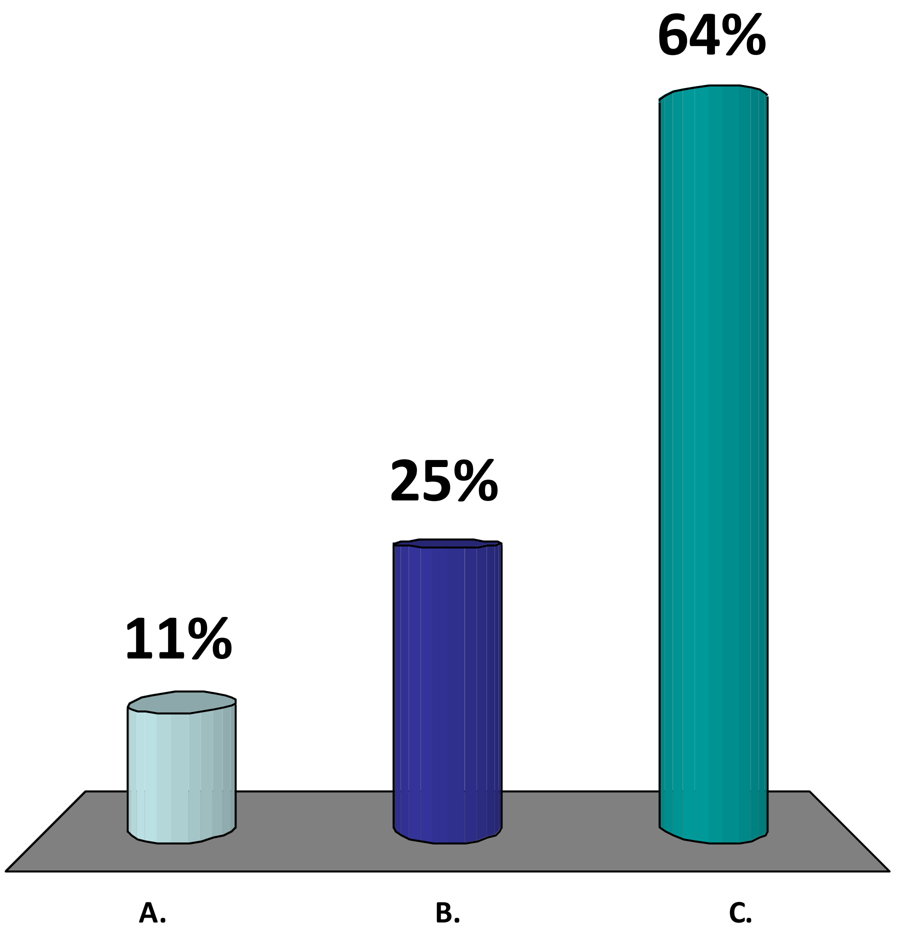
Paranoid QuickSort

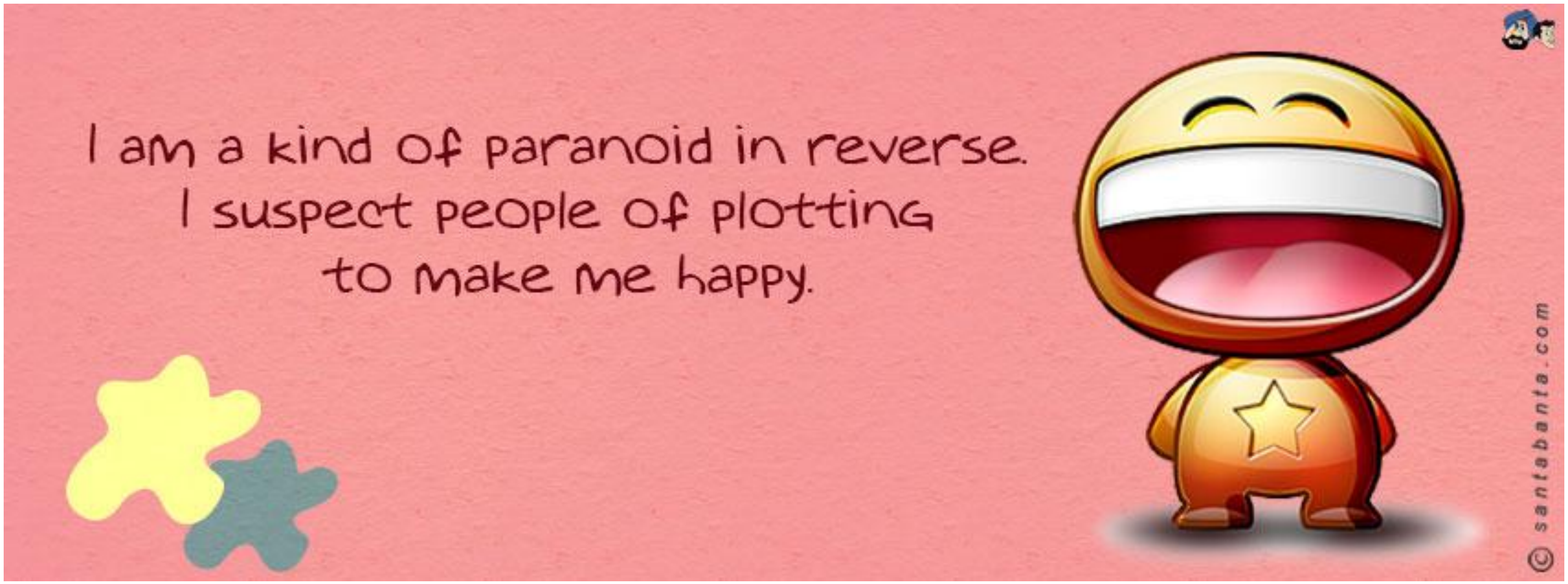
```
ParanoidQuickSort(A[1..n], n)
    if (n == 1) then return;
    else
        repeat
            pIndex = random(1, n)
            p = partition(A[1..n], n, pIndex)
        until p > (1/10)n and p < (9/10)

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

Are you paranoid?

A. Yes
B. No
C. Who is this!?




I am a kind of paranoid in reverse.
I suspect people of plotting
to make me happy.
© santabanta.com

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

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

```
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

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)

1

O(n)

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

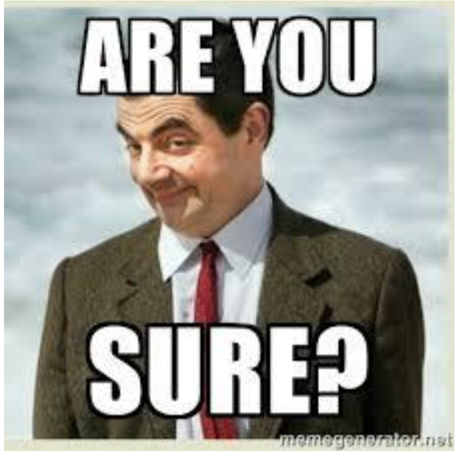
Key claim:

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

Then we know:

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

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



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

Expected value:

- Weighted average

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

- Pr(**A** = 12) = ¼
- Pr(**A** = 60) = ¾

Expected value of A:

$E[A] = (¼)12 + (¾)60 = 48$

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

2 * ¼	=	½
1 * ¼	=	¼
1 * ¼	=	¼
0 * ¼	=	0
		1

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

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

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

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

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

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

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

Probability Theory

Flipping an (unfair) coin:

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

How many flips to get at least one head?

If $p = \frac{1}{2}$, the expected number of flips to get one head equals:

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

Paranoid QuickSort

```
ParanoidQuickSort(A[1..n], n)
    if (n == 1) then return;
    else
        repeat
                pIndex = random(1, n)
                p = partition(A[1..n], n, pIndex)
        until p > (1/10)n and p < (9/10)

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

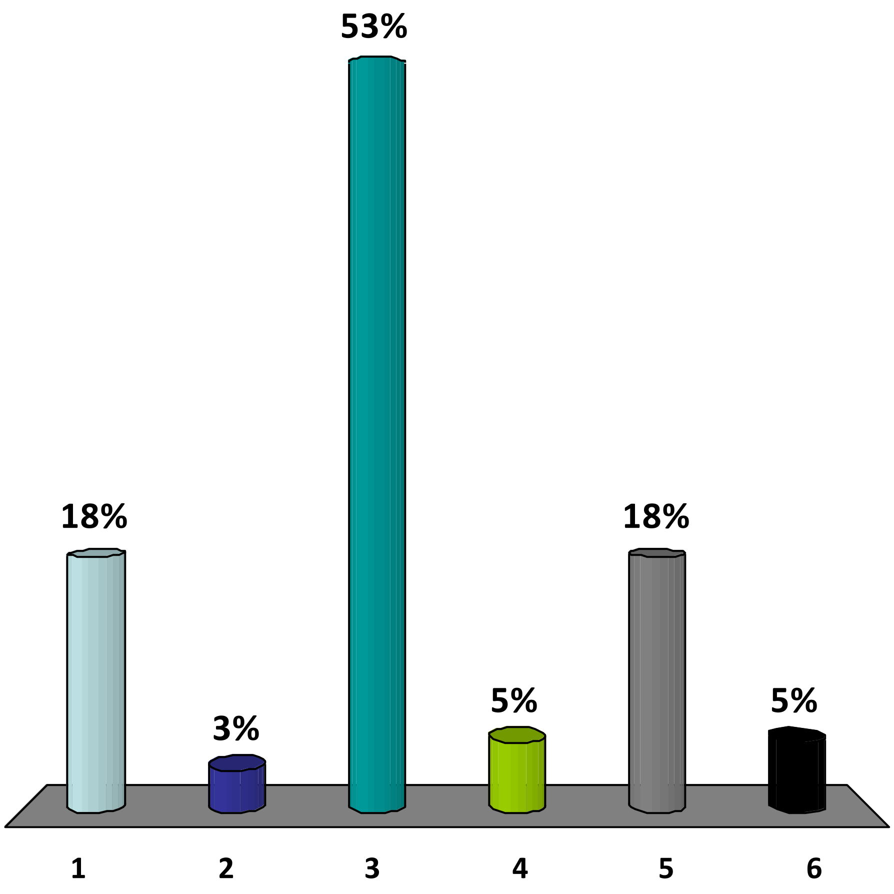
QuickSort Partition

Remember:

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

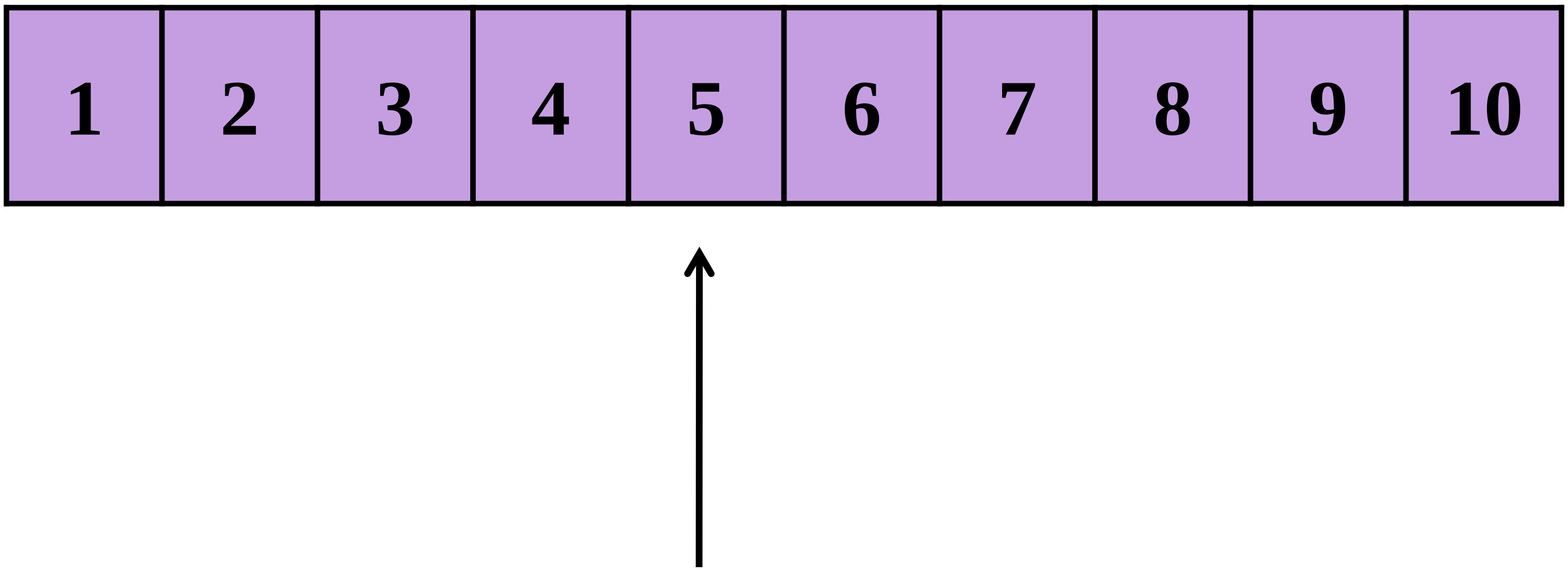
If we choose a pivot at random, what is the probability that it is <u>good</u>?

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:

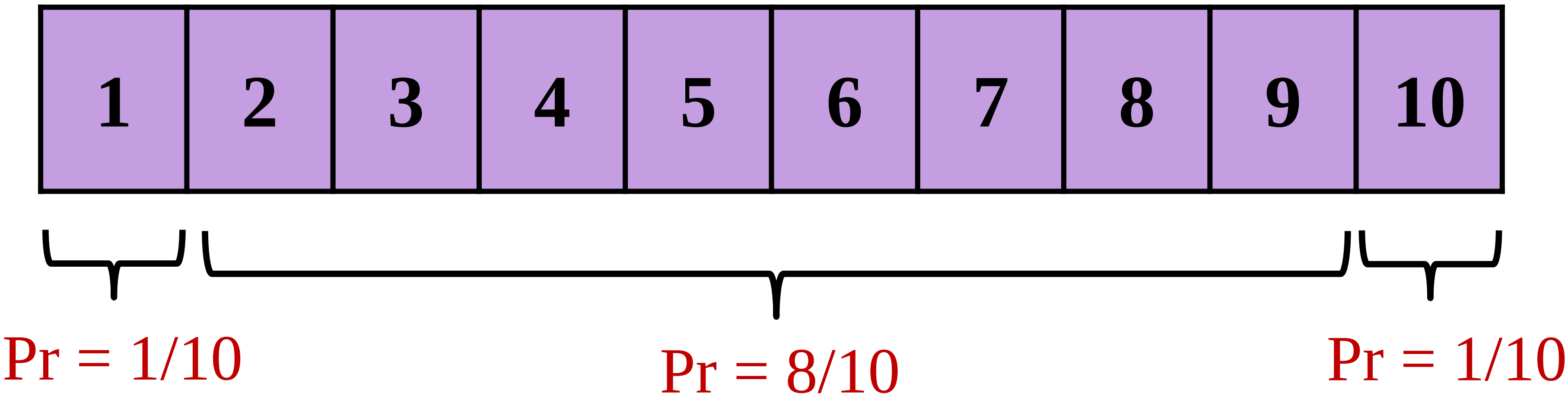
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

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:

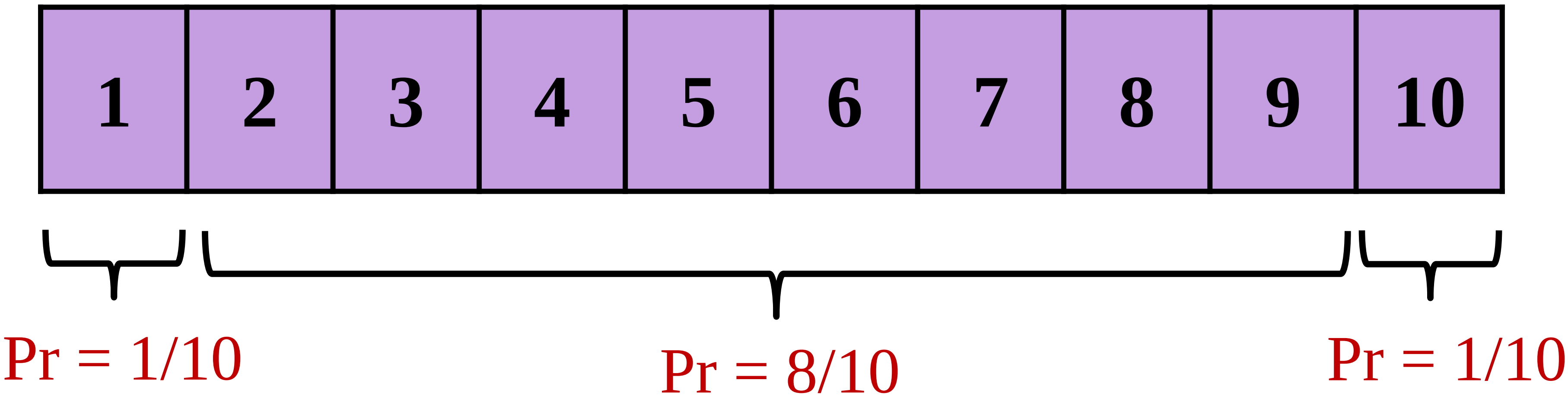


Choose a random point at which to partition.

- 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:

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

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

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

QuickSort Optimizations

Many, many optimizations and variants:

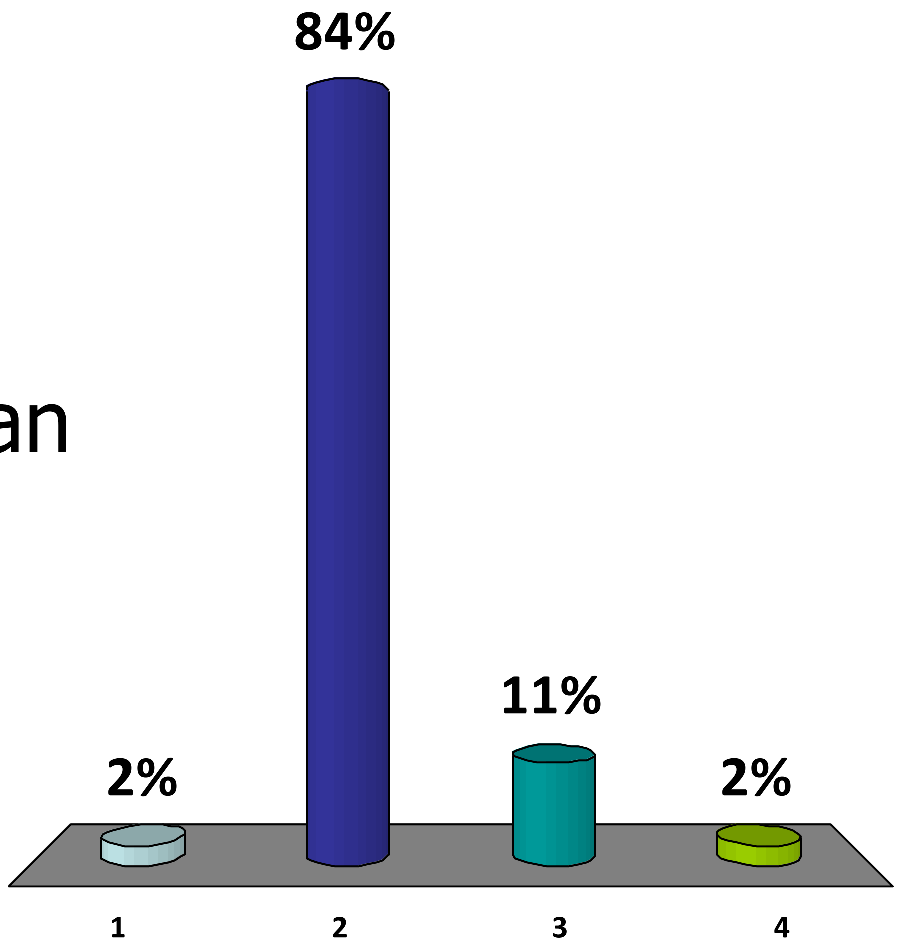
1. To save space, recurse into smaller half first.
   – Only need O(log n) extra space.

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!

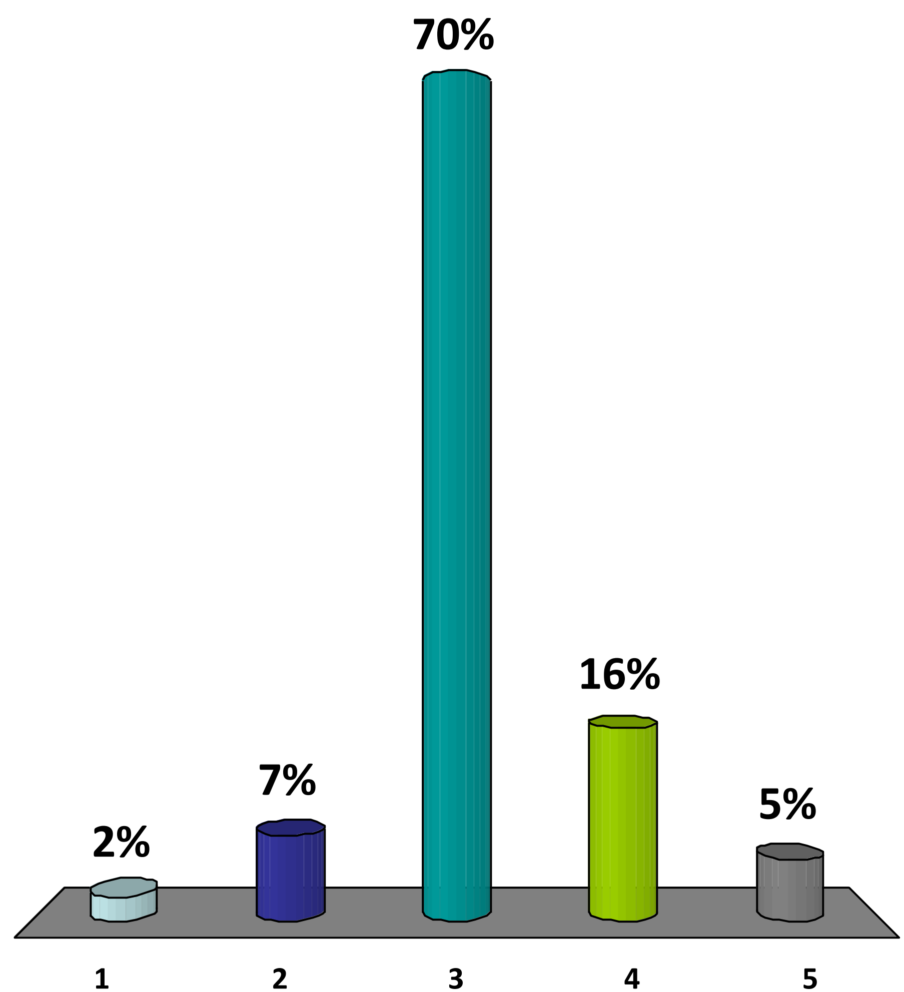
Which of the following is most important for QuickSort to be efficient?

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



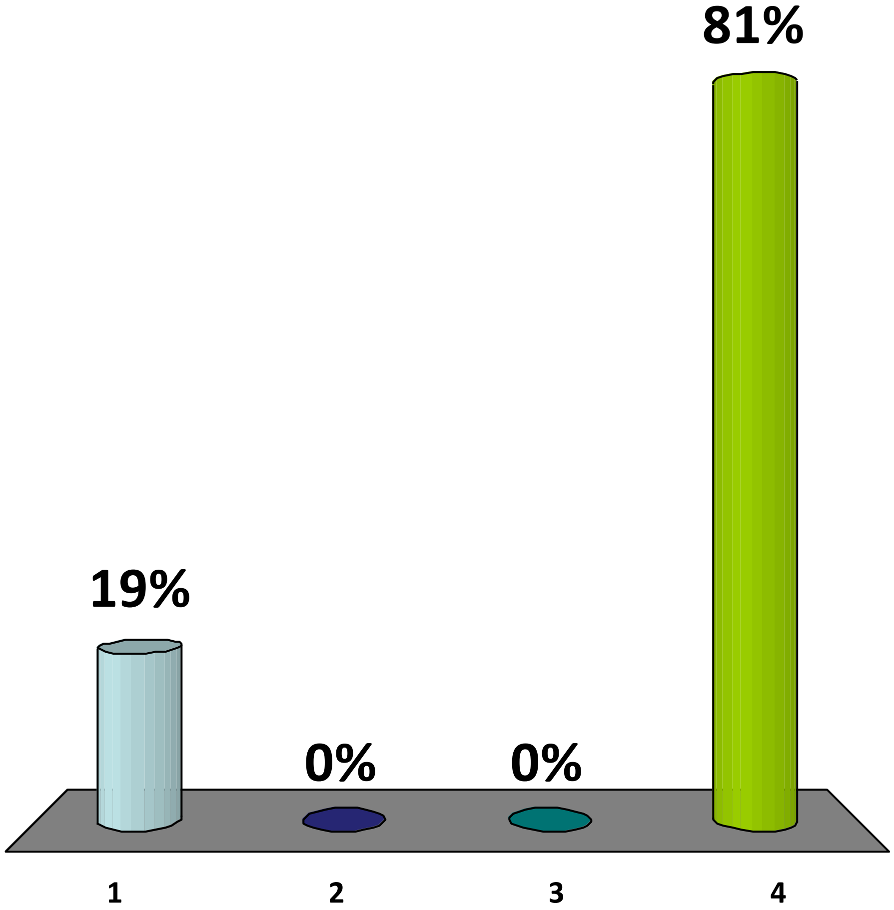
Which of the following is **not** true of the partition algorithm?

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.



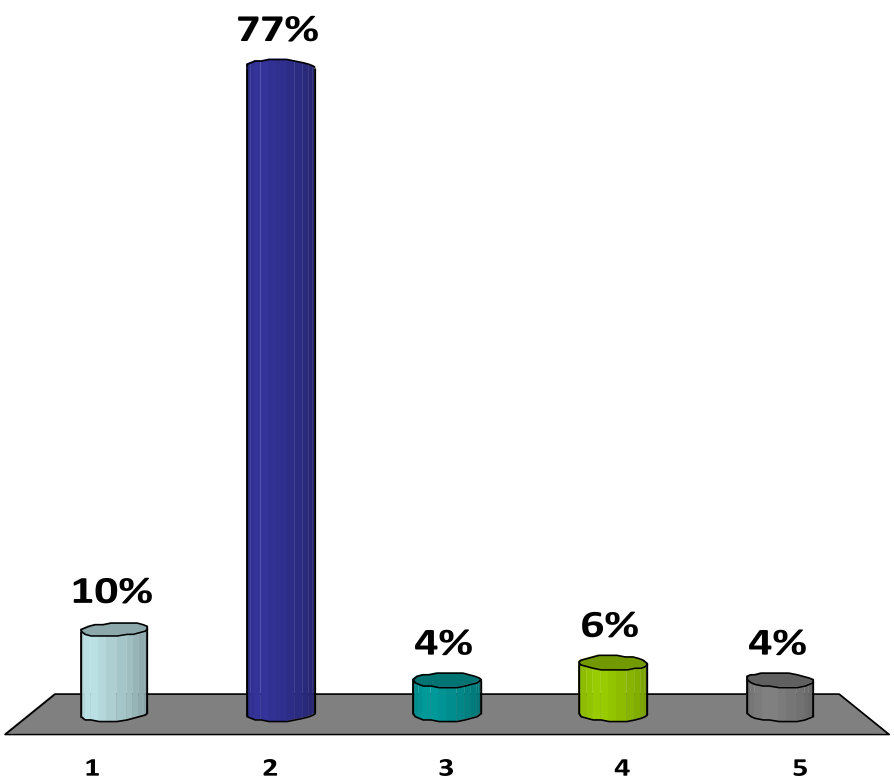
If the pivot is chosen to be A[1], which of the following has the worst running time?

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]



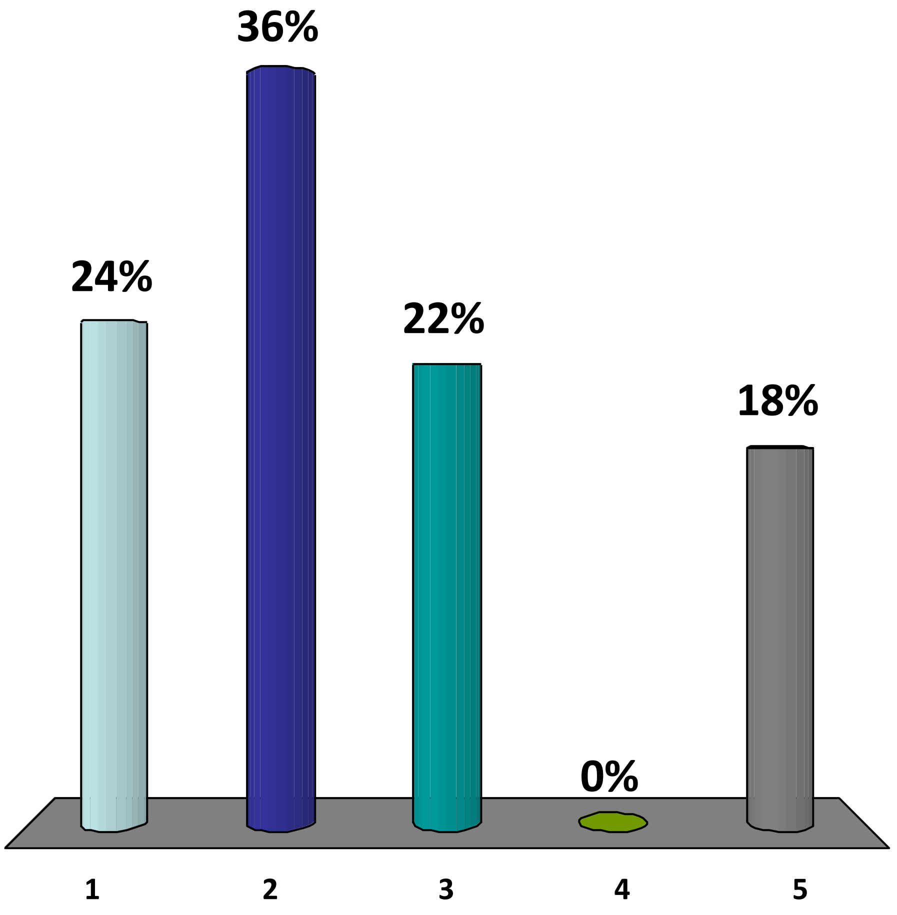
If the pivot is chosen at random, what is the expected number of times to partition before choosing a pivot that partitions the array into a: 1/4 : 3/4 split

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



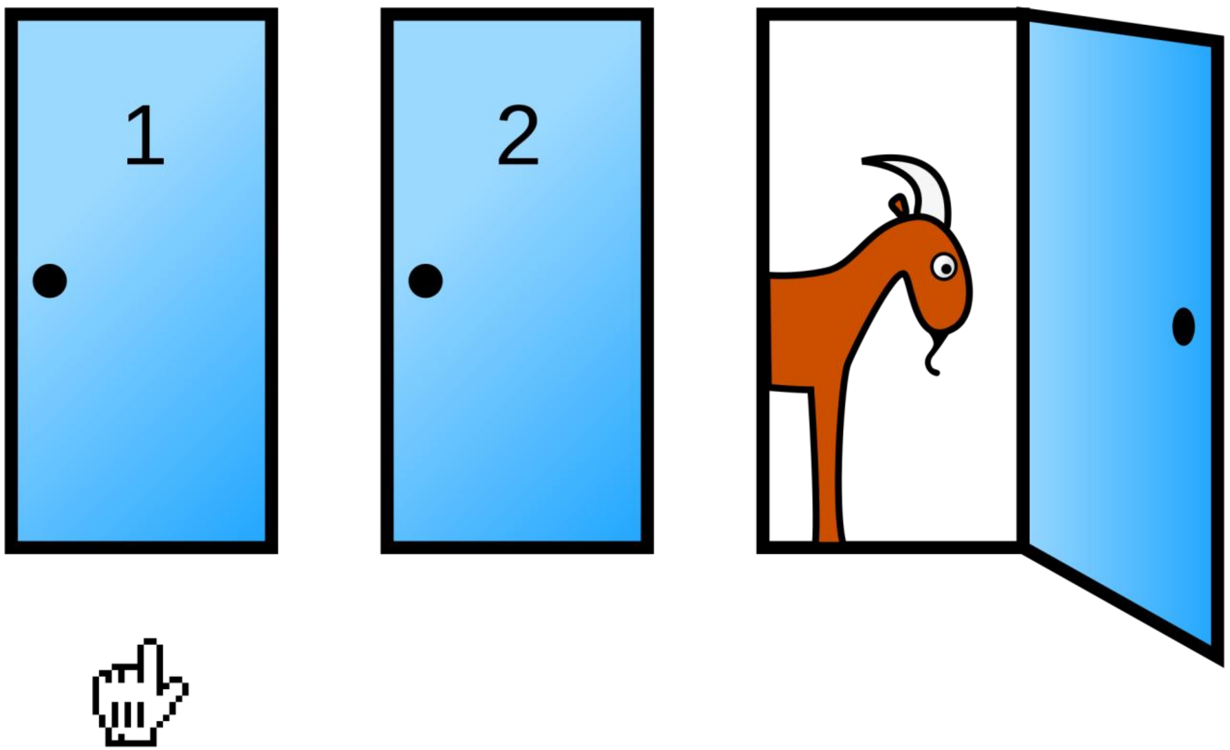
Which of the following helps to explain why QuickSort is faster than other sorting algorithms:

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.



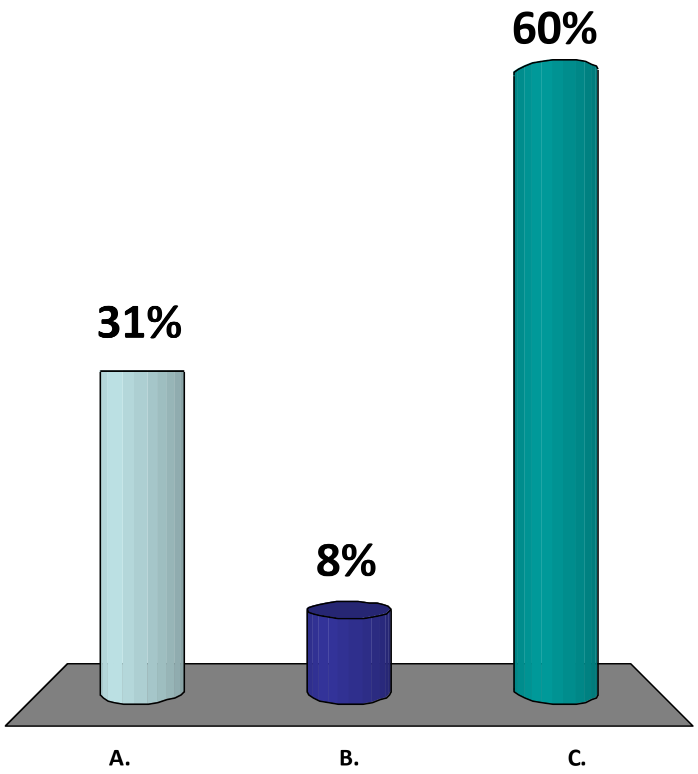
Probability Theory

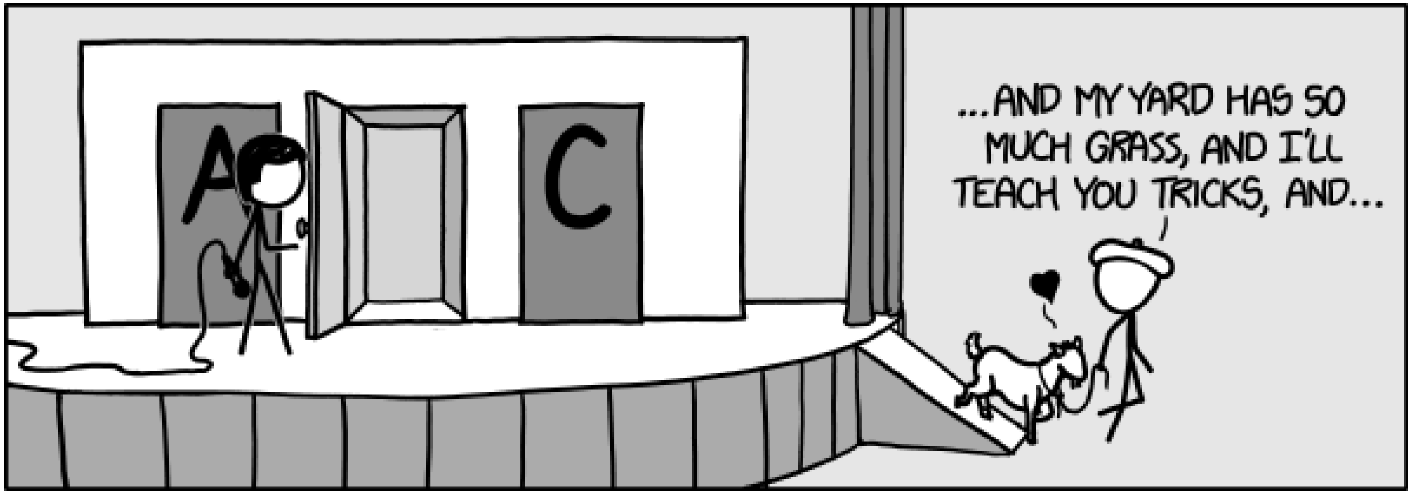
Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?



Will you switch?

A. Yes
B. No
C. Let's just bring the goat home





Probability Theory

Probability Theory

Flipping a coin:

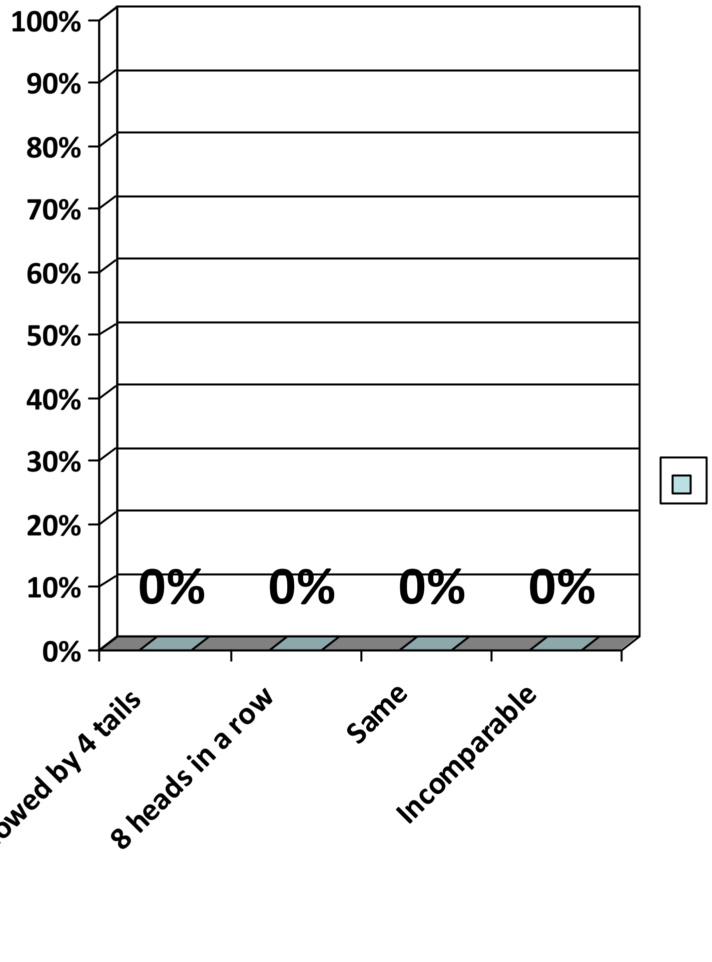
- Pr(heads) = ½
- Pr(tails) = ½

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. Same
d. Incomparable



Response Counter

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**:

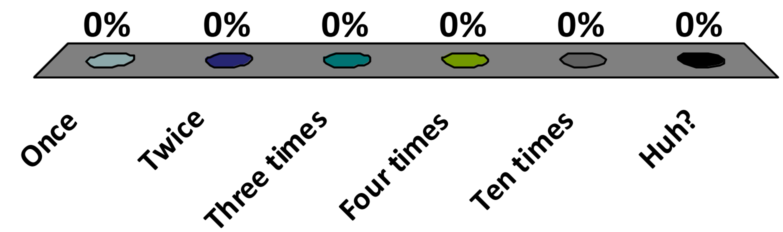
- Pr(**A**), Pr(**B**)
- **A** and **B** are independent (e.g., unrelated random coin flips)

Pairwise:

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

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

A. Once
B. Twice
C. Three times
D. Four times
E. Ten times
F. Huh?



Response Counter

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

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

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

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

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

Probability Theory

Set of outcomes for X = (e_1, e_2, e_3, …, 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
- B = # heads in 2 coin flips
- $A + B$ = # heads in 4 coin flips

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

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

E[X]= expected number of flips to get one head

Example: X = 7

T T T T T T H

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

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

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

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

Pr(T H)*2 +

Pr(T T H)*3 +

Pr(T T T H)*4 +

…

Probability Theory

Flipping an (unfair) coin:

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

How many flips to get at least one head?

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

Probability Theory

Flipping an (unfair) coin:

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

How many flips to get at least one head?

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

How many more flips to get a head?

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

Probability Theory

Flipping an (unfair) coin:

- Pr(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:

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

How many flips to get at least one head?

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

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

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

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

Probability Theory

Flipping an (unfair) coin:

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

How many flips to get at least one head?

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

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

Summary

QuickSort:

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

Next time: applications of sorting techniques…