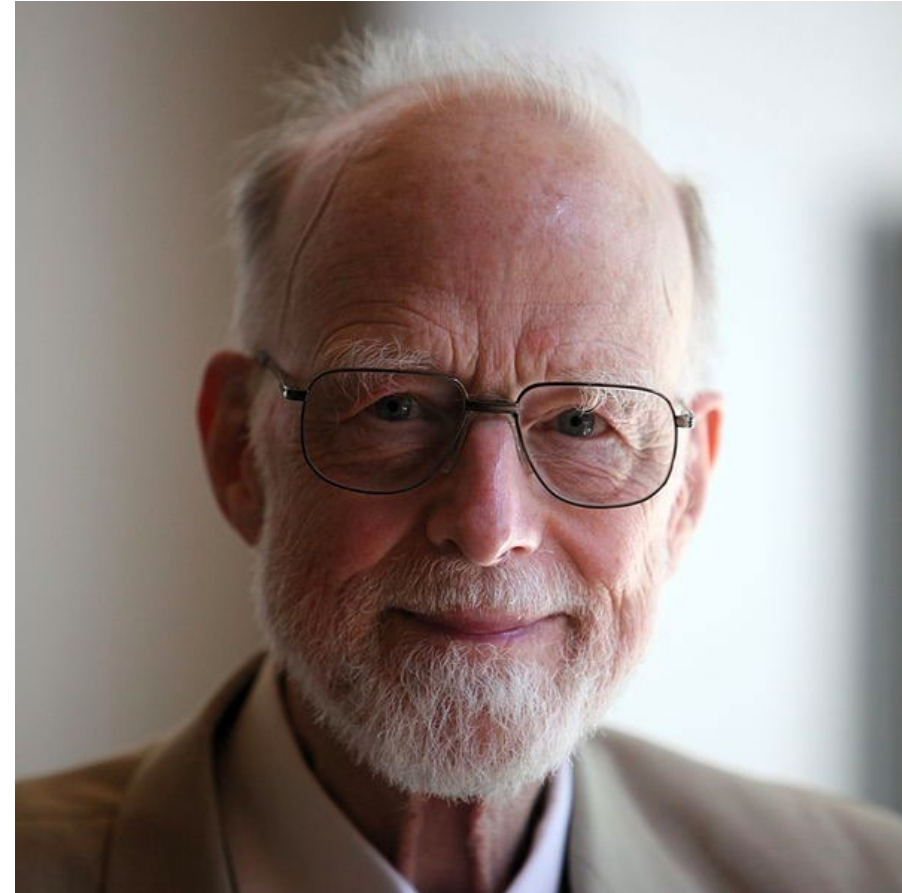


QuickSort

QuickSort History

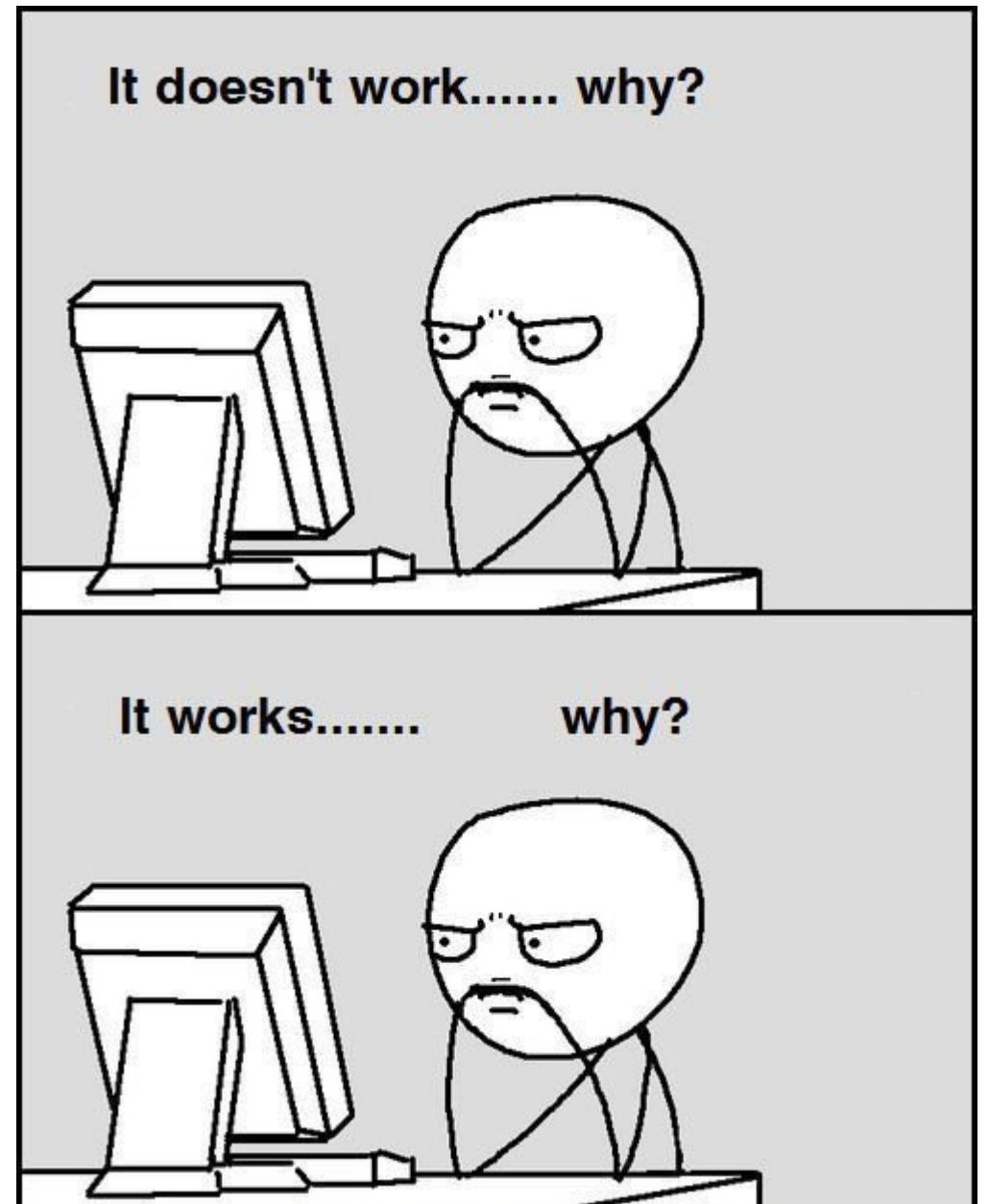
- Invented by C.A.R. Hoare in 1960
 - Turing Award: 1980
- Visiting student at Moscow State University
- Used for machine translation (English/Russian)



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.”

My 40 Years of Experience



QuickSort

- History:
 - Invented by C.A.R. Hoare in 1960
 - Used for machine translation (English/Russian)
- In practice:
 - Very fast
 - Many optimizations
 - In-place (i.e., no extra space needed)
 - Good caching performance
 - Good parallelization

QuickSort Today

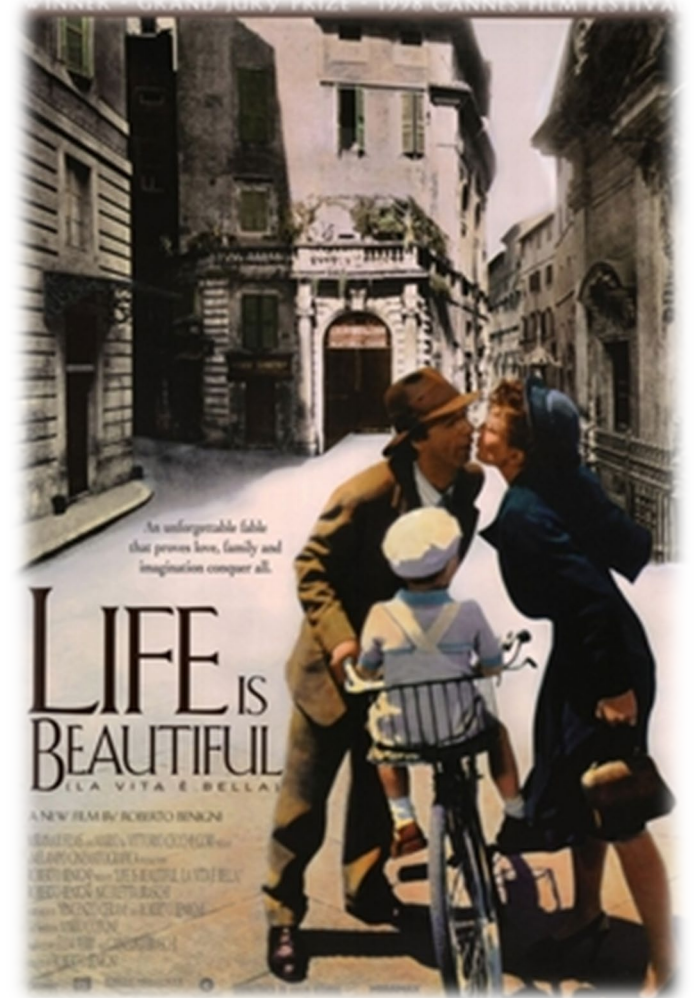
- 1960: Invented by Hoare
- 1979: Adopted everywhere (e.g., Unix qsort)
- 1993: Bentley & McIlroy improvements
- 2009: Vladimir Yaroslavskiy
 - Dual-pivot Quicksort !!!
 - Now standard in Java 7
 - 10% faster!
- 2012: Sebastian Wild and Markus E. Nebel
 - “Average Case Analysis of Java 7’s Dual Pivot...”
 - Best paper award at ESA

QuickSort

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

QuickSort First Assumption

- For starter, let's assume the world is beautiful....
- For a lot of algorithms, it's better to be explained in a simplified problem first
- But it doesn't mean it cannot work on the "original" problem



Let's Assume that

- Every element in the array is unique



Recall: MergeSort

```
MergeSort (A, n)
```

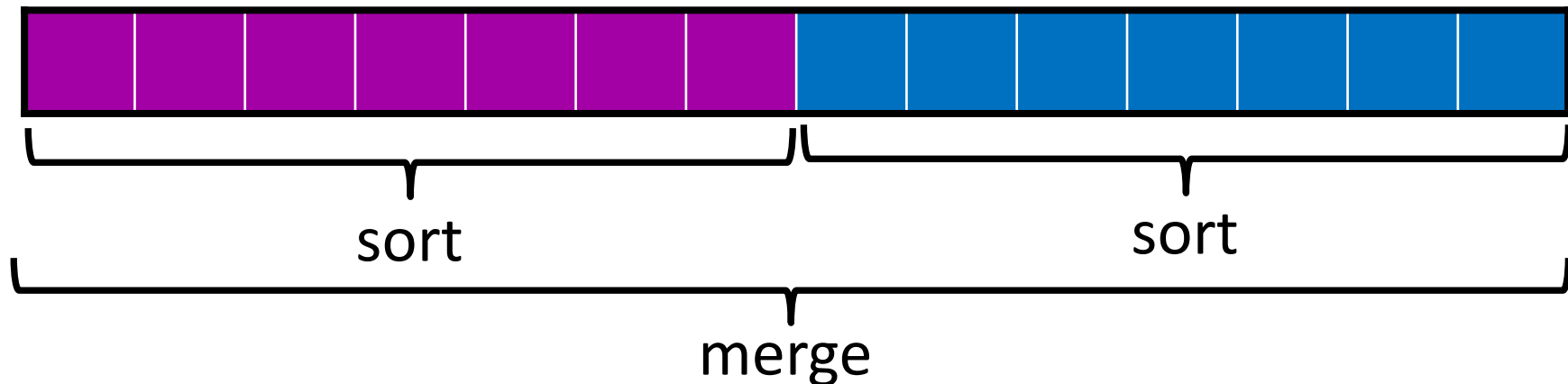
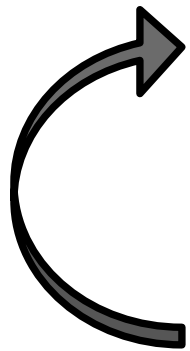
```
  if (n=1) then return;
```

```
  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)
```

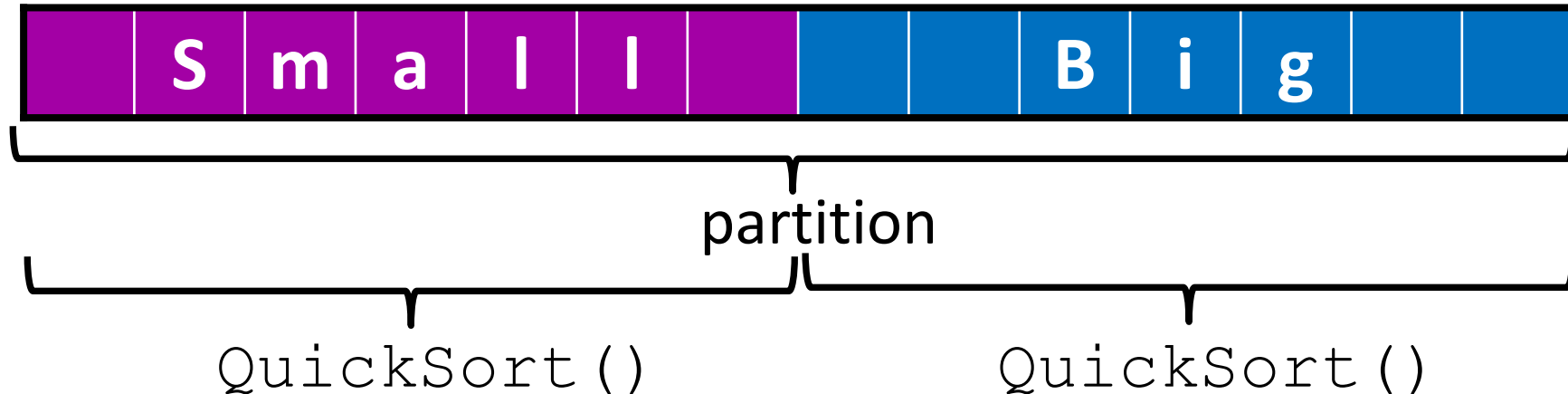
```
    if (n==1) then return;
```

```
    else
```

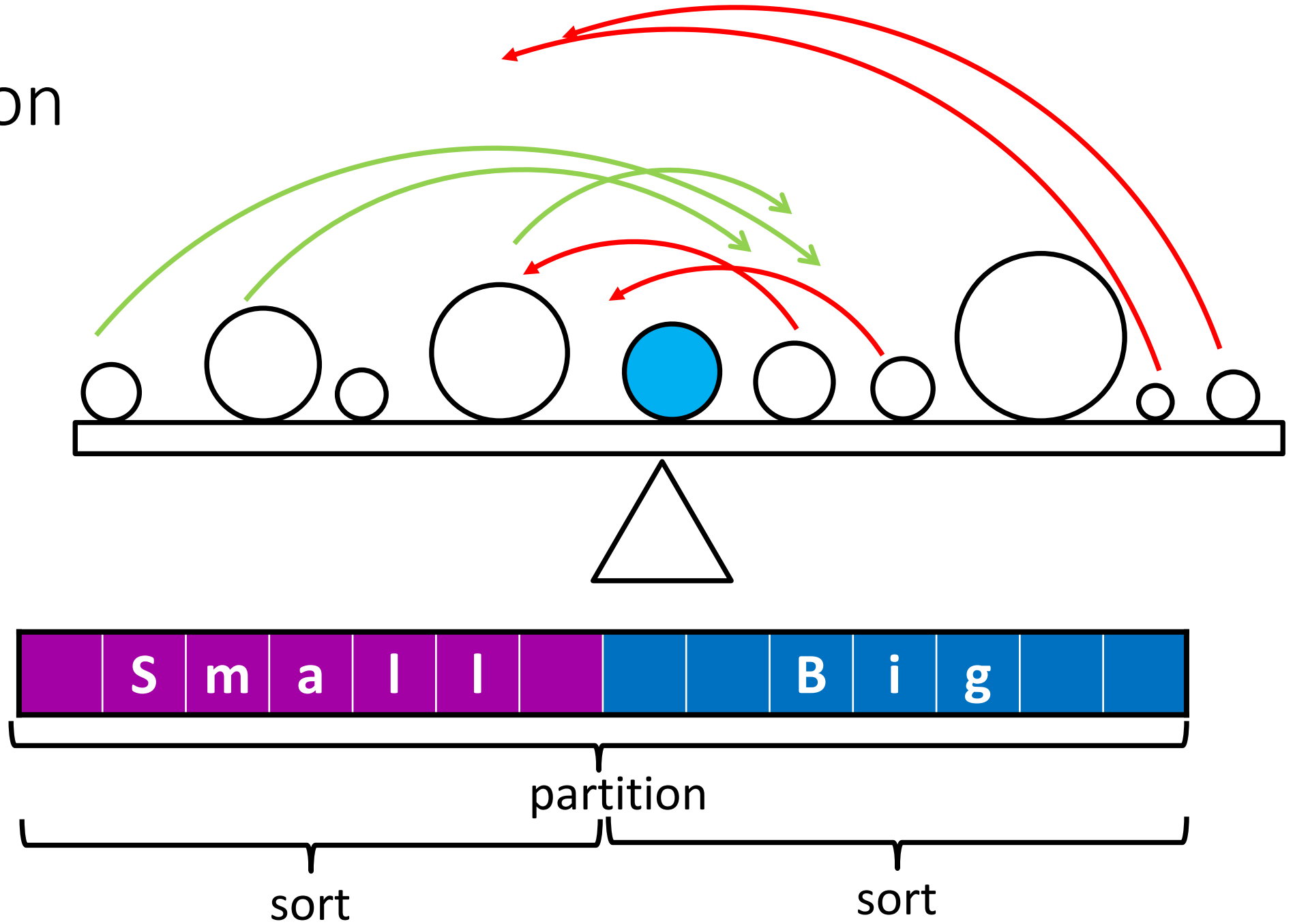
```
        p = partition(A[1..n], n)
```

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

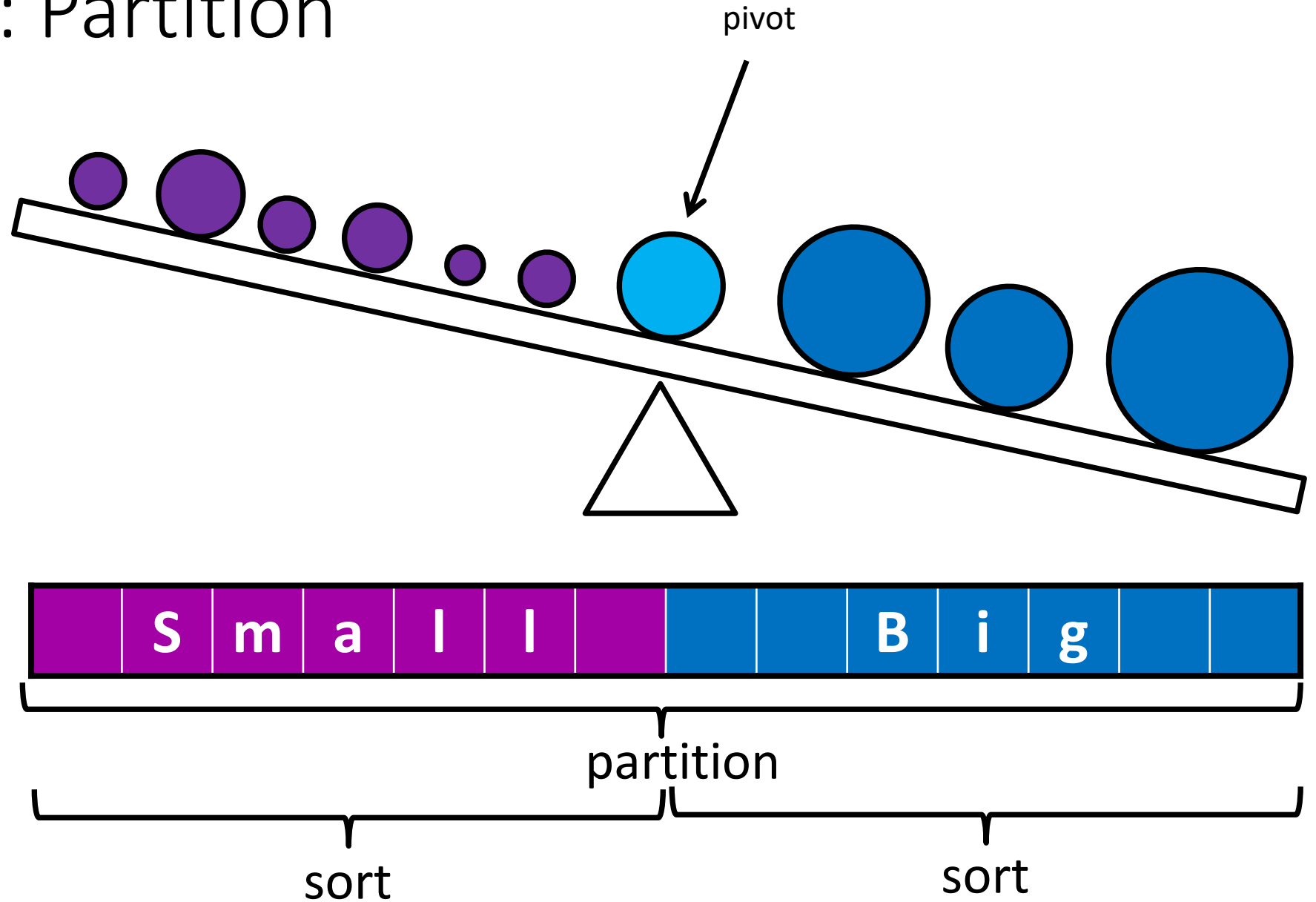
```
        y = QuickSort(A[p+1..n], n-p)
```



Partition



QuickSort: Partition



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.



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



Key: efficient *partition* sub-routine

Partitioning

- 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.



* a lot of rooms to discuss

Let's Assume We Got the Magic to Partition

- Given

6	3	9	8	4	1
---	---	---	---	---	---

- Pick a pivot, say the first item "6"

3	4	1	6	9	8
---	---	---	---	---	---

- "6" is "sorted"
- QuickSort the left and the right

Let's Assume We Got the Magic to Partition

- “6” is sorted, QuickSort the left ~~and the right~~



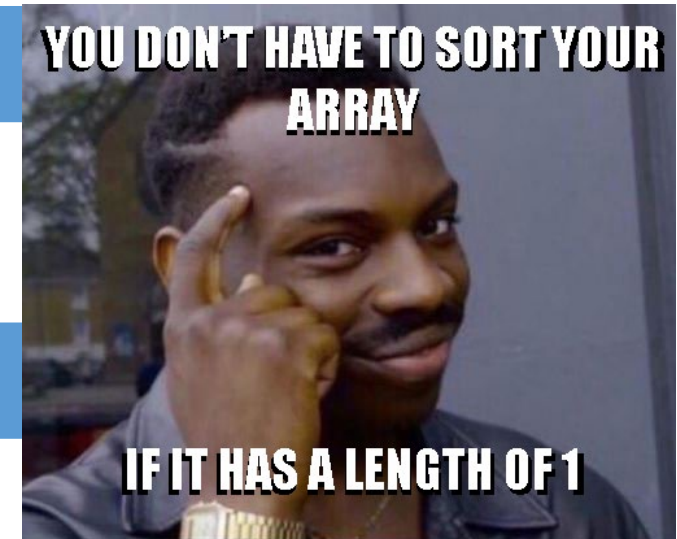
- Pick a pivot, say 3



- After partition on the left array, “3” is sorted



- And the two “arrays” with one element are sorted



Let's Assume We Got the Magic to Partition

- QuickSort the right



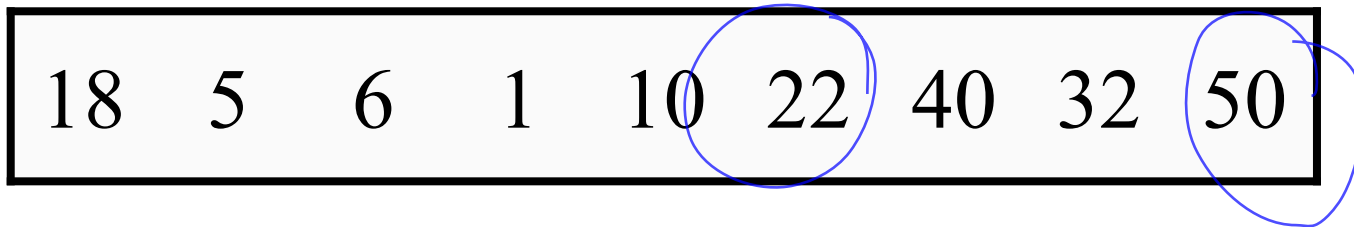
- Pick a pivot, say 9



- After partition on the left array, "9" is sorted
- And the "array" with "8" is also sorted
- DONE!

Which one is the pivot?

- If the following array is partitioned before further recursion, which one is the pivot?



Partitioning

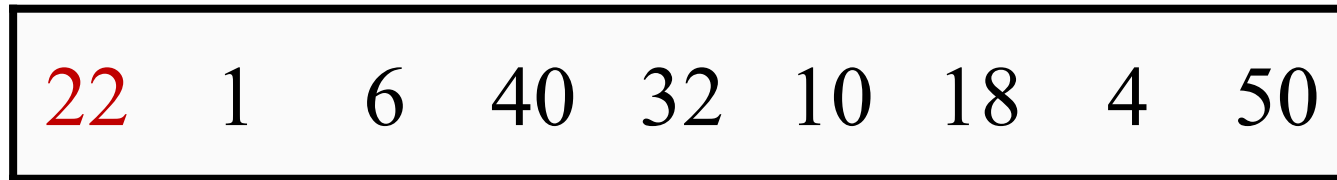
- Three steps:

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.



- What is the time complexity for partitioning once?

Partitioning



low
 < 22



Move until it's bigger
than the pivot



high
 > 22



Move until it's less
than the pivot

Partitioning

22	1	6	40	32	10	18	4	50
----	---	---	----	----	----	----	---	----

< 22



low



high

> 22

Partitioning

22	1	6	40	32	10	18	4	50
----	---	---	----	----	----	----	---	----

< 22



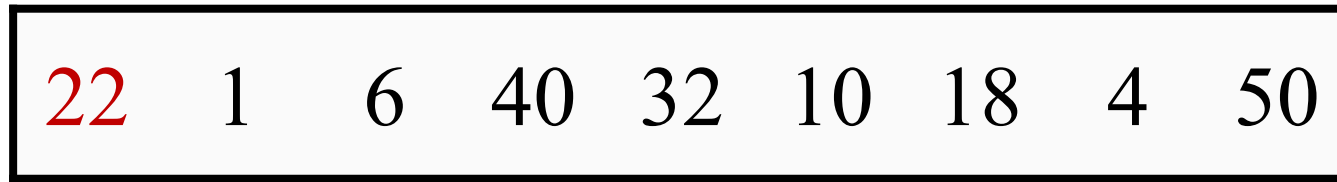
low



high

> 22

Partitioning



< 22



low



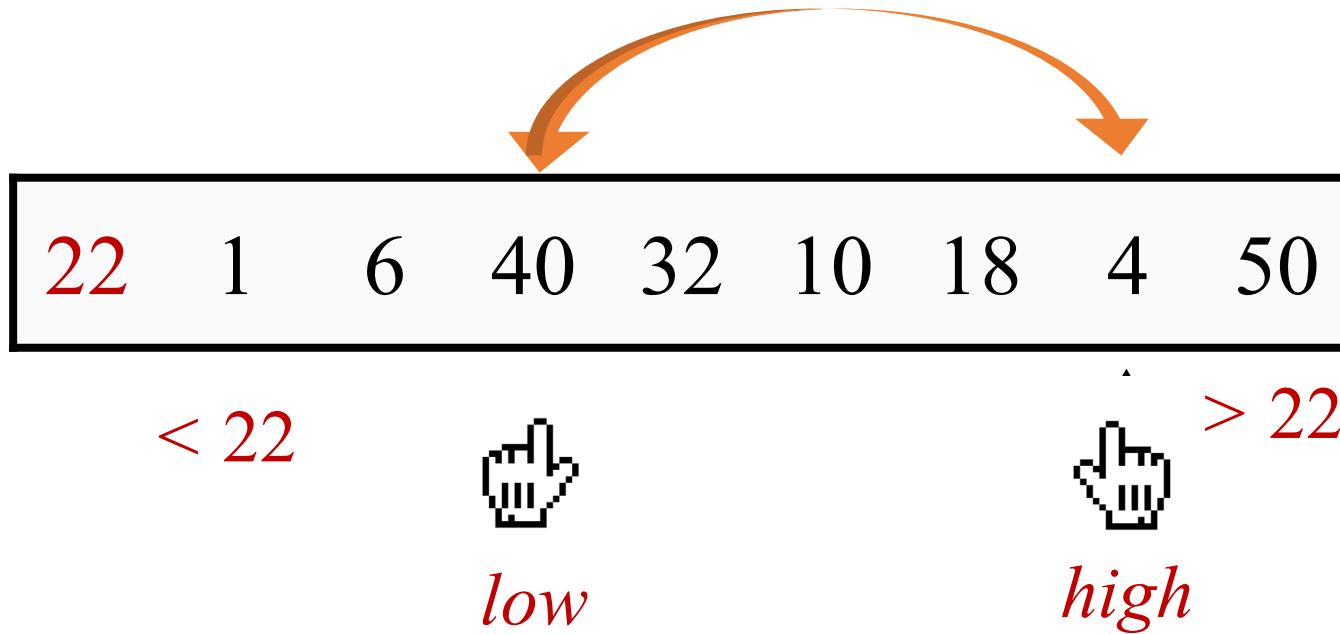
> 22



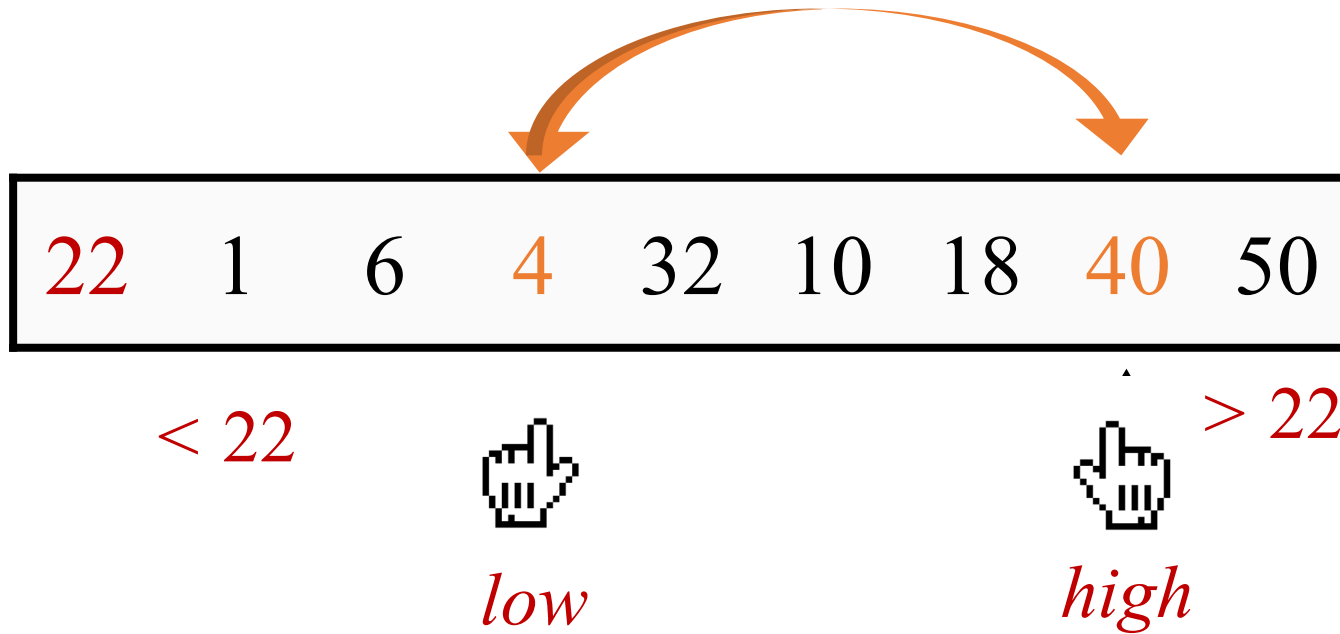
high



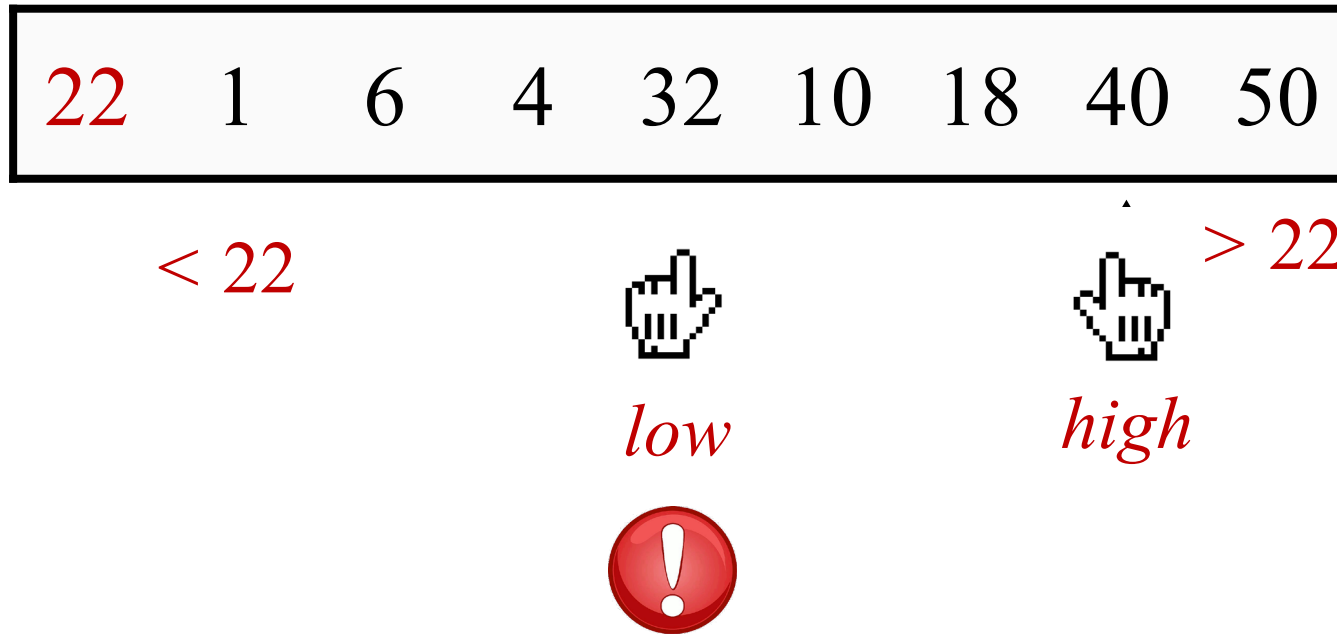
Partitioning



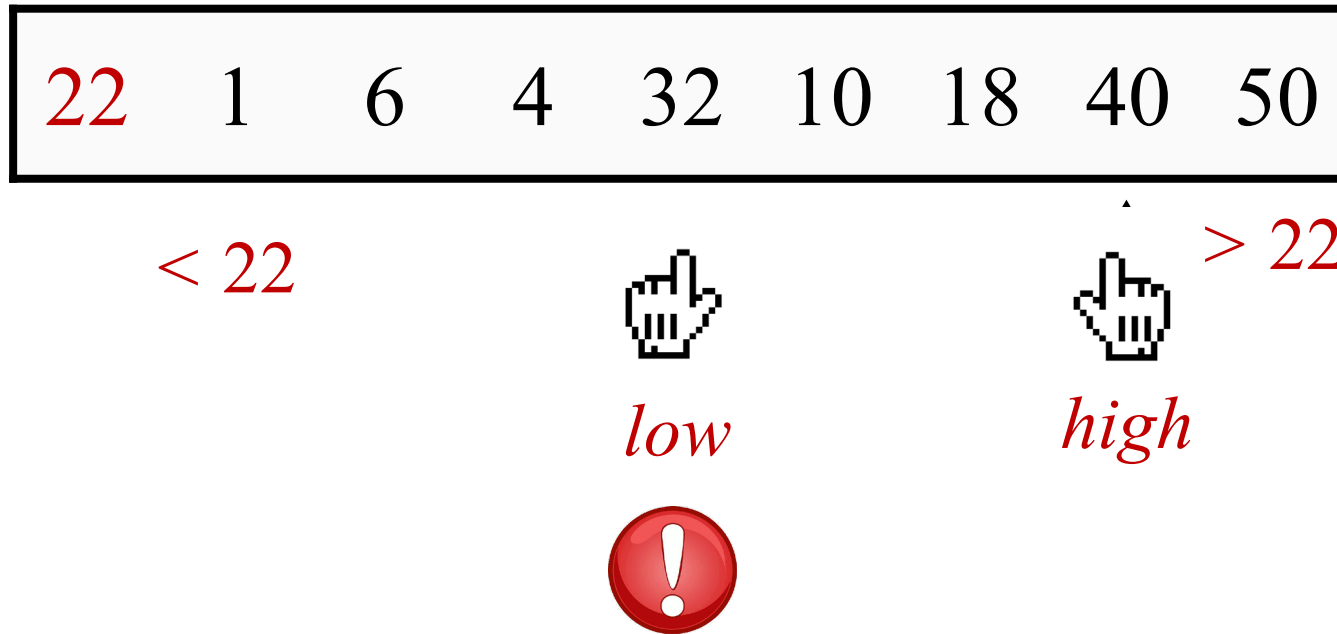
Partitioning



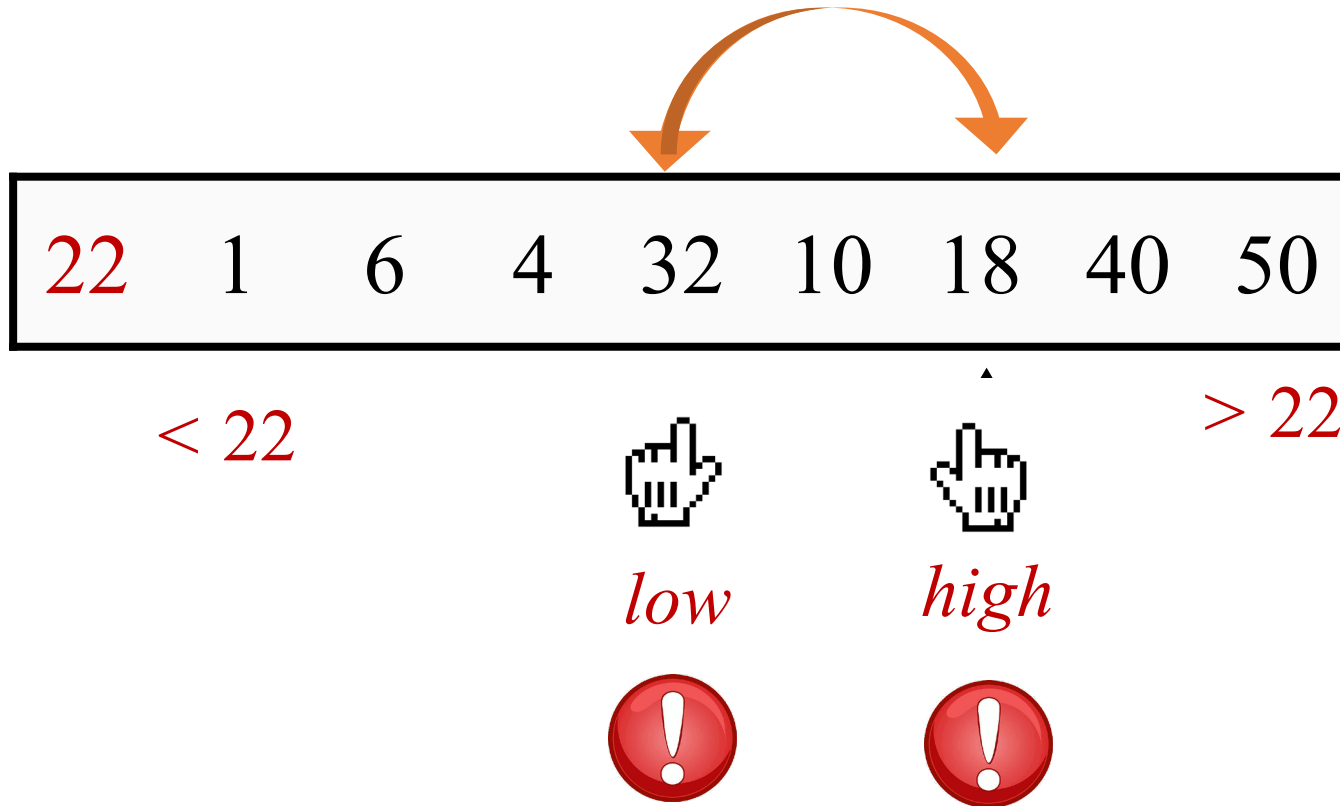
Partitioning



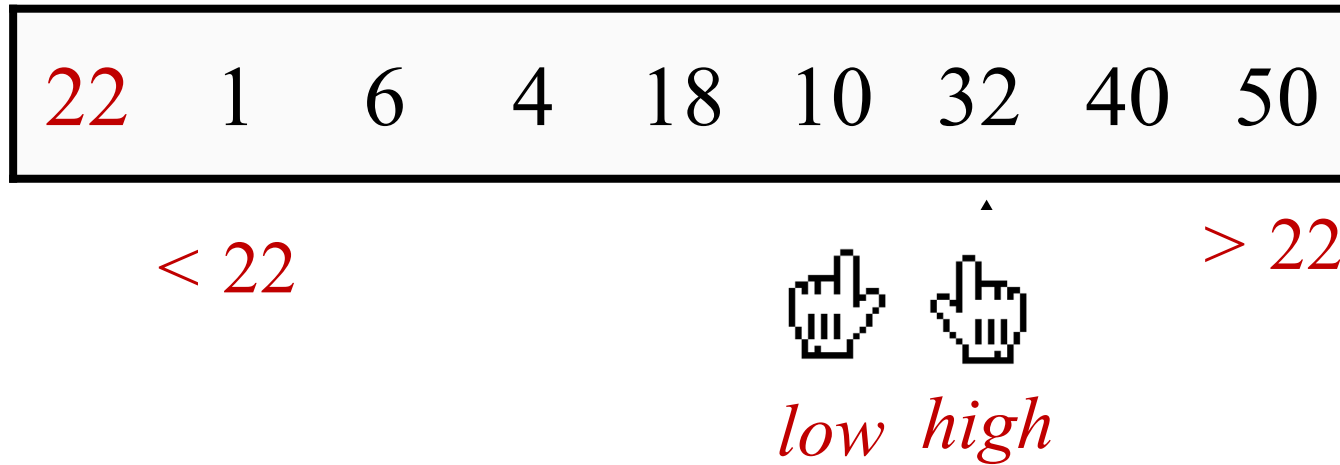
Partitioning



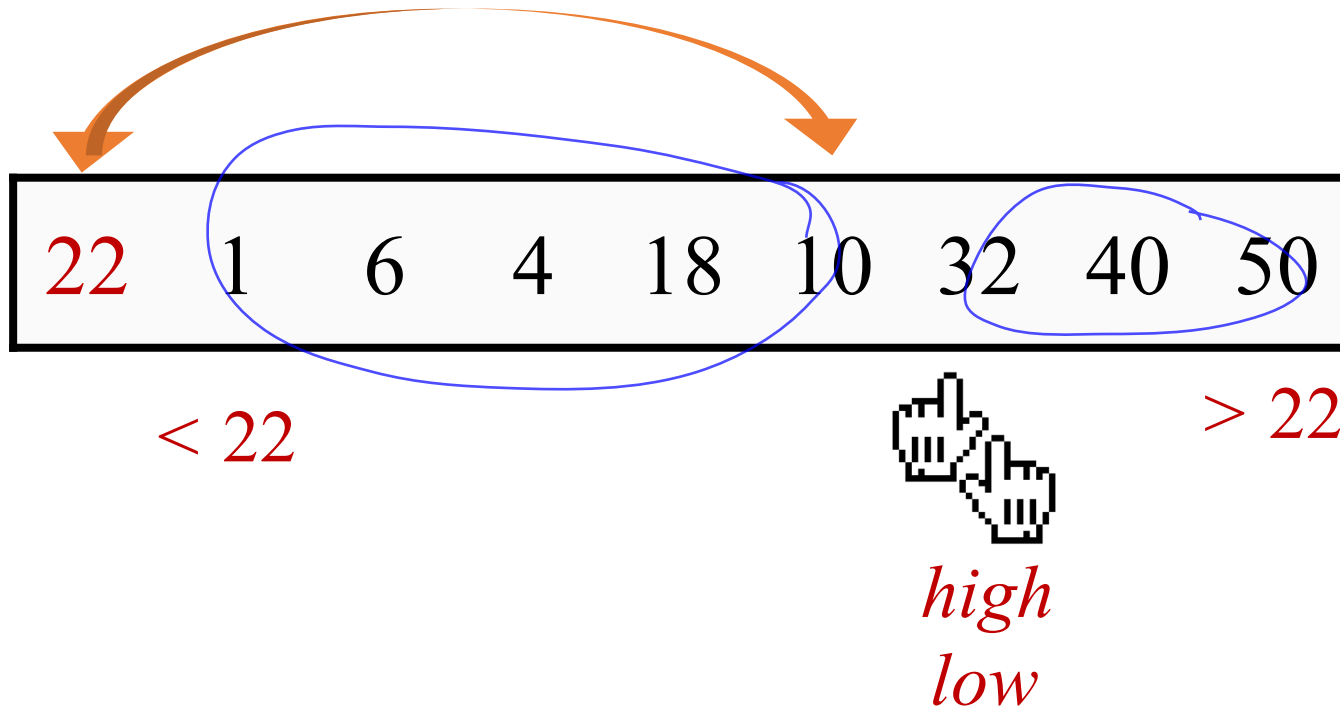
Partitioning



Partitioning



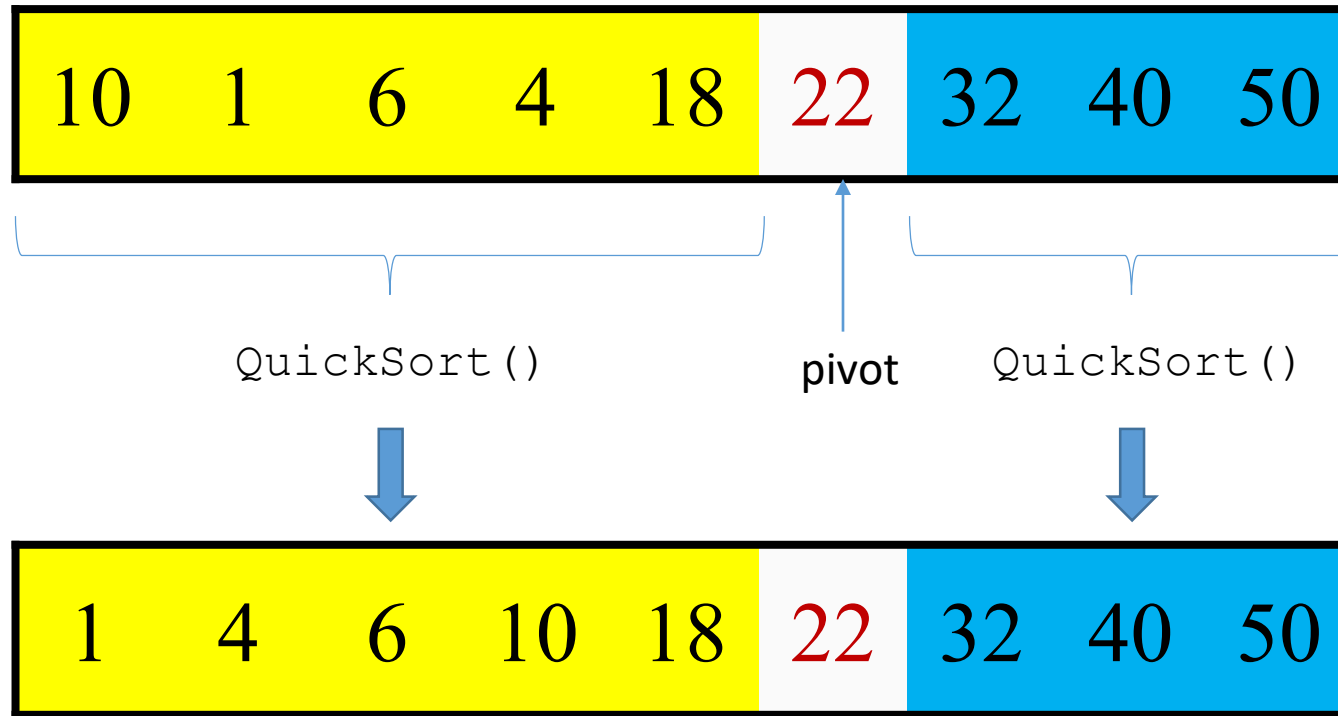
Partitioning



`high == low`



Partitioning Done




```
partition(A[1..n], n)
```

```
    pivot = 1
```

```
    low = 2;
```

// start after pivot in $A[1]$

```
    high = n+1;
```

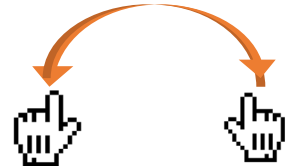
// Define: $A[n+1] = \infty$

```
    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;
```



QuickSort

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

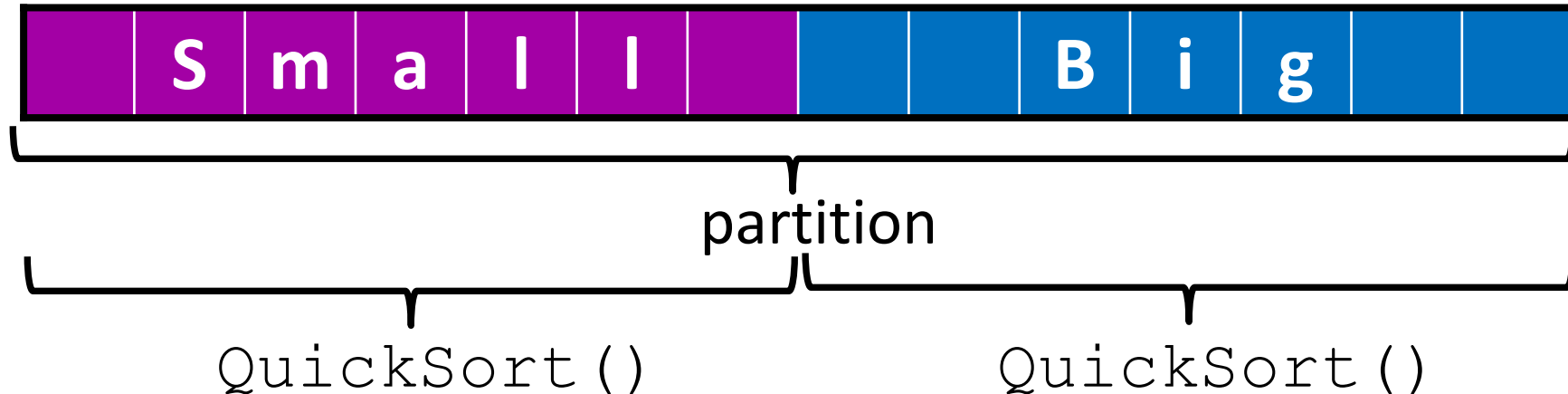
```
    if (n==1) then return;
```

```
    else
```

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

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

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



What if there are duplicates?



Where will it go wrong?

```
partition(A[1..n], n)
    pivot = 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 will get “stuck”

22	1	6	22	32	10	18	22	50
----	---	---	----	----	----	----	----	----

< 22



low



> 22



high



Where will it go wrong?

```
partition(A[1..n], n)
```

```
    pivot = 1
```

```
    low = 2;
```

// start after pivot in $A[1]$

```
    high = n+1;
```

// Define: $A[n+1] = \infty$

```
    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;
```

Nothing
changed



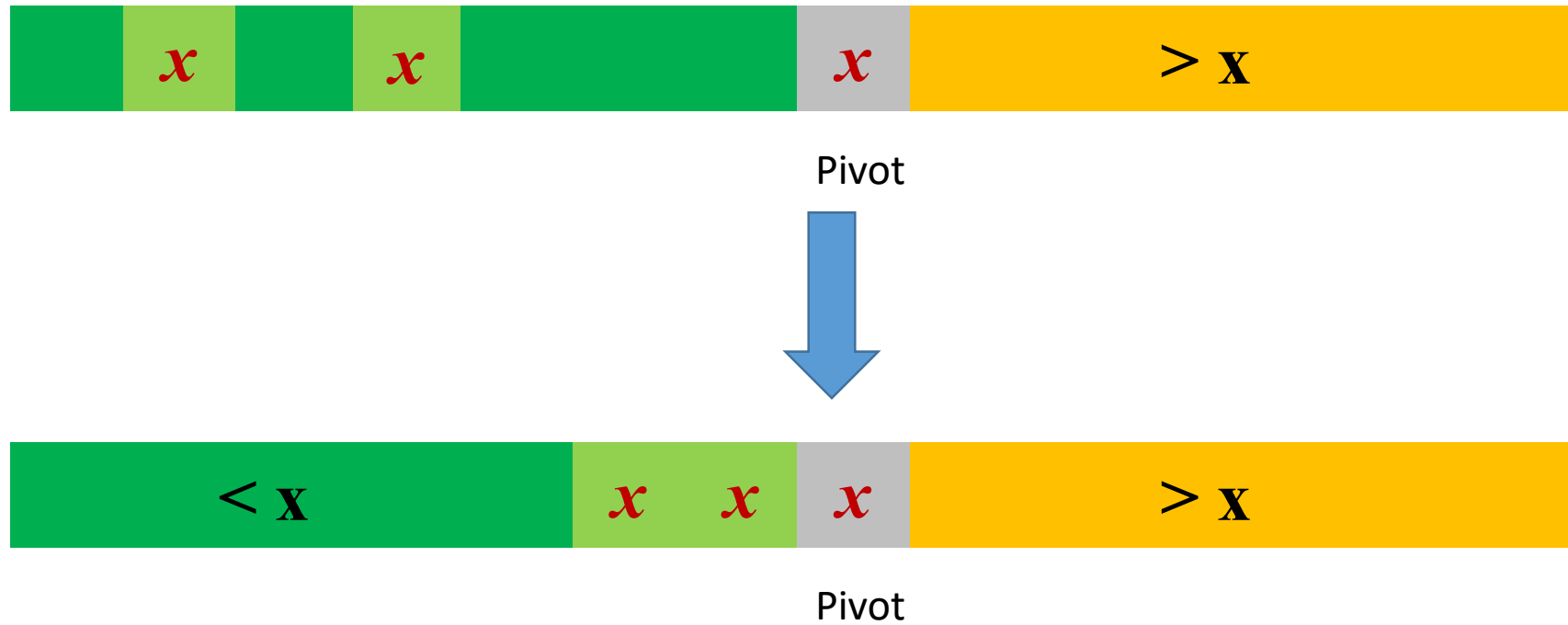
Let it go, let it go...

```
partition(A[1..n], n)
    pivot = 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;
```



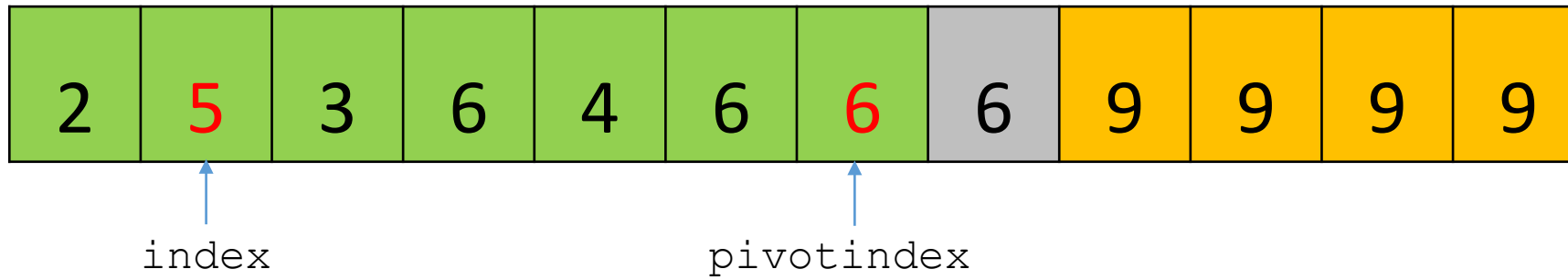
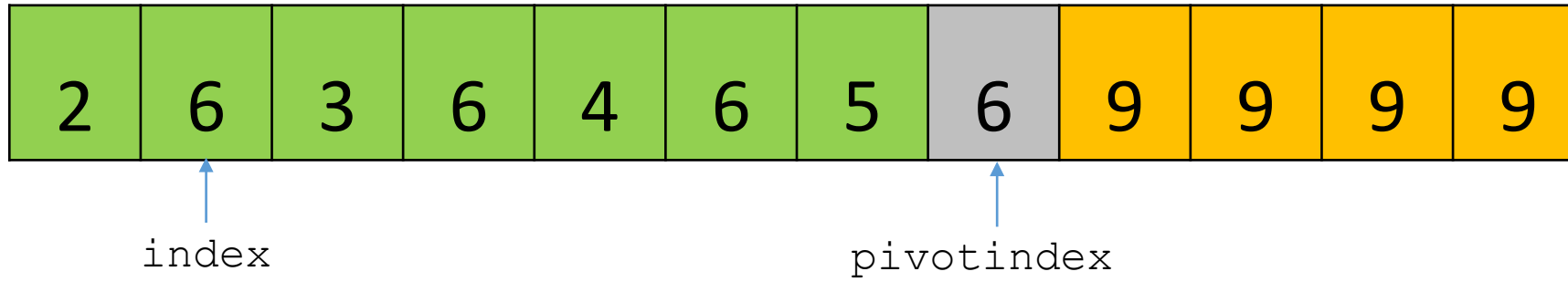
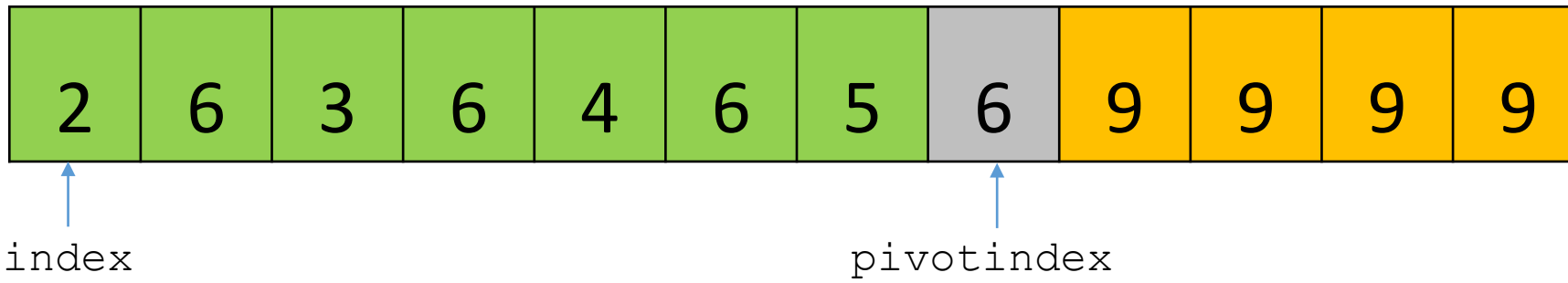
Pivot

Pack Duplicates

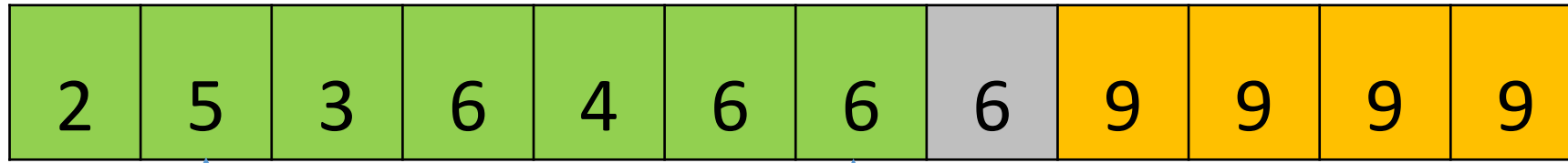


Pack Duplicates

```
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 ++;
    }
```

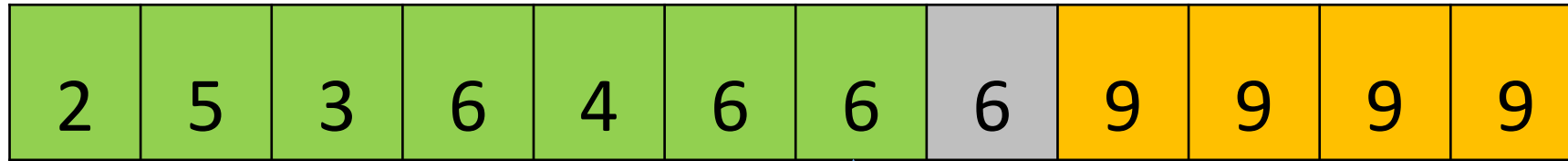


```
pivotIndex--;  
swap(A[index], A[pivotIndex]);
```



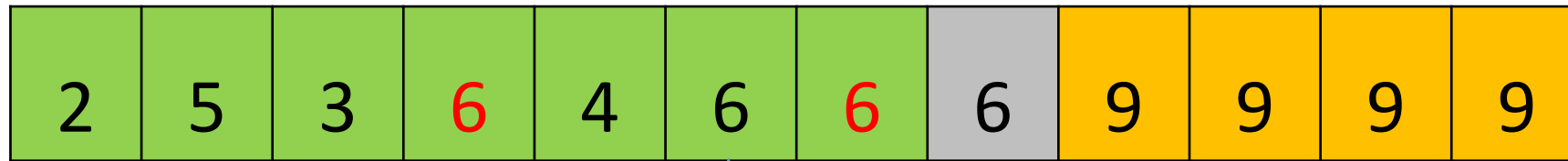
index

pivotindex



→ → index

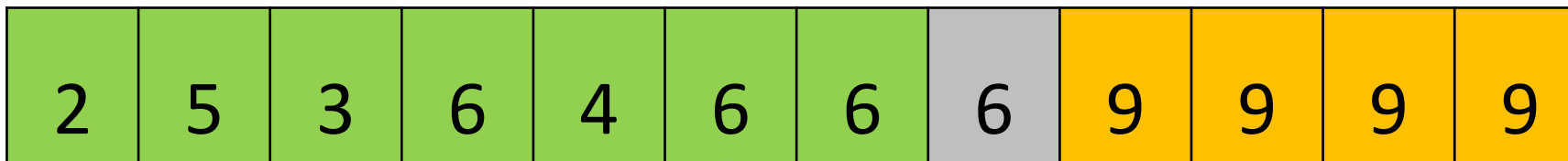
pivotindex



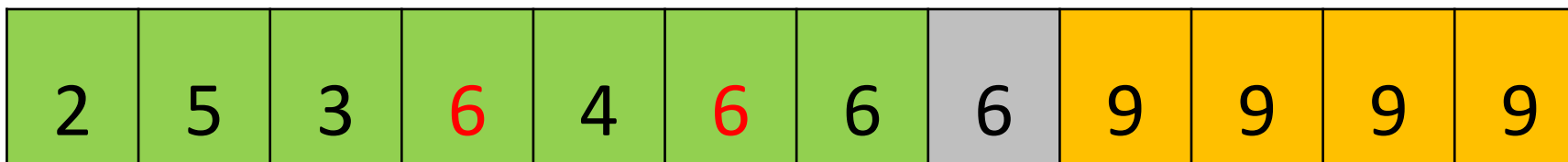
index

pivotindex

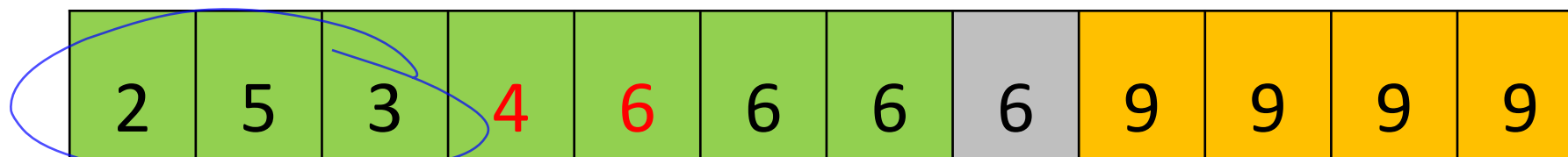
```
pivotIndex--;  
swap(A[index], A[pivotIndex]);
```



index pivotIndex



index
pivotIndex



index
pivotIndex

```
pivotIndex--;  
swap(A[index], A[pivotIndex]);
```

Pack Duplicates

```
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 ++;
    }
```

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

QuickSort

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

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

```
    else
```

```
        p = ThreeWayPartition (A[1..n], n)
```

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

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



Is QuickSort Stable?



→ → index

↑ pivotindex



↑ index ↑ pivotindex

```
pivotIndex--;  
swap(A[index], A[pivotIndex]);
```

Time Complexity?

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

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

```
    else
```

```
        p = ThreeWayPartition (A[1..n], n) ←  $O(n)$ 
```

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

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

- Lucky case
 - If $p = n/2$ all the time
- $T(n) = cn + 2 T(n/2)$
- Same as MergeSort!



The pivot we picked is always the median of the array

Time Complexity?

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

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

```
    else
```

```
        p = ThreeWayPartition (A[1..n], n) ←  $O(n)$ 
```

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

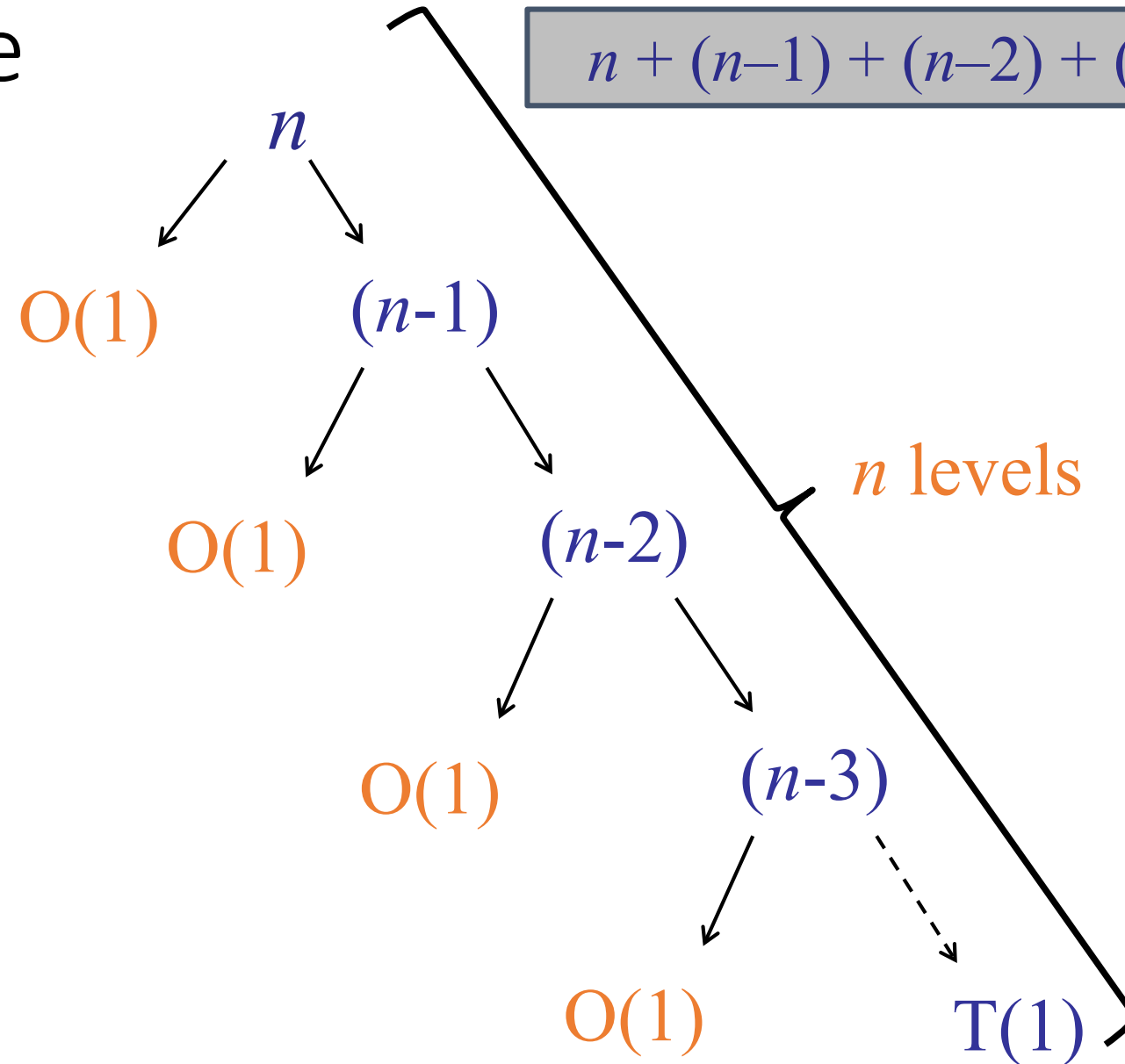
```
        y = QuickSort (A[p+1..n], n-p) ←  $T(n-p)$ 
```

- But what if $p = 1$ all the time?

- $$\begin{aligned} T(n) &= cn + T(n-1) + T(1) \\ &= cn + c(n-1) + T(n-2) + T(1) + T(1) \\ &= cn + c(n-1) + c(n-2) + T(n-2) + T(1) + T(1) + T(1) \\ &= c(n + (n-1) + (n-2) + (n-3) + \dots + 1) + T(n) = O(n^2) \end{aligned}$$



Worst-case



Time Complexity

- Lucky case
 - If $p = n/2$ all the time
 - $T(n) = cn + 2 T(n/2) = O(n \log n)$
- Worst case
 - if $p = 1$ all the time
 - $T(n) = O(n^2)$

Can we always choose
the median as pivot?!

Time Complexity

- Lucky case
 - If $p = n/2$ all the time
 - $T(n) = cn + 2 T(n/2) = O(n \log n)$
- Worst case
 - if $p = 1$ all the time
 - $T(n) = O(n^2)$
- Next: How about choose something in the middle?

