# CS2040S
# Data Structures and Algorithms

## Welcome!

---

**Algorithm 1** ICan'tBelieveItCanSort($A[1..n]$)

---

**for** $i = 1$ **to** $n$ **do**
  **for** $j = 1$ **to** $n$ **do**
    **if** $A[i] < A[j]$ **then**
      swap $A[i]$ and $A[j]$

---

Does this sorting algorithm work correct? If not, can you fix it…

# Last Time: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

# Today: more sorting!

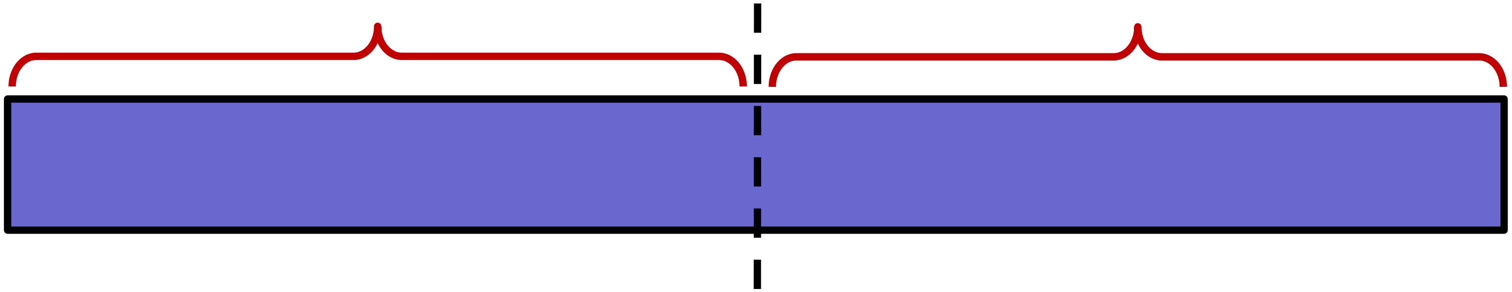
QuickSort:

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

# Recursive…

Step 1:
Divide array into two pieces.

```
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);
```



# Recursive…

Step 2:
Recursively sort the two halves.

MergeSort(A, n)

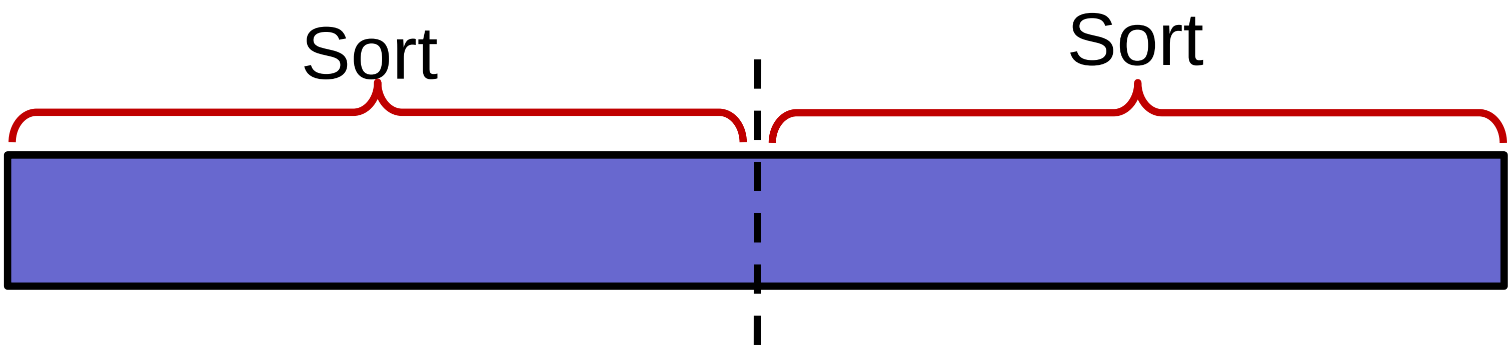
**if** (n=1) **then return;**

**else:**

X ←MergeSort**(**A[1..n/2], n/2**);**

Y ←MergeSort**(**A[n/2+1, n], n/2**);**

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



# Recursive...

Step 3:
Merge the two halves into one sorted array.

MergeSort(A, n)

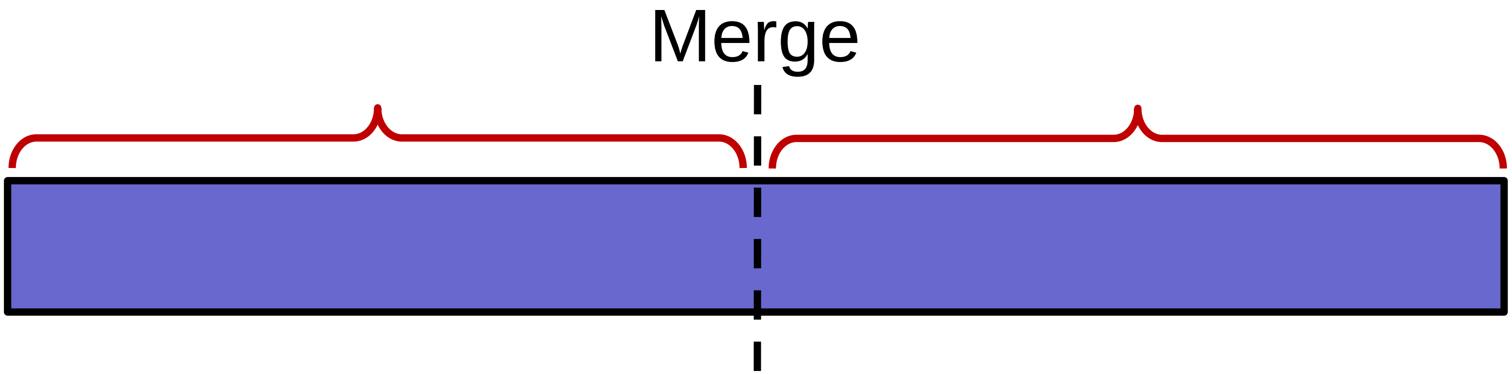
**if** (n=1) **then return;**

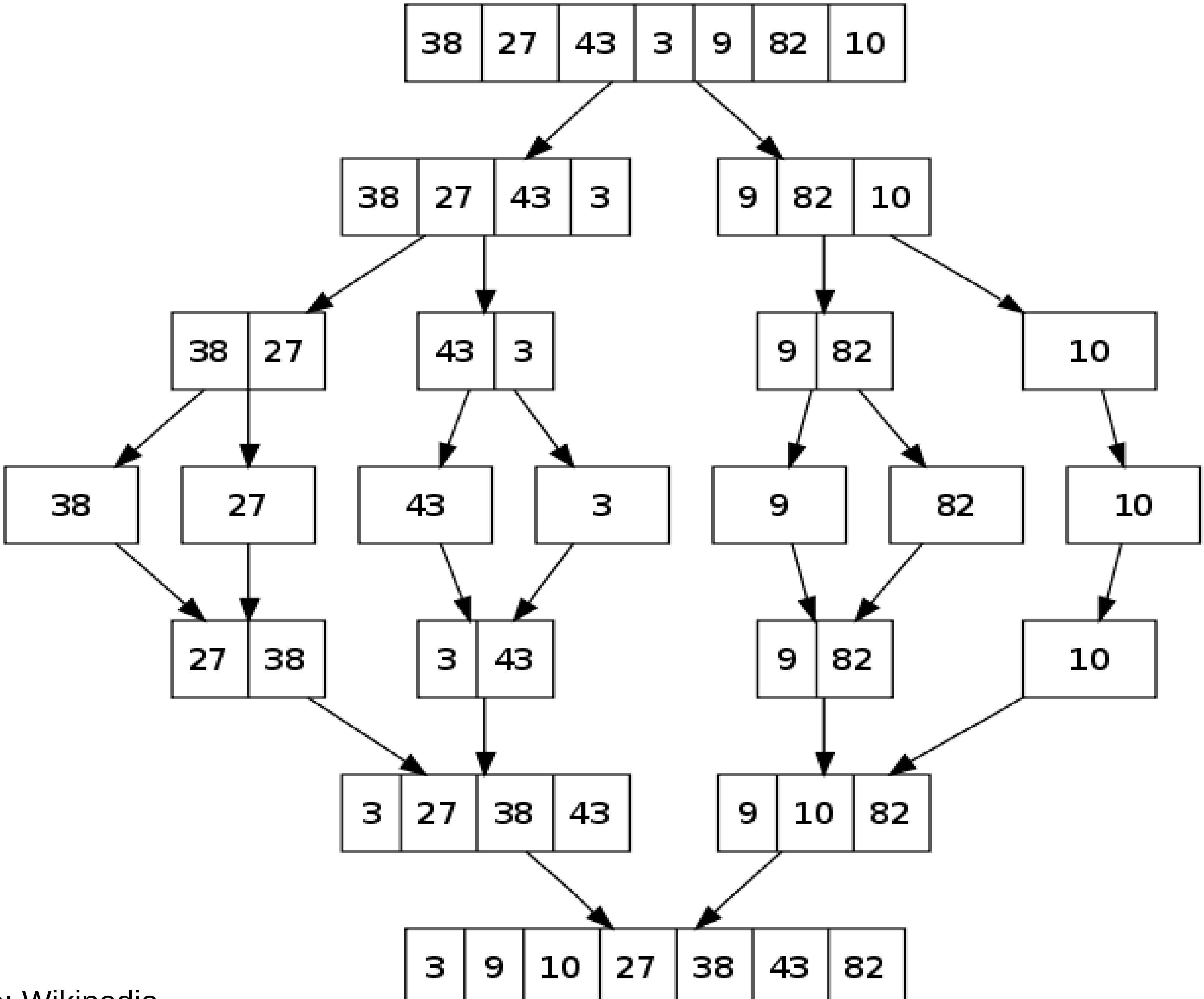
**else:**

X ←MergeSort**(**A[1..n/2], n/2**);**

Y ←MergeSort**(**A[n/2+1, n], n/2**);**

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






Source: Wikipedia

# Challenge 1:

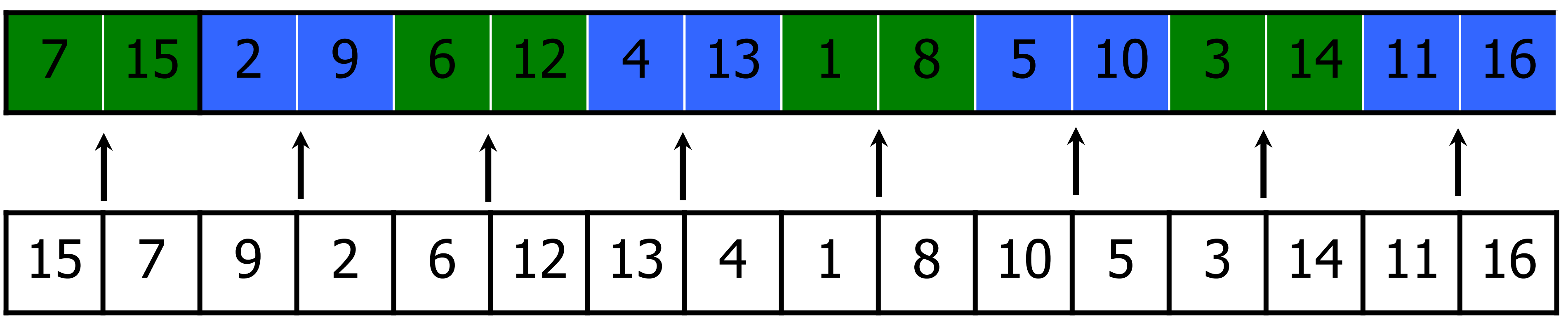
Write an *iterative* version of MergeSort.

No recursion allowed!

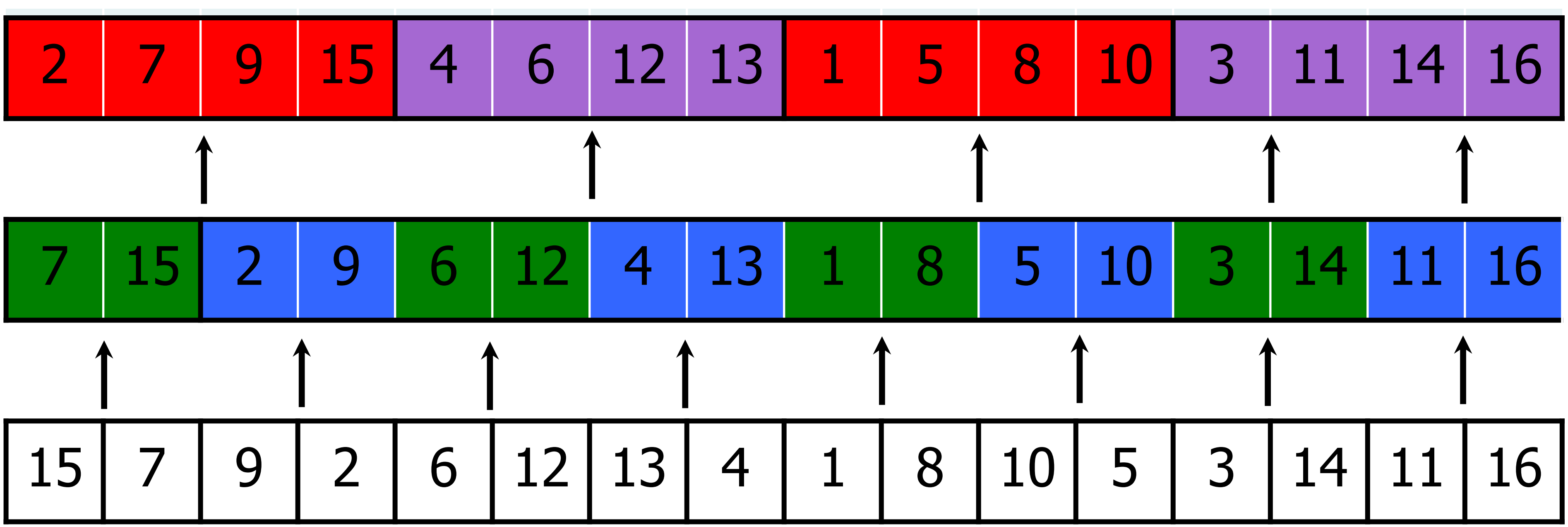
# MergeSort, Bottom Up

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

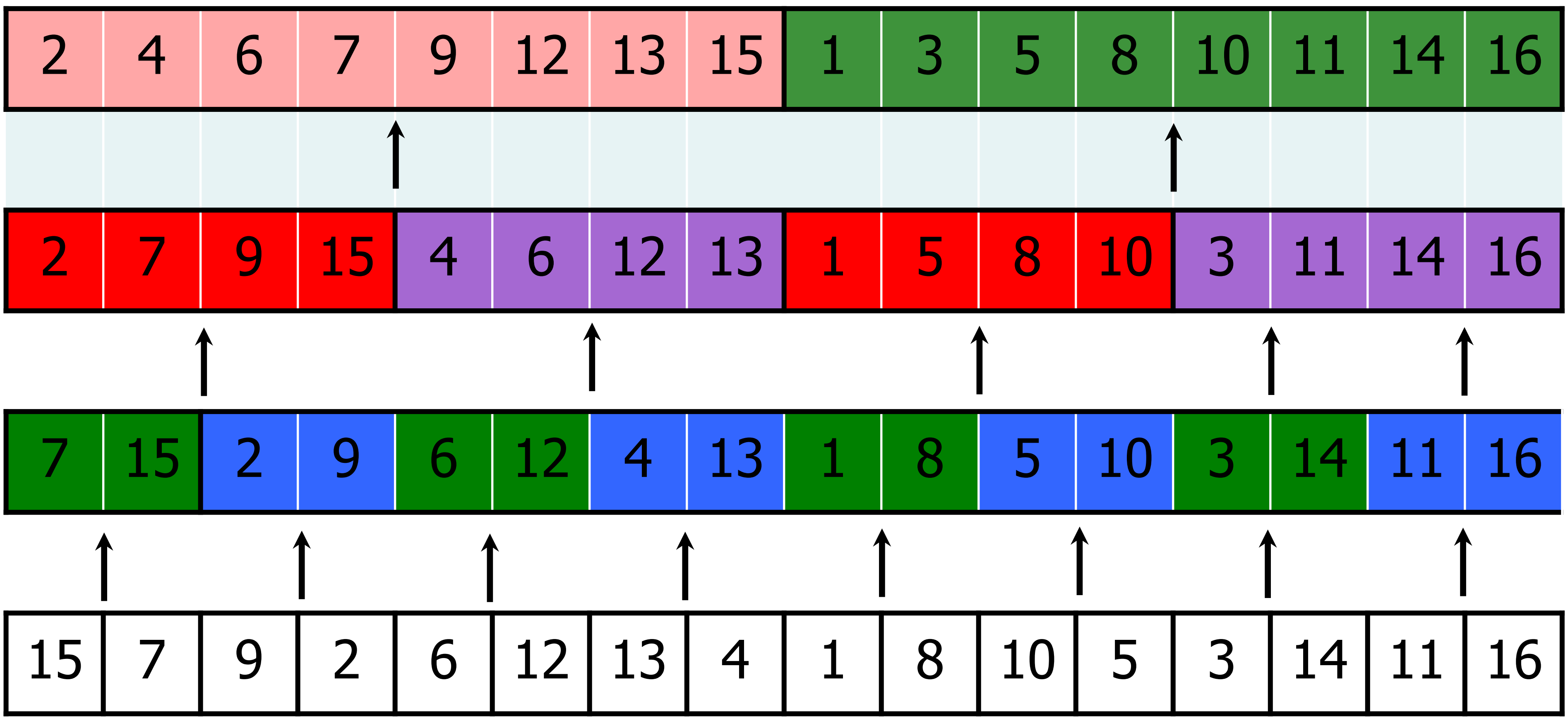
# MergeSort, Bottom Up



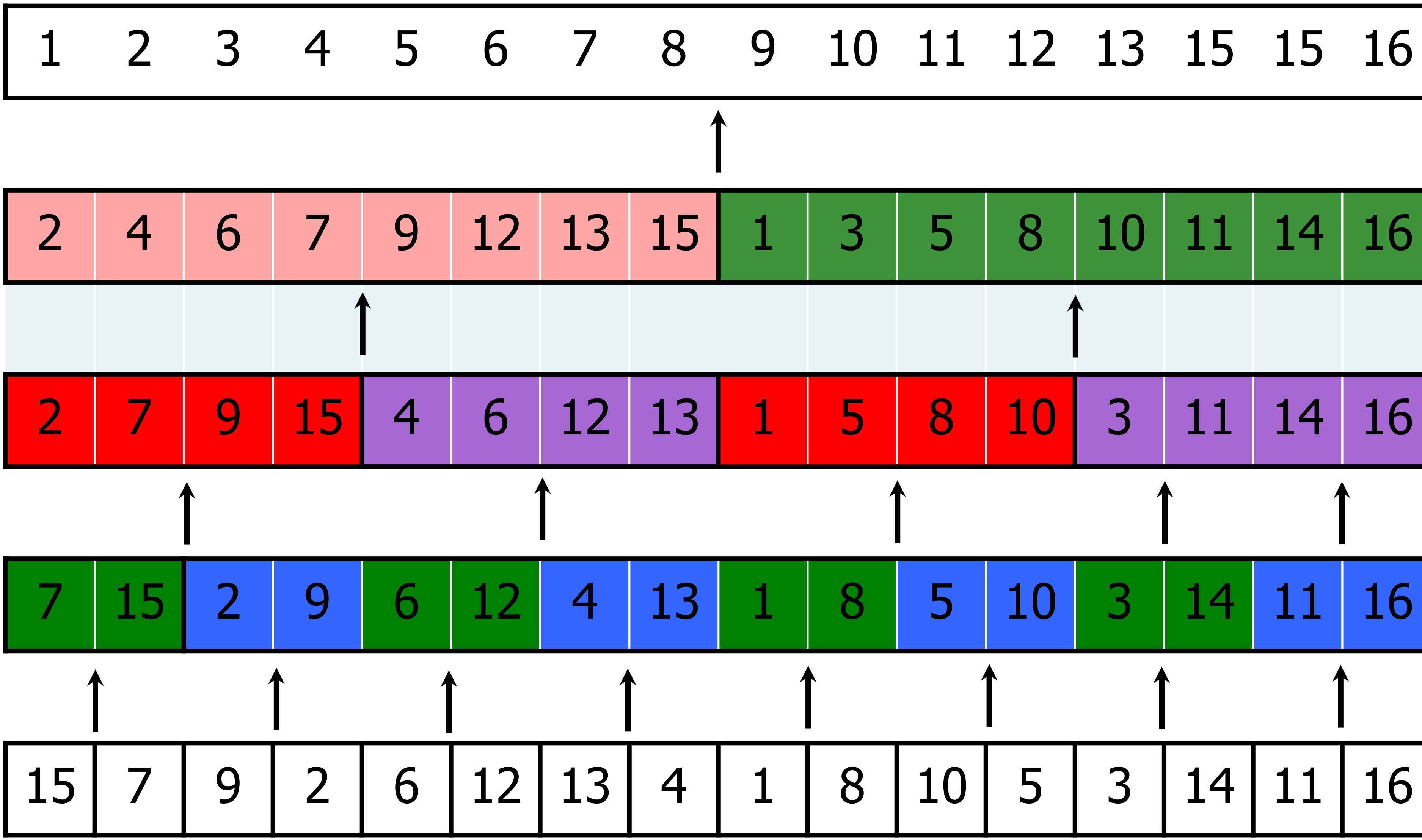
# MergeSort, Bottom Up



# MergeSort, Bottom Up



# MergeSort, Bottom Up



# MergeSort

Space usage…

- Need extra space to do merge.
- Merge copies data to new array.

# Space Complexity

Question:

How much space is allocated during a call to MergeSort?

Note:

Measure total allocated space.

We will not model *garbage collection* or other Java details.

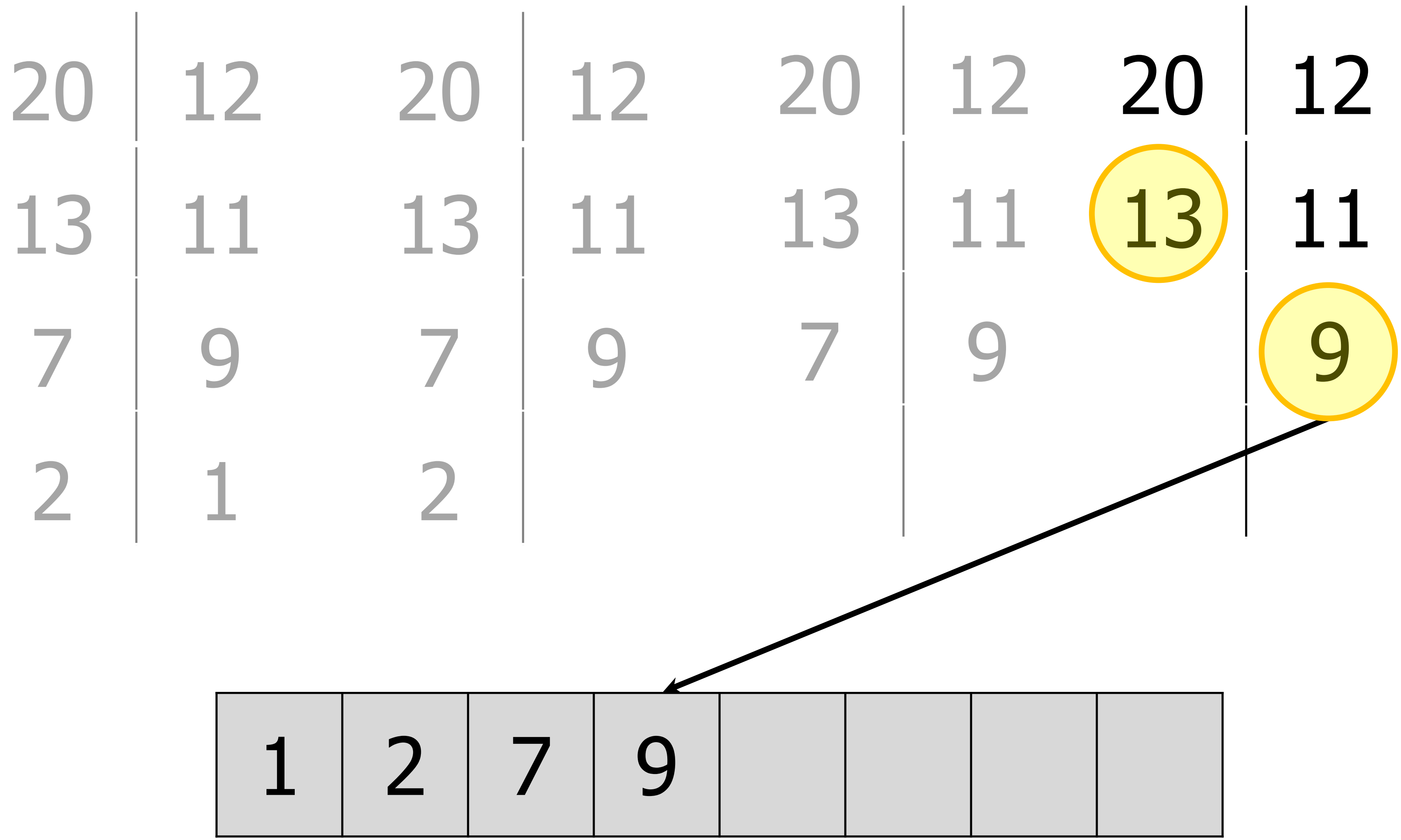
# Space Complexity

Question:

How much space is allocated during a call to MergeSort?

Key subroutine: Merge

# Merging Two Sorted Lists




Need temporary array of size n.

# Space Analysis

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

MergeSort(A, n)

**if** $(n=1)$ **then return;** ← - - - - - - - - - - - $\Theta(1)$

**else:**

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

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

**return** Merge **(**$X, Y, n/2$**);** ← - - - - - - - - - $n$

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

$S(n) = ?$

A. $O(\log n)$
B. $O(n)$
C. $O(n \log n)$
D. $O(n^2)$
E. $O(n^2 \log n)$
F. $O(2^n)$

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

$S(n) = ?$

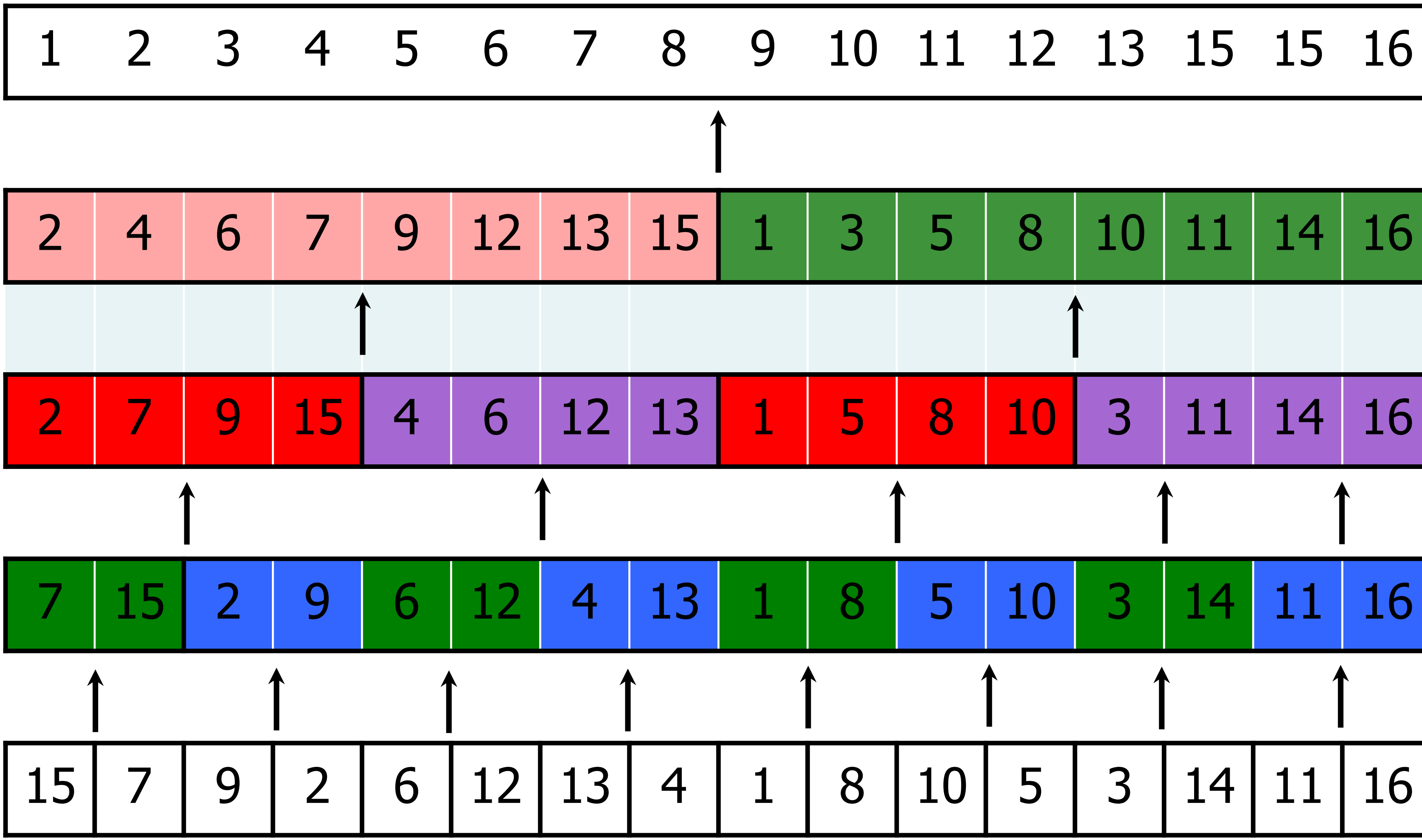
A. $O(\log n)$
B. $O(n)$
✓ C. $O(n \log n)$
D. $O(n^2)$
E. $O(n^2 \log n)$
F. $O(2^n)$

# Space Analysis

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

$$
\begin{aligned}
S(n) &= \theta(1) && \textbf{if } (n=1) \\
&= 2S(n/2) + n && \textbf{if } (n>1) \\
&= O(n \log n)
\end{aligned}
$$

# MergeSort


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

# Space Analysis

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

*merge*

*recursive sort* **n** *recursive sort*

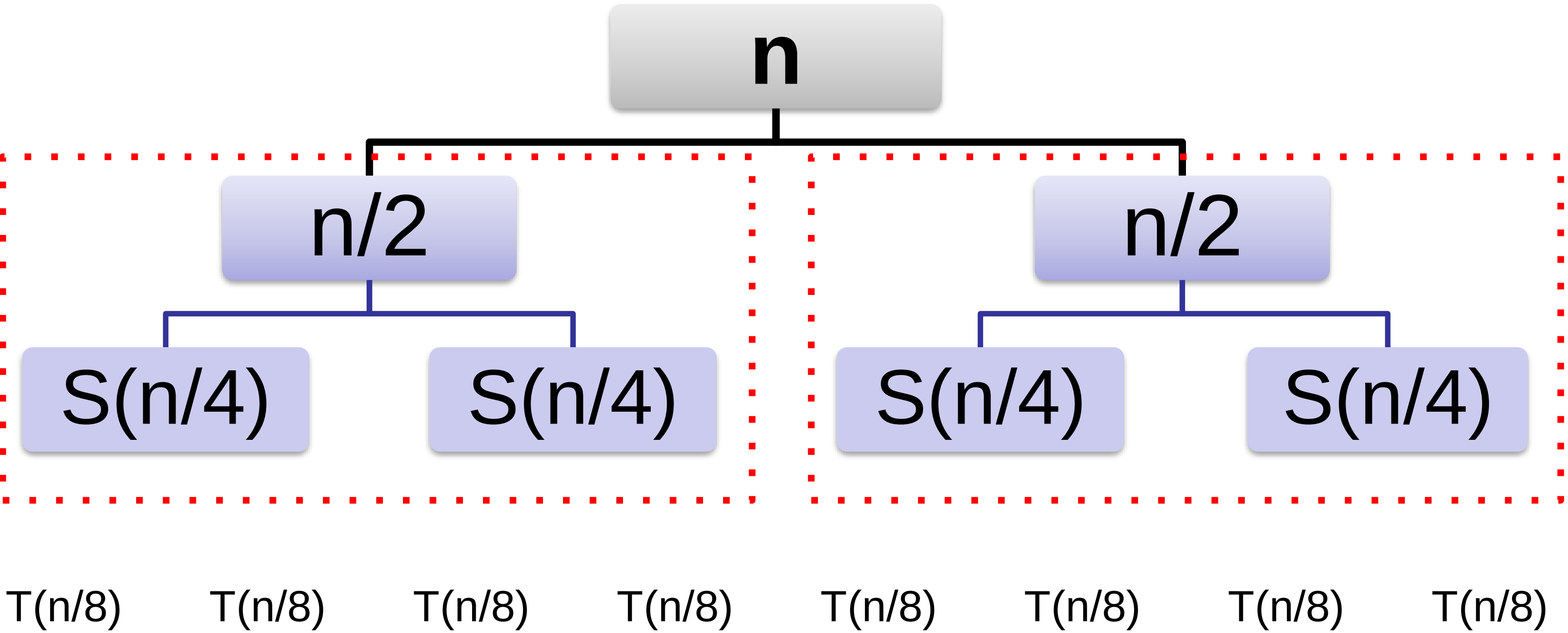
S(n/2) S(n/2)

cn/4 cn/4 cn/4 cn/4

T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8)

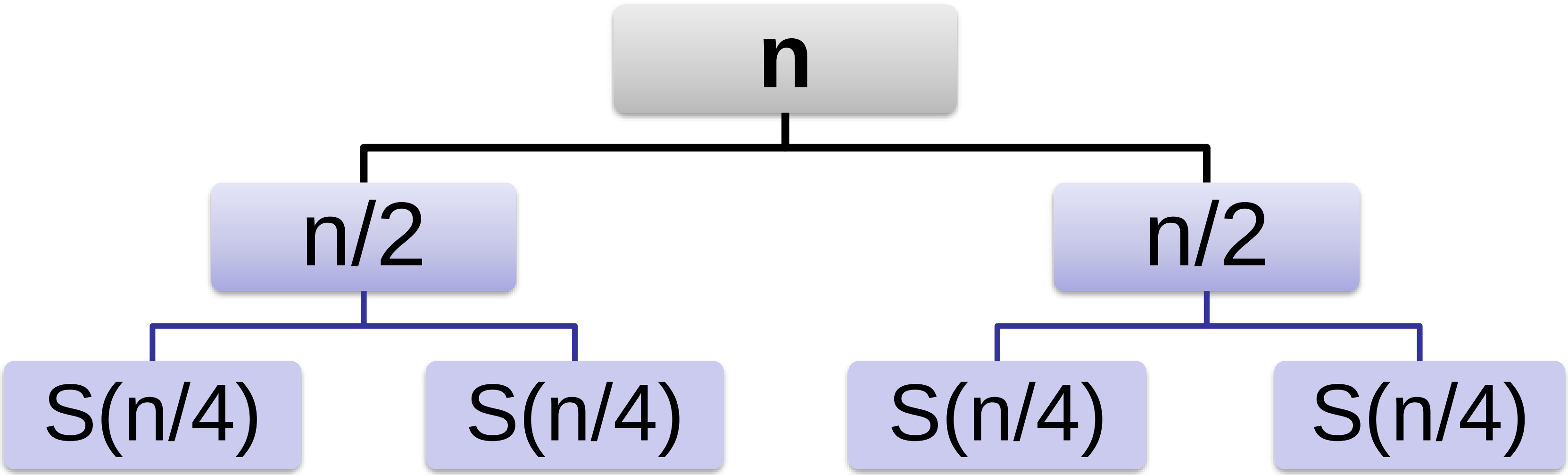
# Space Analysis

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



# Space Analysis

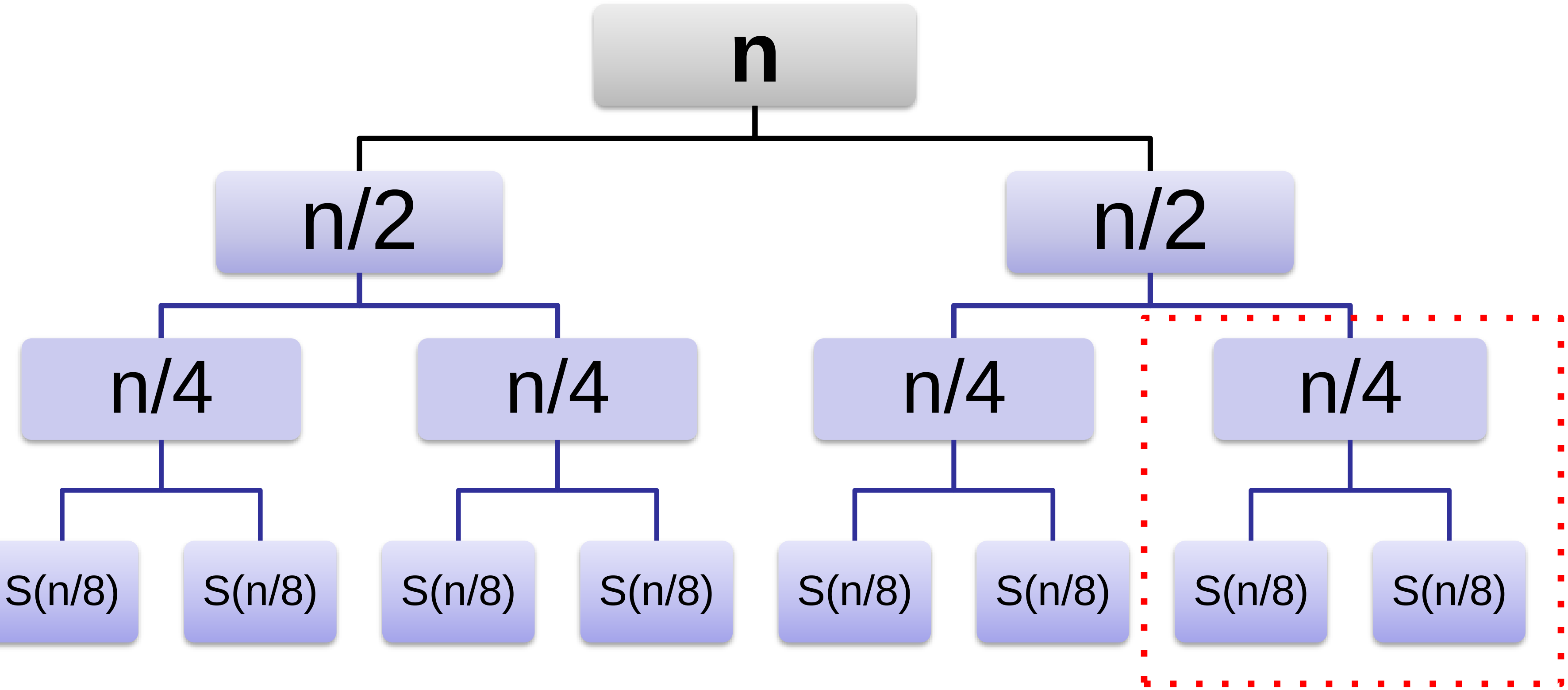
$$S(n) = 2S(n/2) + n$$




T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8)

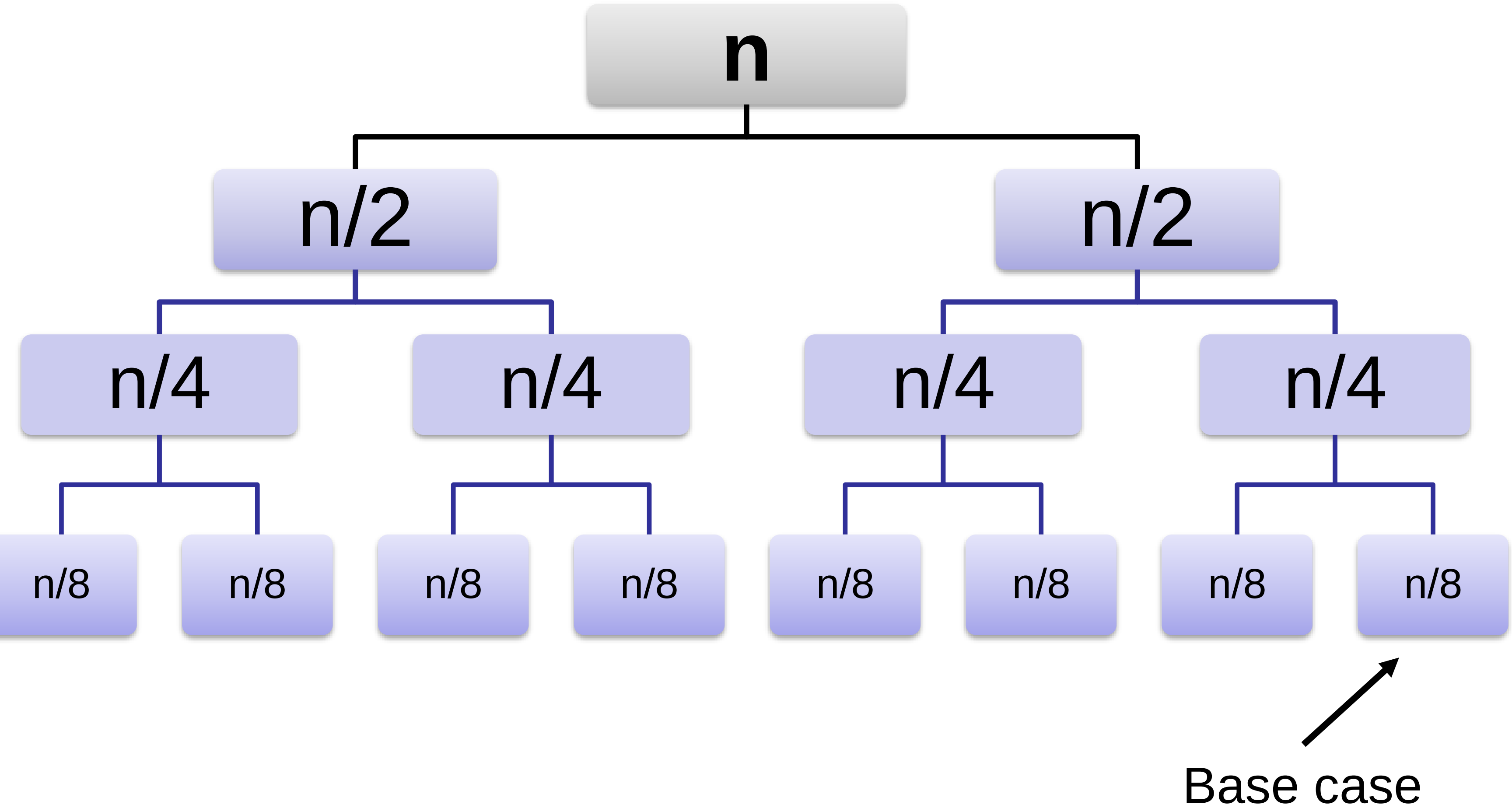
# Space Analysis

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



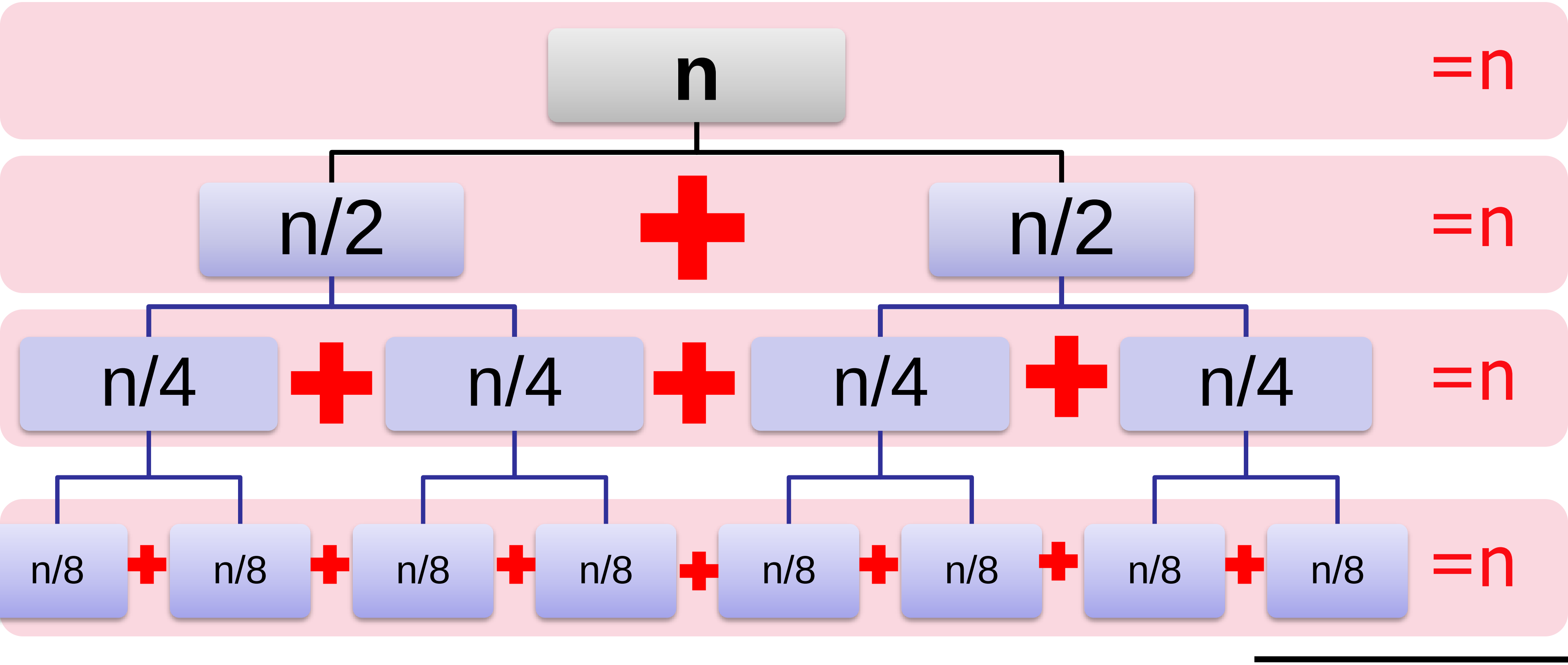
# Space Analysis

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



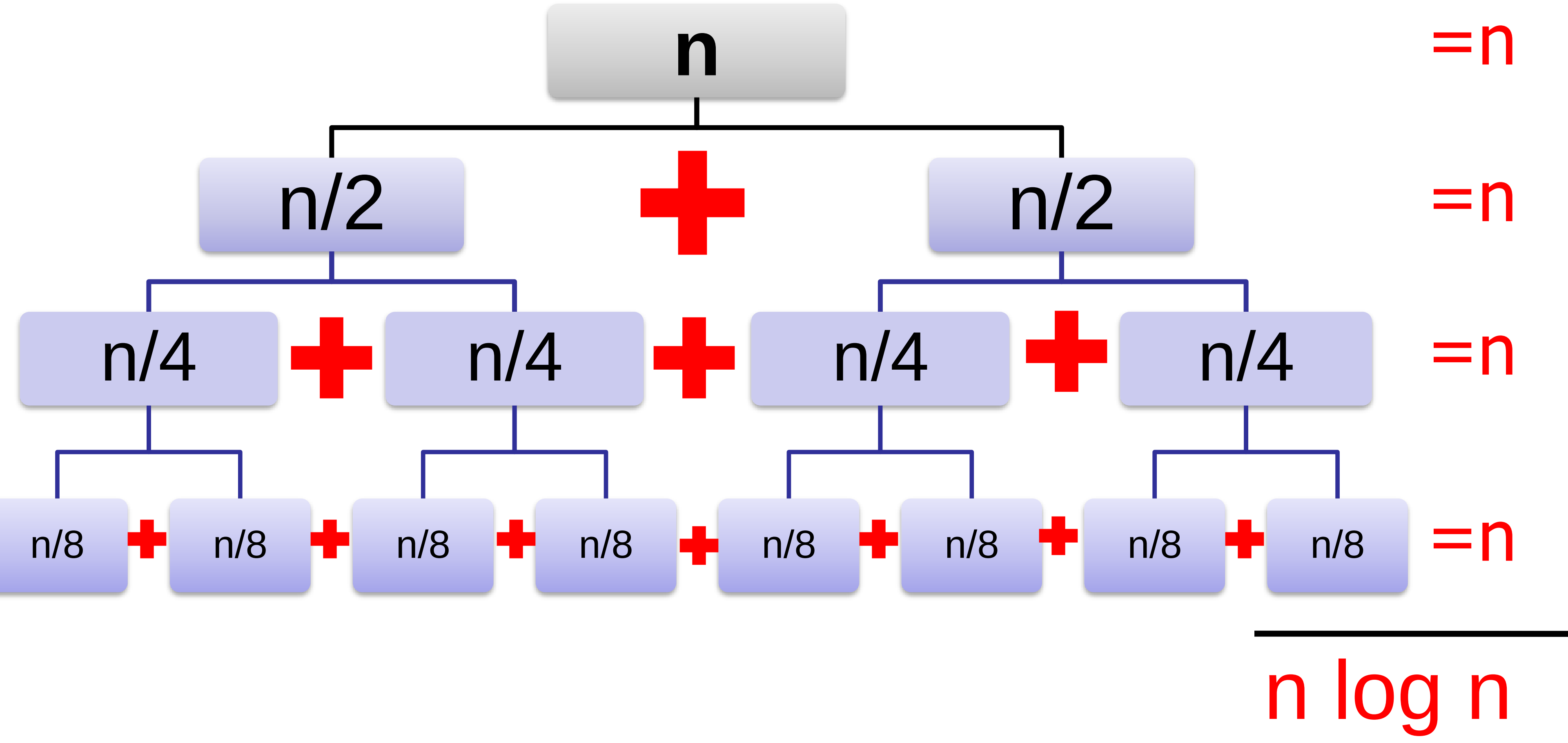
# Space Analysis

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



# Space Analysis

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



# Space Analysis

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

MergeSort(A, n)

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

**else:** ←------------ $\Theta(1)$

X ←MergeSort**(**…**);**

Y ←MergeSort**(**…**);** ←------- $S(n/2)$

**return** Merge **(**X,Y, n/2**);** ←------- $S(n/2)$

←--------- $\Theta(n)$

## Challenge 2:

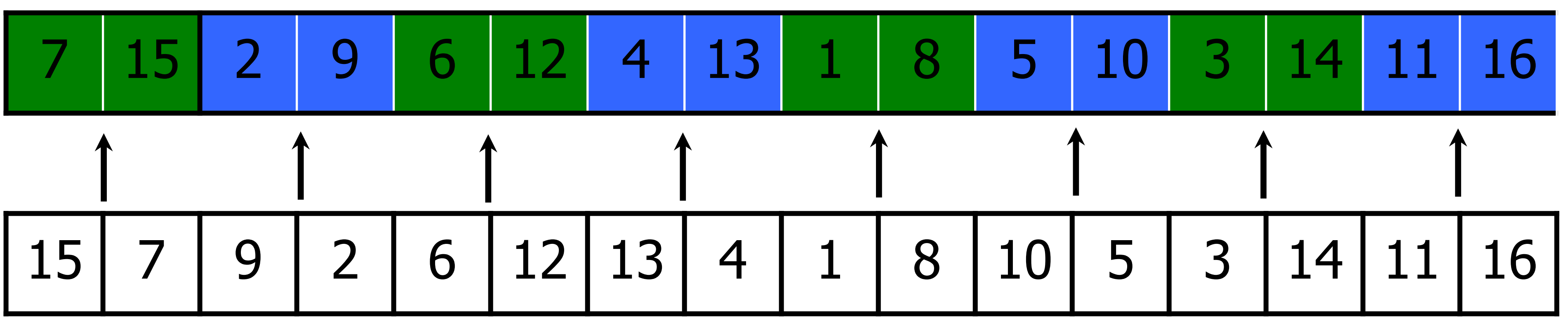
Design a version of MergeSort that minimizes the amount of extra space needed.

Hint: Do not allocate any new space during the recursive calls!

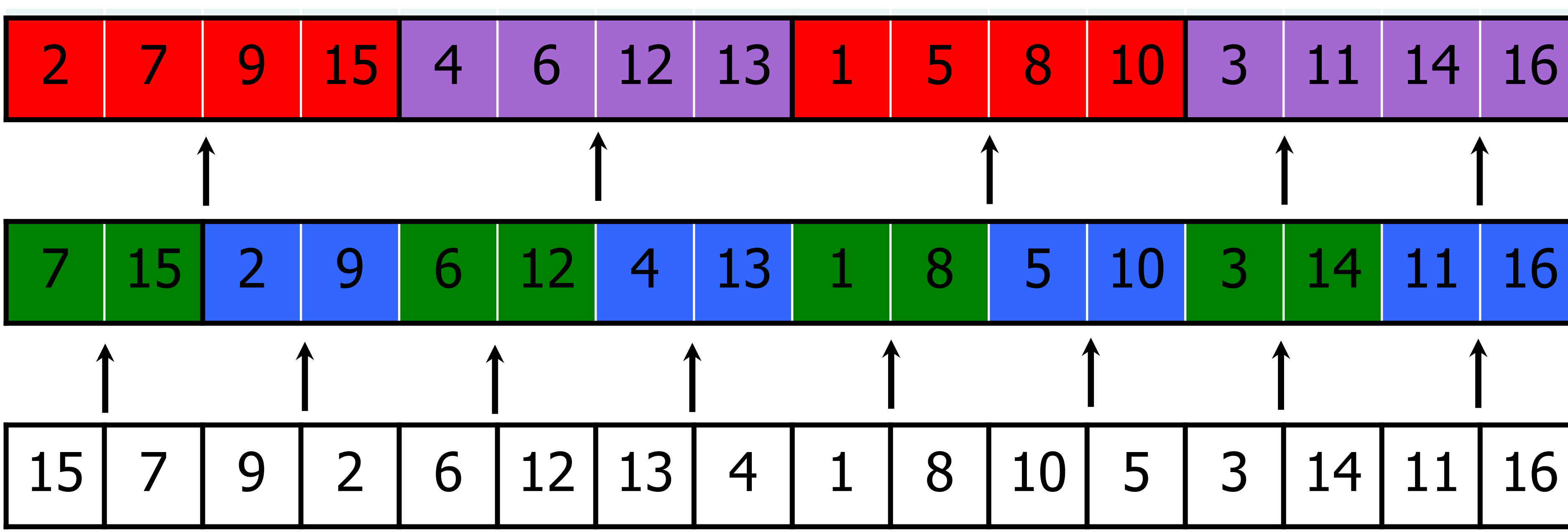
# MergeSort: log(n) arrays

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

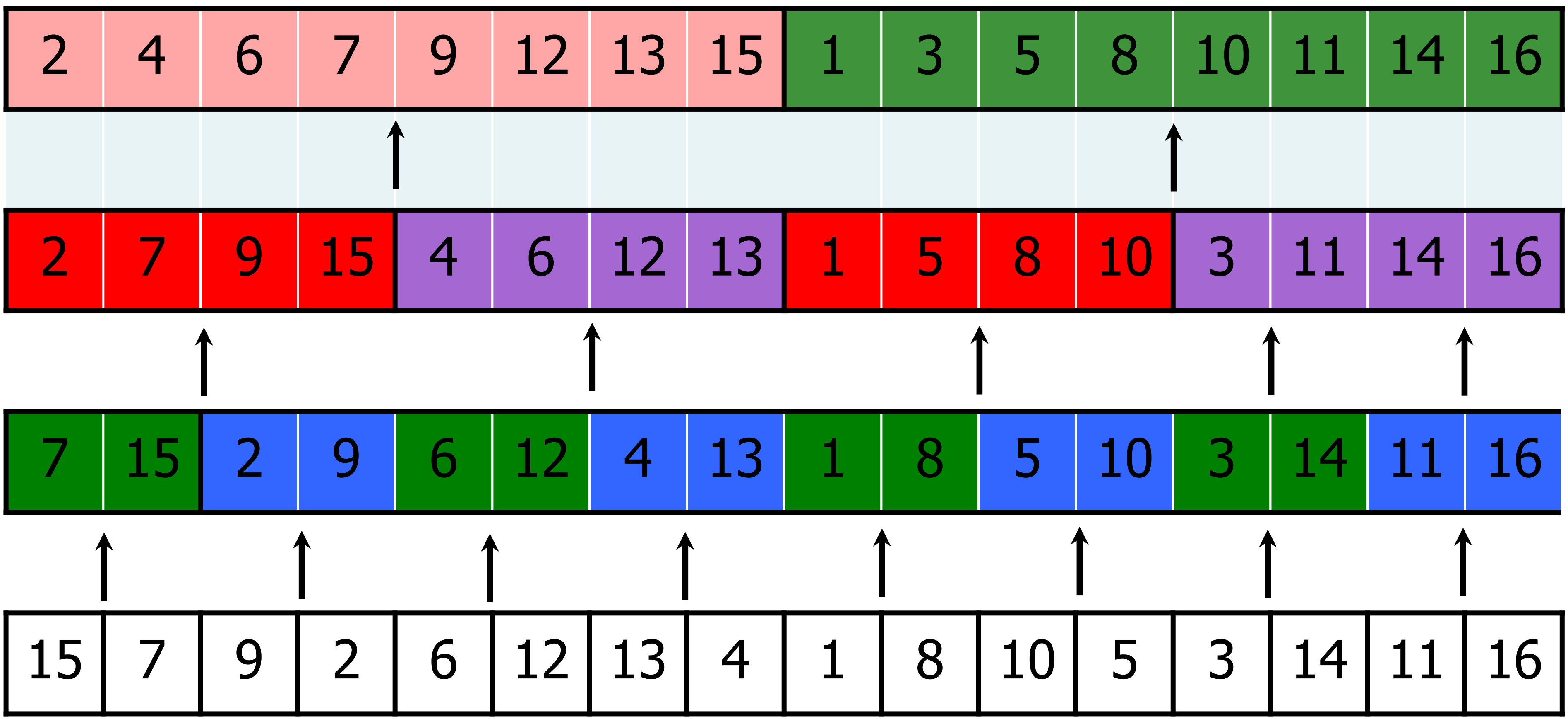
# MergeSort: log(n) arrays



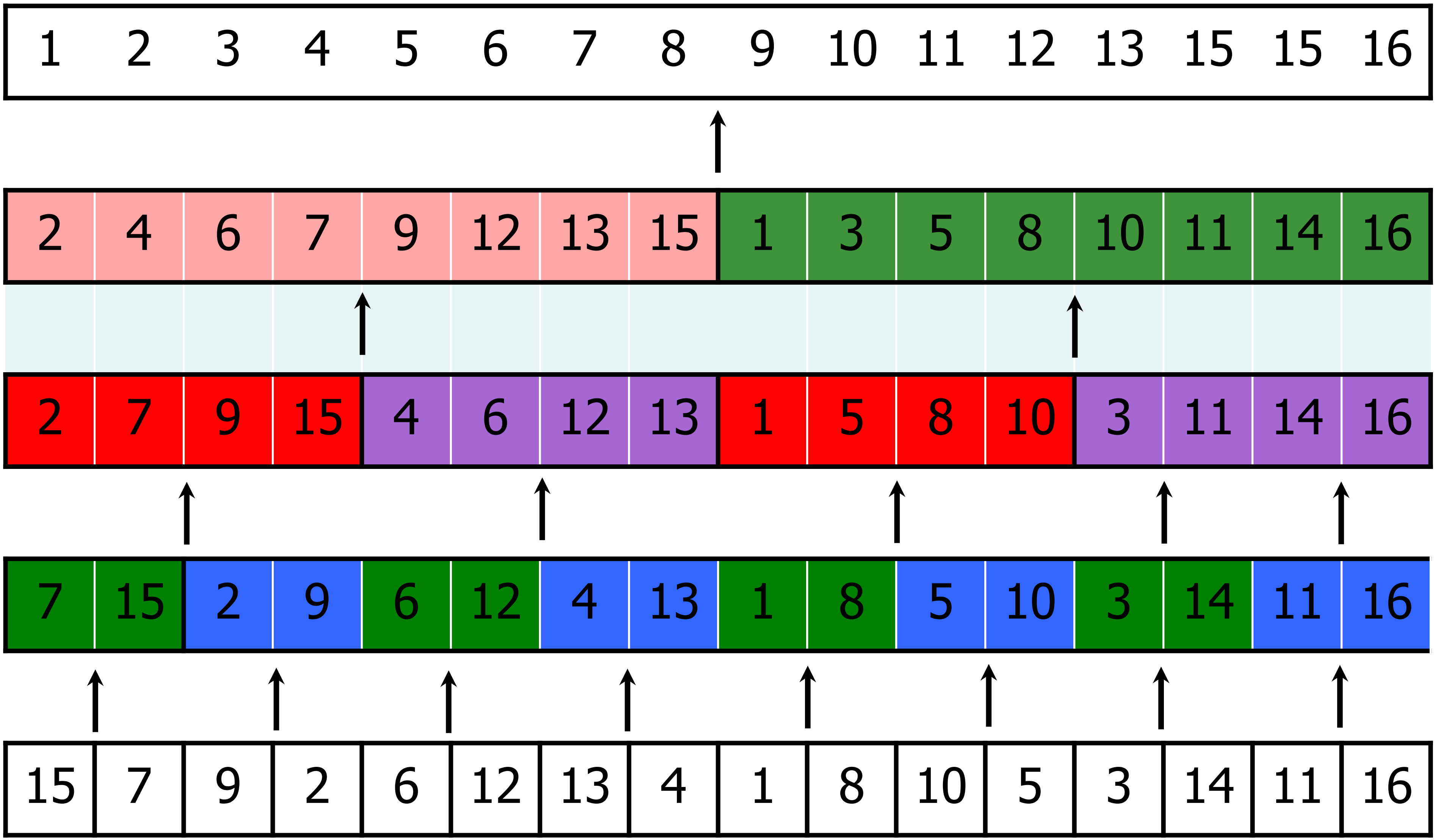
# MergeSort: log(n) arrays



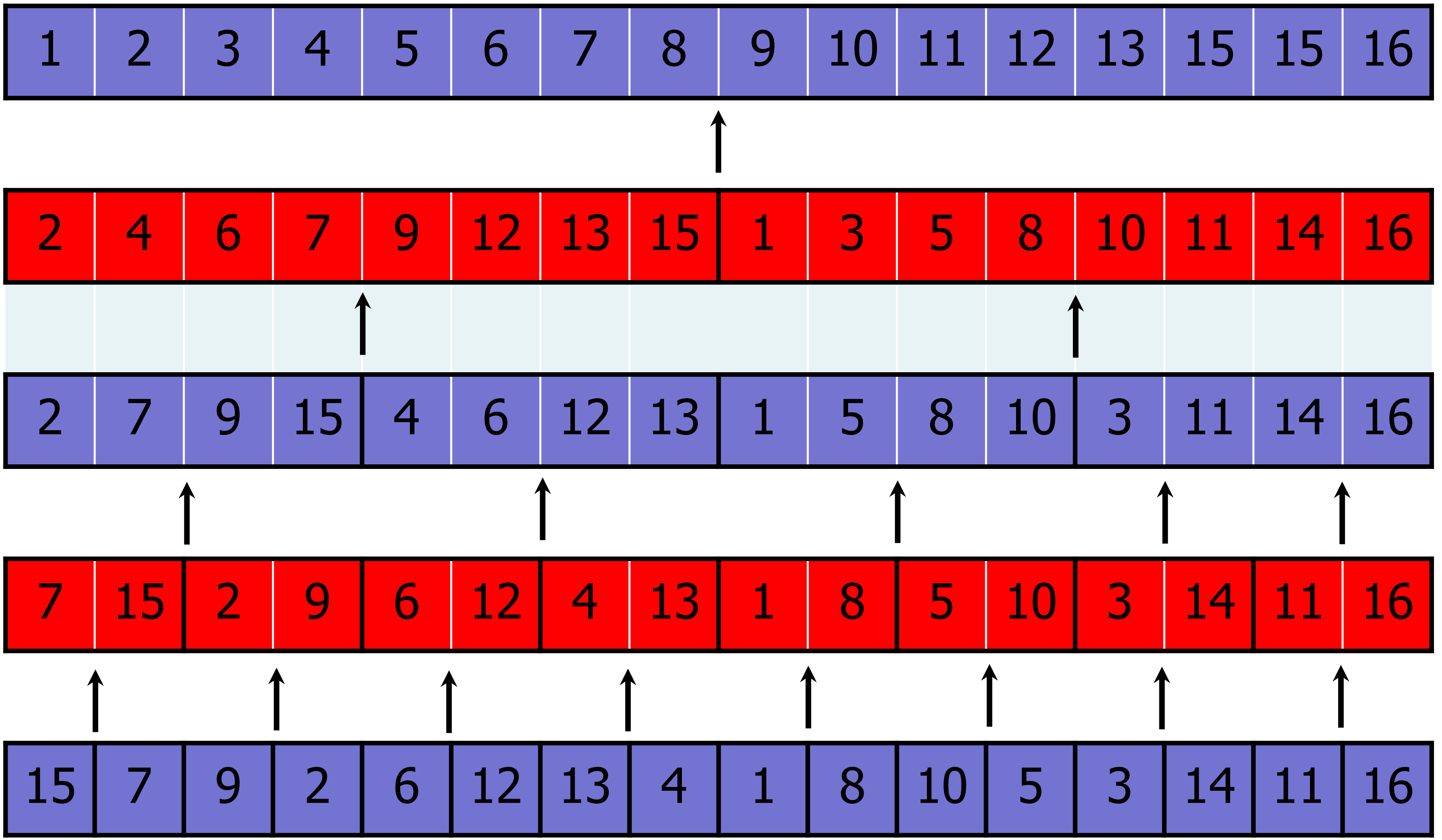
# MergeSort: log(n) arrays



# MergeSort: log(n) arrays



# MergeSort: 2 arrays



# Better Space Usage

Use only one temporary array!

On termination, items in range [begin,end] are sorted in A.

The tempArray is used for workspace.

```
MergeSort(A, begin, end, tempArray)
    if (begin=end) then return;
    else:
        mid = begin + (end-begin)/2
        MergeSort(A, begin, mid, tempArray);
        MergeSort(A, mid+1, end, tempArray);
      Merge(A[begin..mid], A[mid+1, end], tempArray);
      Copy(tempArray, A, begin, end);
```

Merge copies items into tempArray.

We then copy the items back into array A.

# Better Space Usage

$S(n) = 2S(n/2) + O(1) = O(n)$

```
MergeSort(A, begin, end, tempArray)
    if (begin=end) then return;
    else:
        mid = begin + (end-begin)/2
        MergeSort(A, begin, mid, tempArray);
        MergeSort(A, mid+1, end, tempArray);
        Merge(A[begin..mid], A[mid+1, end], tempArray);
        Copy(tempArray, A, begin, end);
```

# Better Space Usage

Still a problem: can we avoid the extra copying of data?

```
MergeSort(A, begin, end, tempArray)
        if (begin=end) then return;
        else:
                mid = begin + (end-begin)/2
                MergeSort(A, begin, mid, tempArray);
                MergeSort(A, mid+1, end, tempArray);
                Merge(A[begin..mid], A[mid+1, end], tempArray);
                Copy(tempArray, A, begin, end);
```

# Better Space Usage

Idea: switch temporary array at every step!

```
MergeSort(A, B, begin, end)
    if (begin=end) then return;
    else:
        mid = begin + (end-begin)/2
        MergeSort(B, A, begin, mid);
        MergeSort(B, A, mid+1, end);
        Merge(A, B, begin, mid, end);
        ~~Copy(B, A, begin~~, end);
```

Initially, both A and B have copies of the unsorted array.

Switch the order of A and B at every recursive call.

# Today: more sorting!

QuickSort:

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

# QuickSort

History:

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

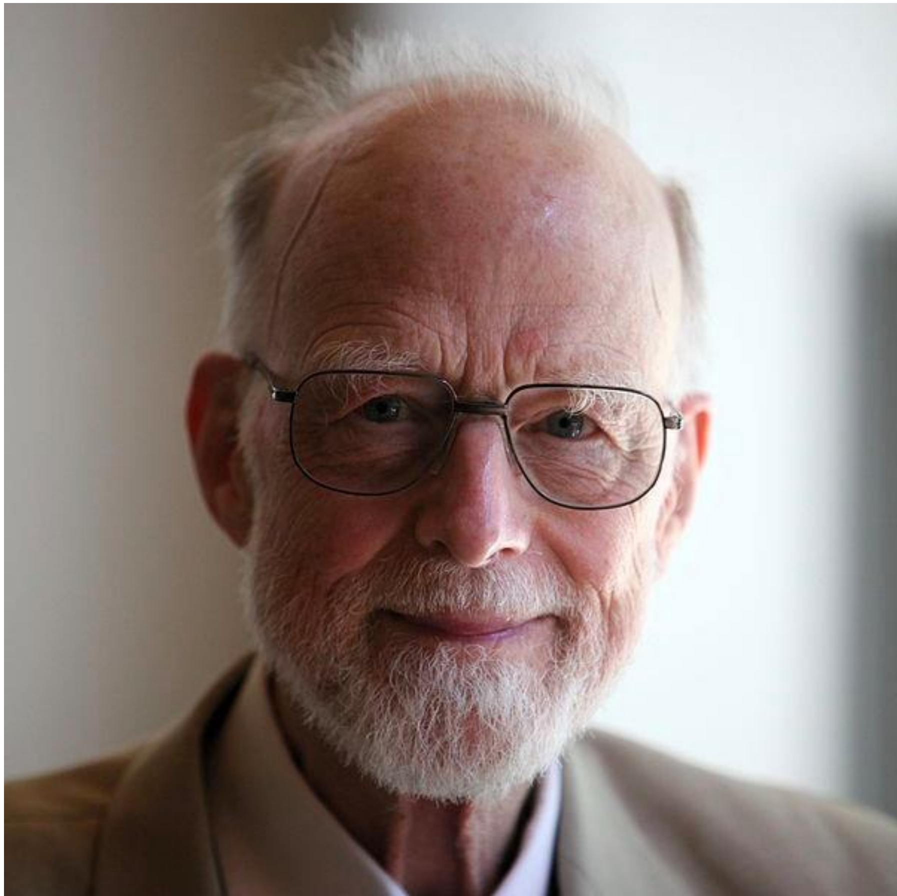


Photo: Wikimedia Commons (Rama)

# Hoare

Quote:

"There are two ways of constructing a software design:

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

The first method is far more difficult."

# QuickSort

History:

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

In practice:

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

# QuickSort Today

1960: Invented by Hoare

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

1993: Bentley & McIlroy improvements

"Engineering a sort function"

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

# QuickSort Today

1960: Invented by Hoare

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

1993: Bentley & McIlroy improvements

”Ok, QuickSort is done,” said everyone.

Every algorithms class since 1993:

Punk in the front row:

“But what if we used more pivots?”

Every algorithms class since 1993:

Punk in the front row:

"But what if we used more pivots?"

Professor:

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

In 2009:

Punk in the front row:

“But what if we used more pivots?”

Professor:

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

Punk in the front row:

“Huh… let me try it.  Wait a sec, it’s faster!”

# QuickSort Today

1960: Invented by Hoare

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

1993: Bentley & McIlroy improvements

2009: Vladimir Yaroslavskiy

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

# QuickSort Today

1960: Invented by Hoare

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

1993: Bentley & McIlroy improvements

2009: Vladimir Yaroslavskiy

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

2012: Sebastian Wild and Markus E. Nebel

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

Moral of the story:

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

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

# QuickSort

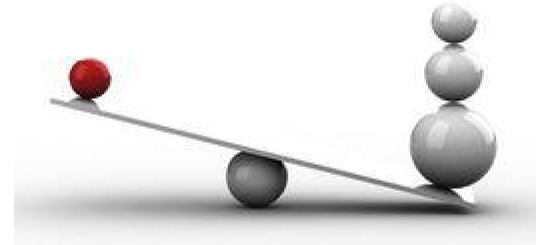
History:

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

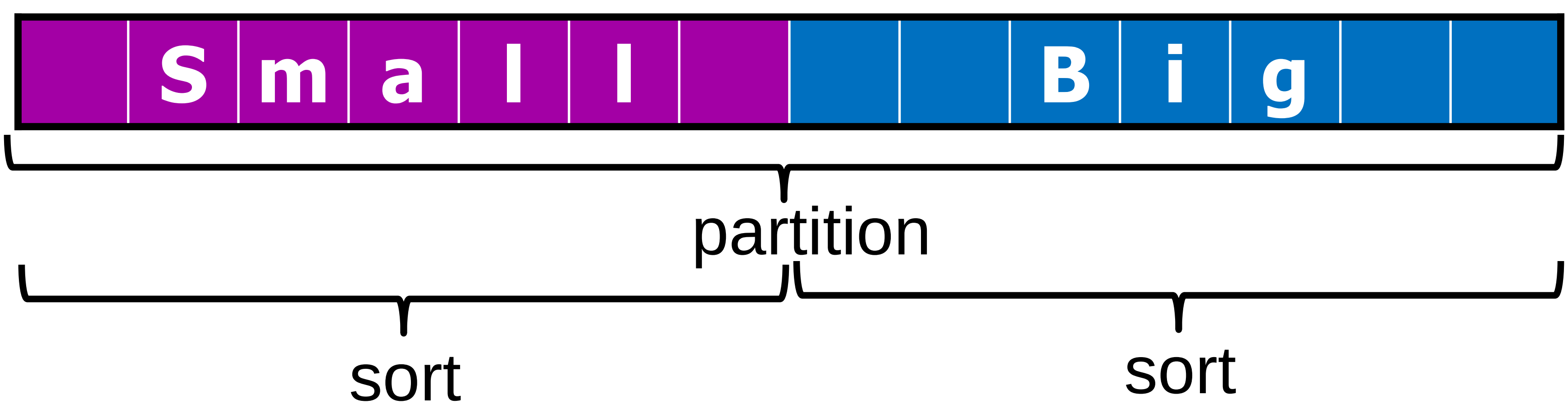
In practice:

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

# QuickSort

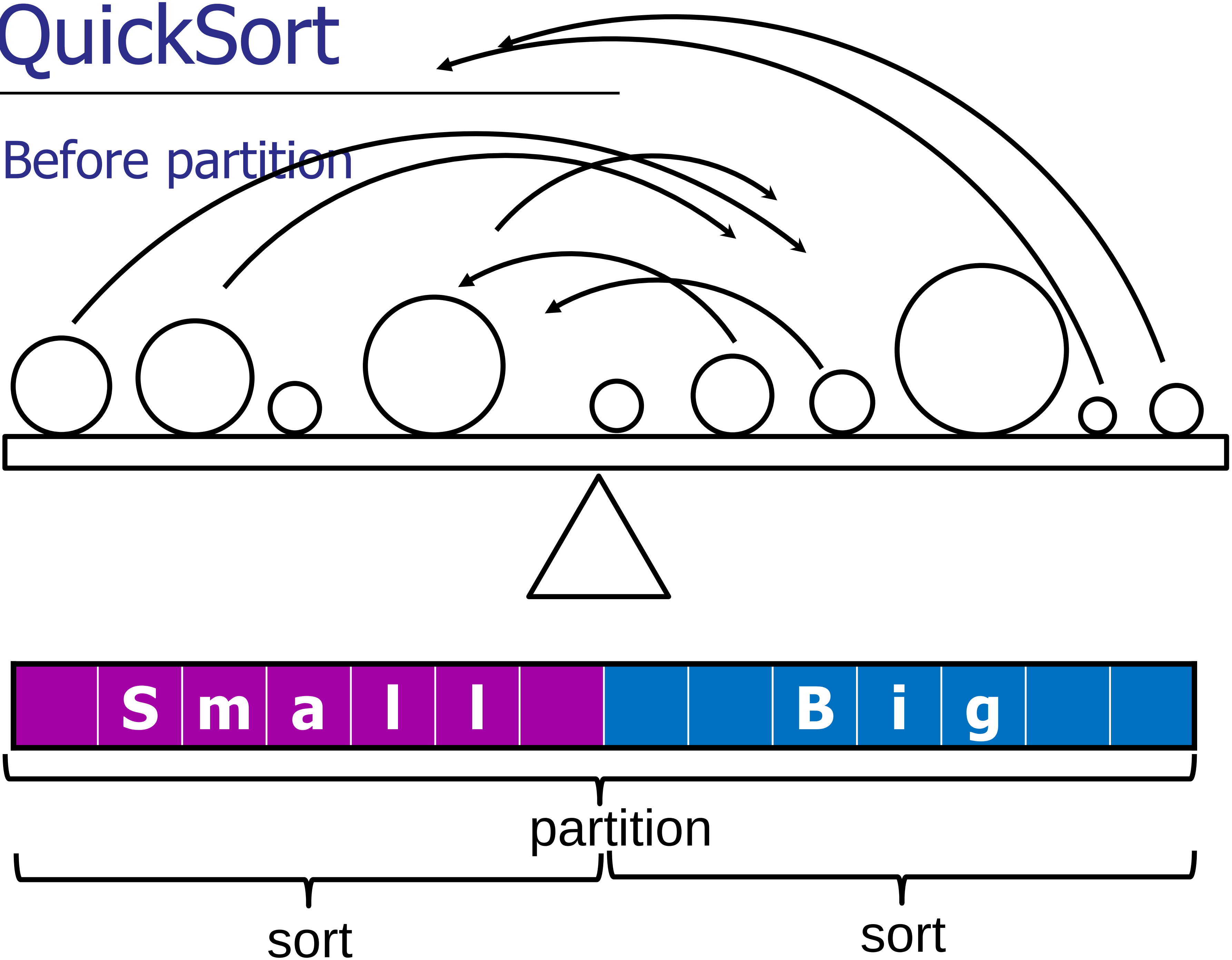


```
QuickSort(A[1..n], n)
    if (n==1) then return;
    else
        p = partition(A[1..n], n)
        x = QuickSort(A[1..p-1], p-1)
        y = QuickSort(A[p+1..n], n-p)
```



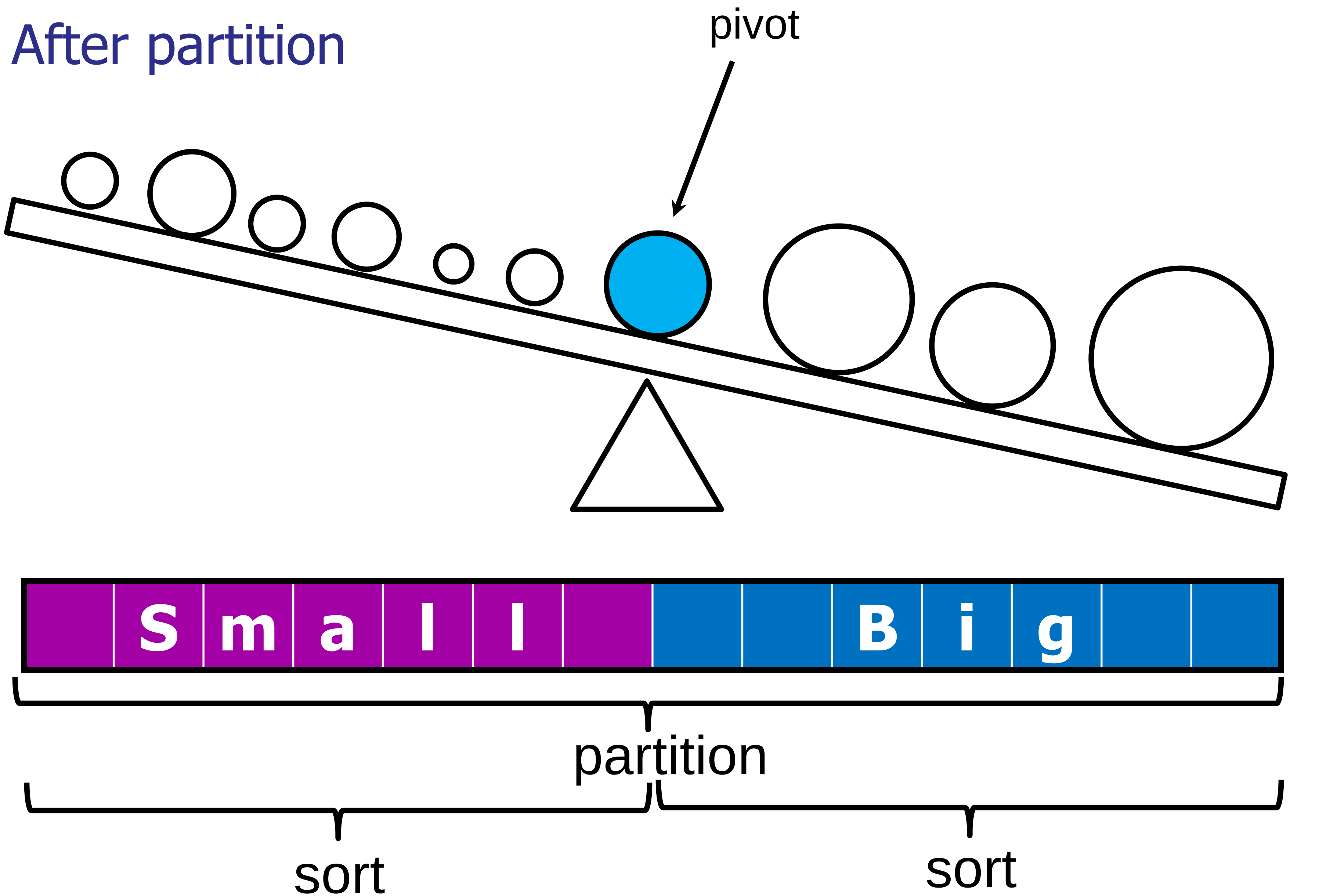
# QuickSort

Before partition

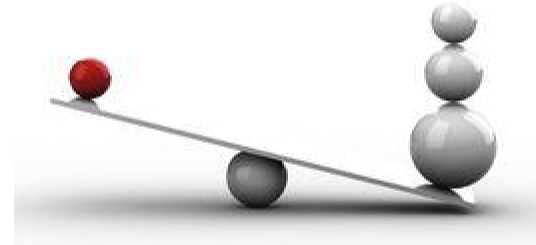


# QuickSort

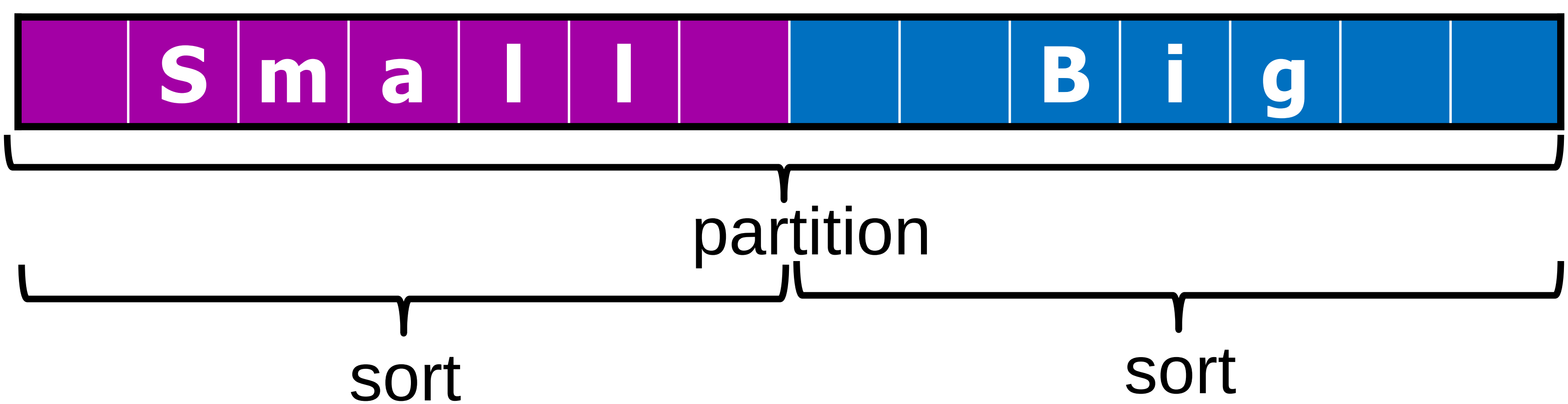
## After partition



# 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.


1. Conquer: Recursively sort the two sub-arrays.
2. 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.

# Quicksort

Example:

6 3 9 8 4 2

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 3 | 9 | 8 | 4 | 2 |
| 3 | 4 | 2 | 6 | 9 | 8 |

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 3 | 9 | 8 | 4 | 2 |
| 3 | 4 | 2 | 6 | 9 | 8 |
| 2 | 3 | 4 | | | |

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 3 | 9 | 8 | 4 | 2 |
| 3 | 4 | 2 | 6 | 9 | 8 |
| 2 | 3 | 4 | | 8 | 9 |

# Quicksort

Example:

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

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 3 | 9 | 8 | 4 | 2 |
| 3 | 4 | 2 | 6 | 9 | 8 |
| 2 | 3 | 4 | 6 | 8 | 9 |

## The following array has been partitioned around which element?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 18 | 5 | 6 | 1 | 10 | 22 | 40 | 32 | 31 |

a. 6
b. 10
c. 22
d. 40
e. 32
f. I don’t know.

The following array has been partitioned around which element?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 18 | 5 | 6 | 1 | 10 | 22 | 40 | 32 | 31 |

a. 6

b. 10

c. 22

d. 40

e. 32

f. I don’t know.

# Partitioning an Array

Example: partition around 22

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

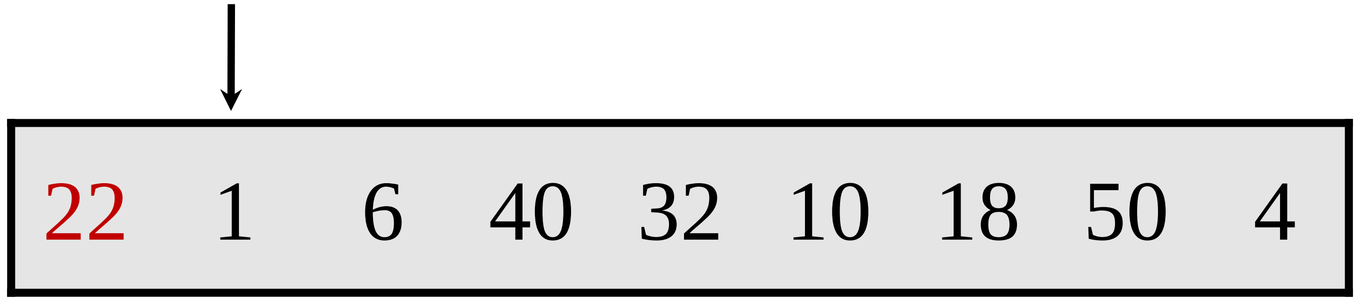
Output array:

*low*
< 22

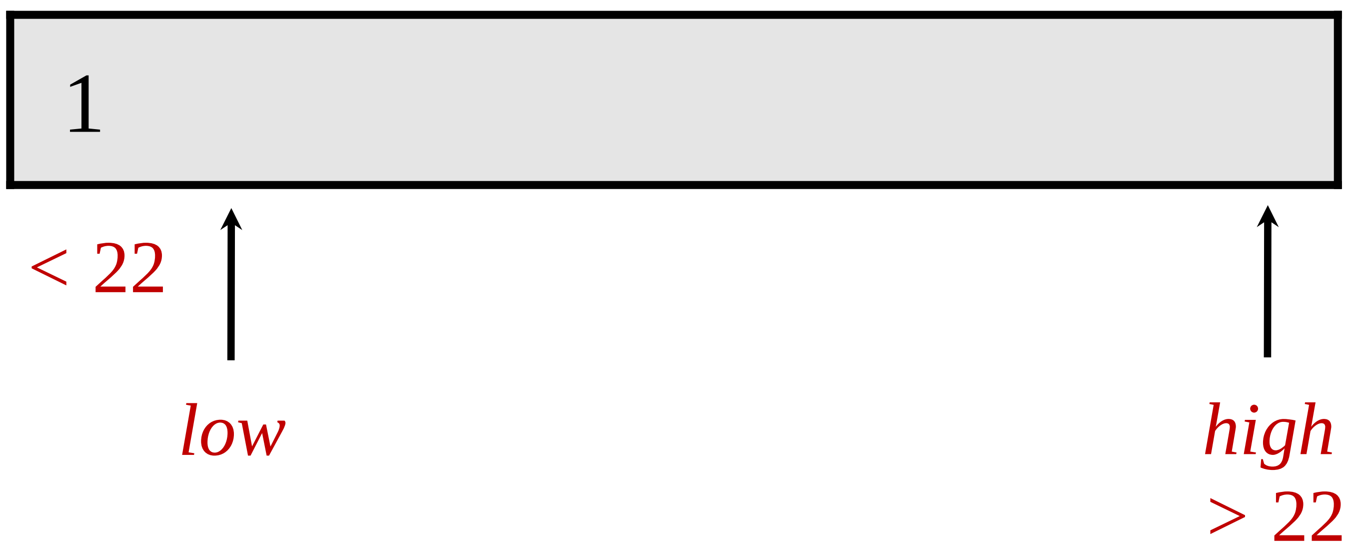
*high*
> 22

# Partitioning an Array

Example: partition around 22

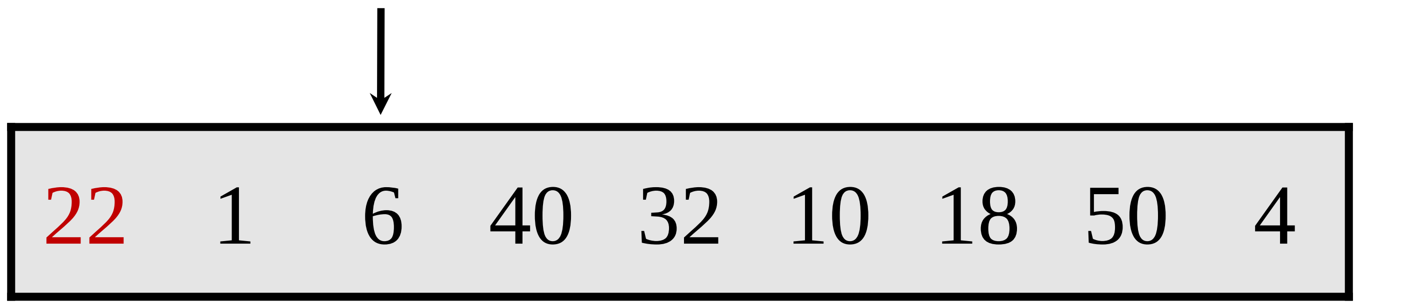



Output array:

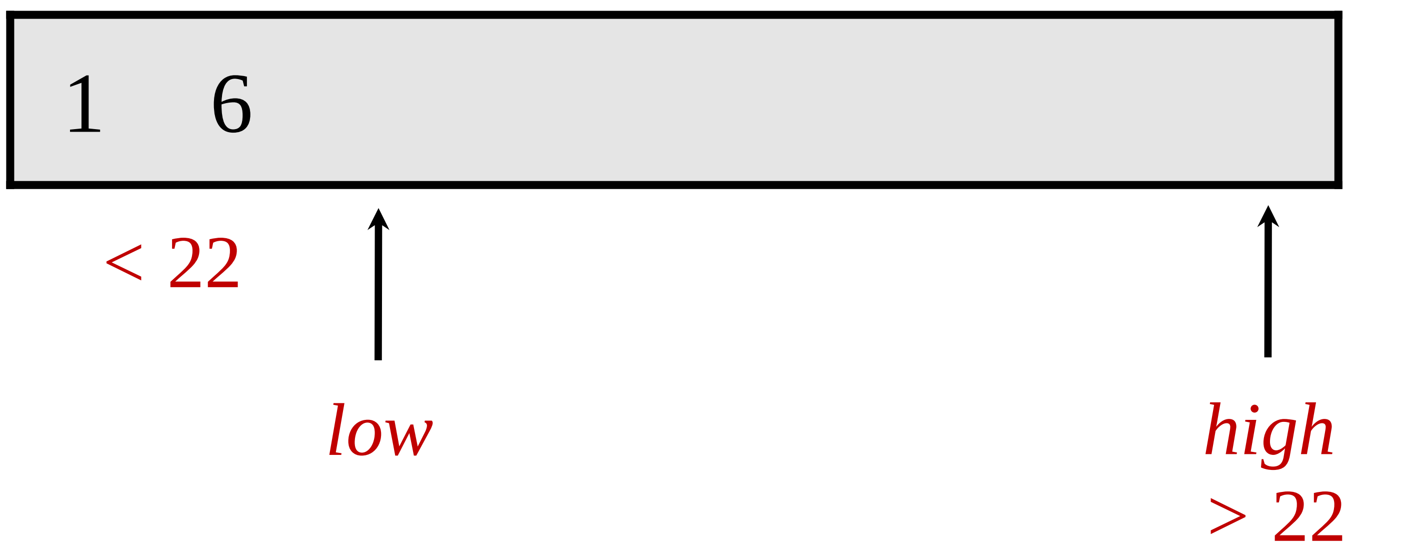


# Partitioning an Array

Example: partition around 22




Output array:



# Partitioning an Array

Example: partition around 22

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

Output array:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 6 | | | | | | | 40 |

< 22

*low*

*high*

> 22

# Partitioning an Array

Example: partition around 22

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

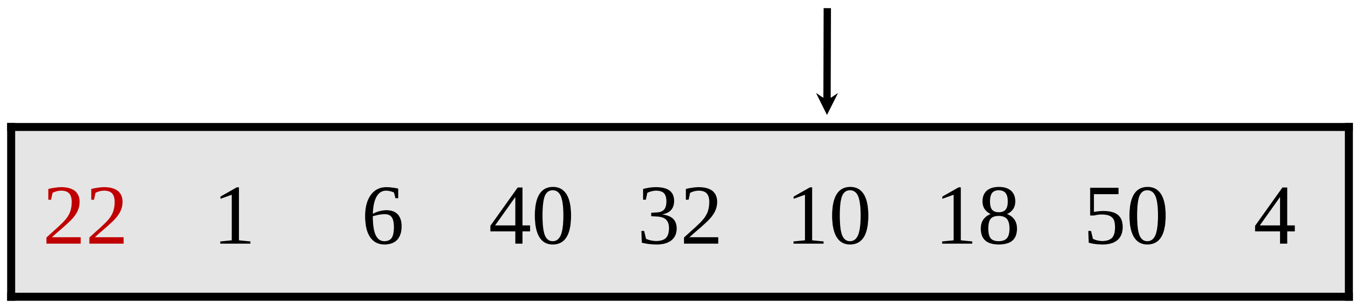
Output array:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 6 | | | | | | 32 | 40 |

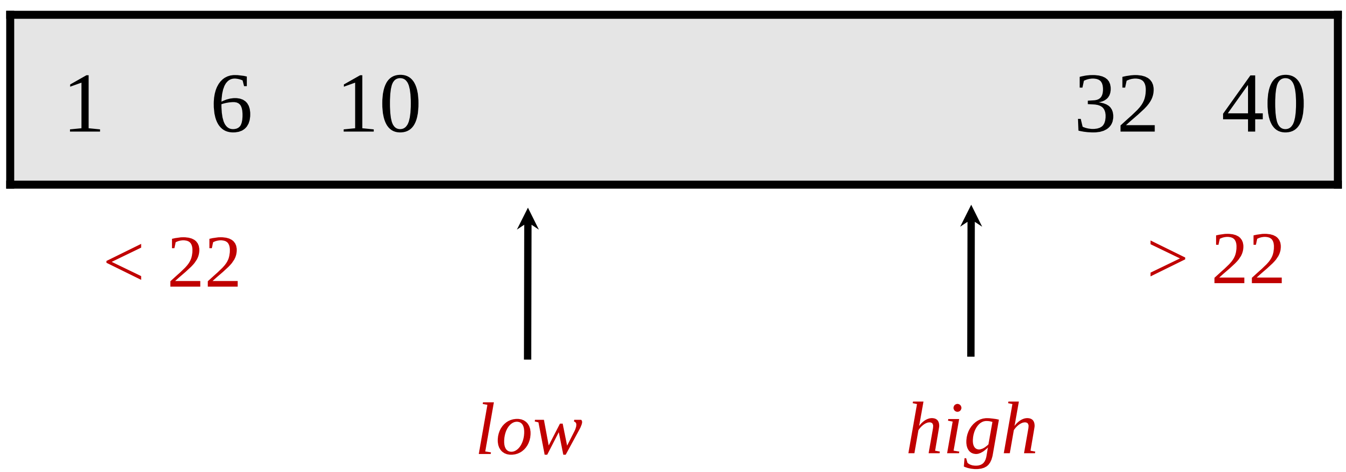
< 22   *low*   *high*   > 22

# Partitioning an Array

Example: partition around 22

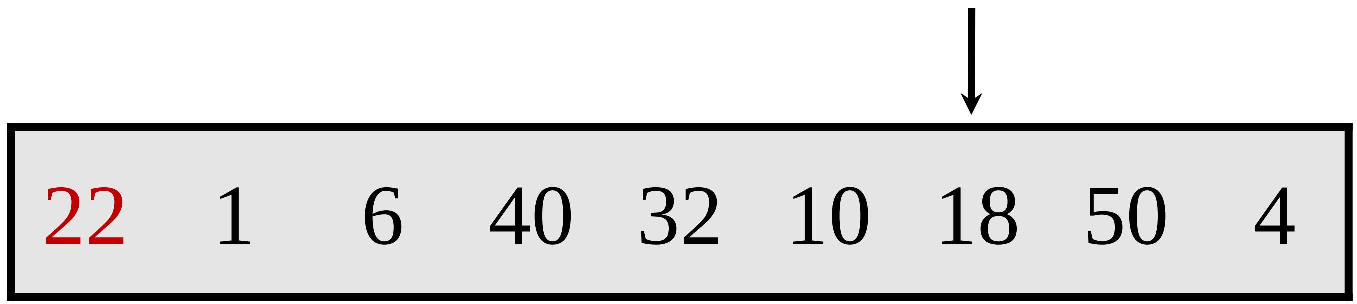



Output array:

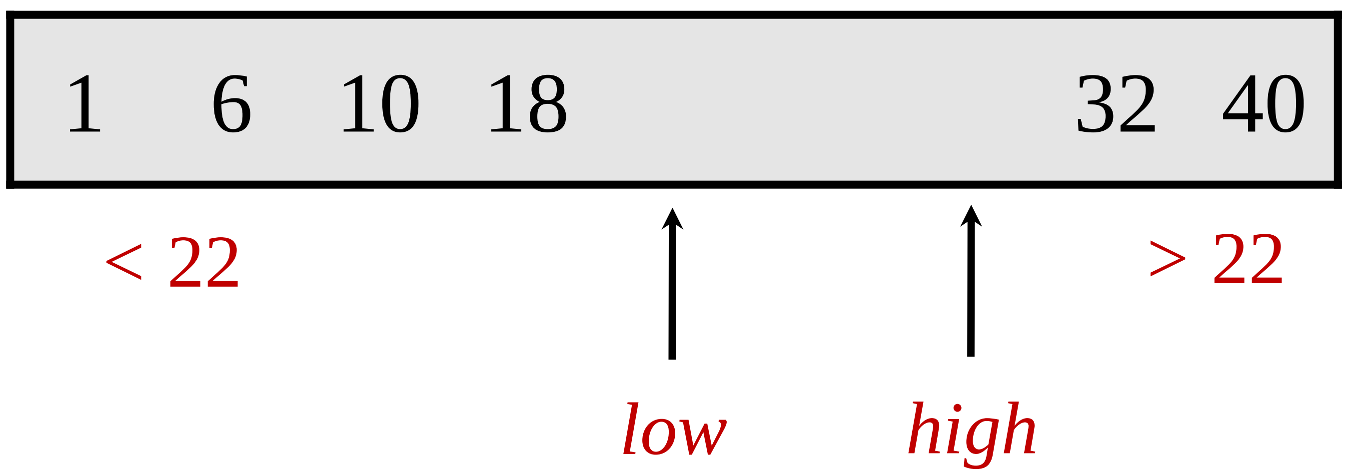


# Partitioning an Array

Example: partition around 22




Output array:



# Partitioning an Array

Example: partition around 22

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

Output array:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 10 | 18 | | | 50 | 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 | 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 |

$< 22$

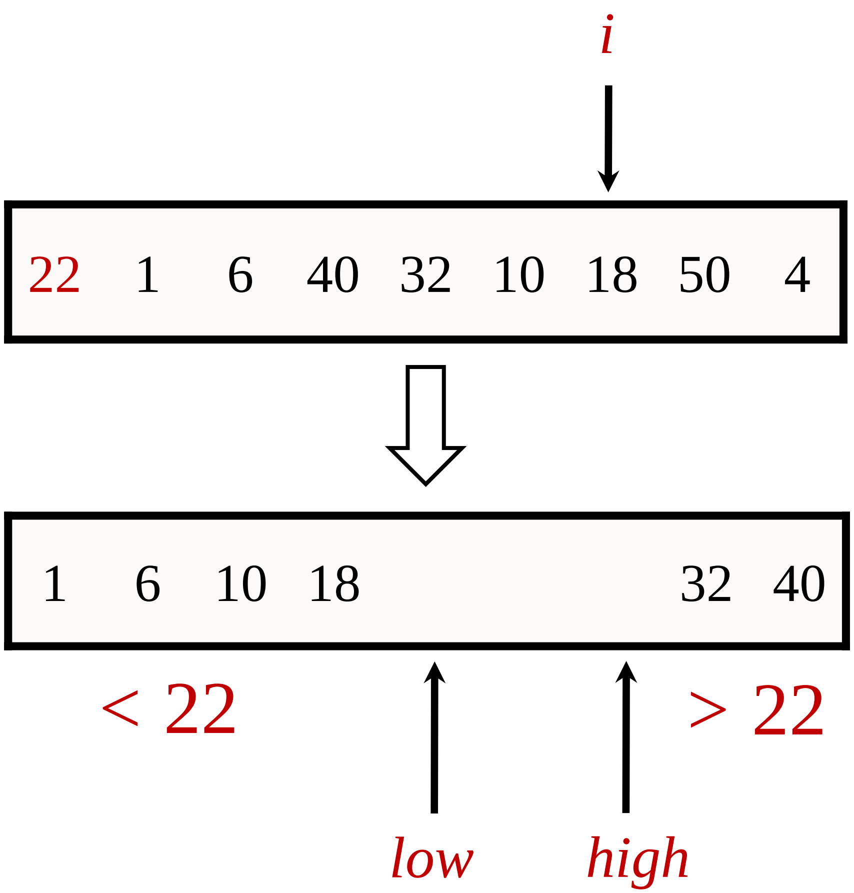
$> 22$

*high*

*low*

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

// Assume no duplicates



# Partition

**Claim**: array B is partitioned around the pivot

**Proof**:

Invariants:

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

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

Then: $B[i] = pivot$

By invariants, B is partitioned around the pivot.

# Partitioning an Array

Example:

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

What is the running time of partition?

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

# Partitioning an Array

Example:

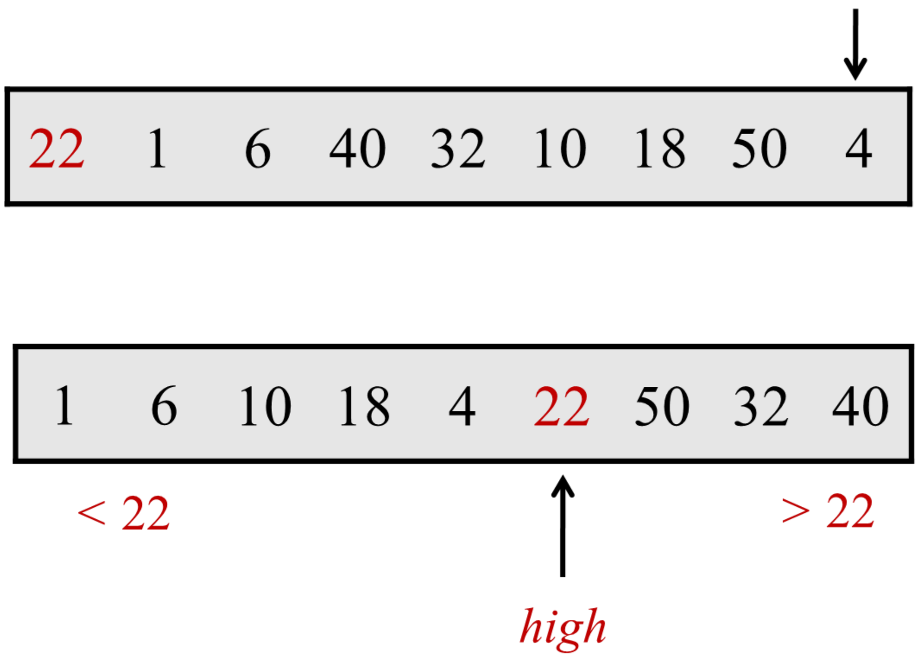
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 22 | 1 | 6 | 40 | 32 | 10 | 18 | 50 | 4 |

What is the running time of partition?

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

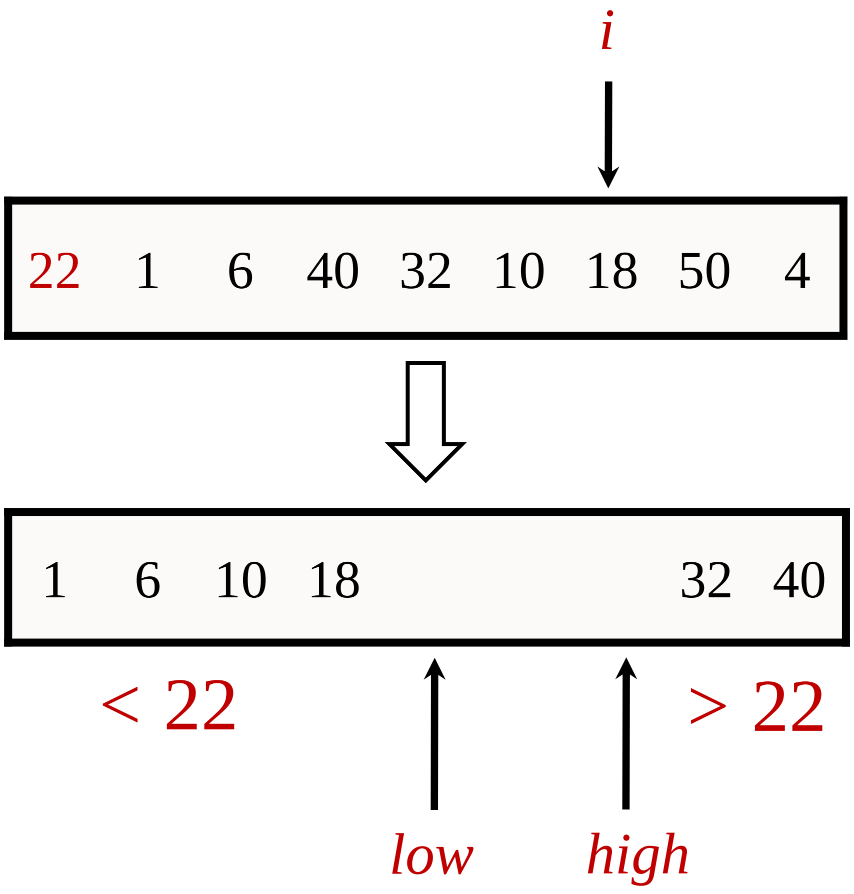
Any bugs?

Anything that can be improved?



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

**// Assume no duplicates**



# Partitioning an Array "in-place"

Example: partition around 22

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

*low*
< 22

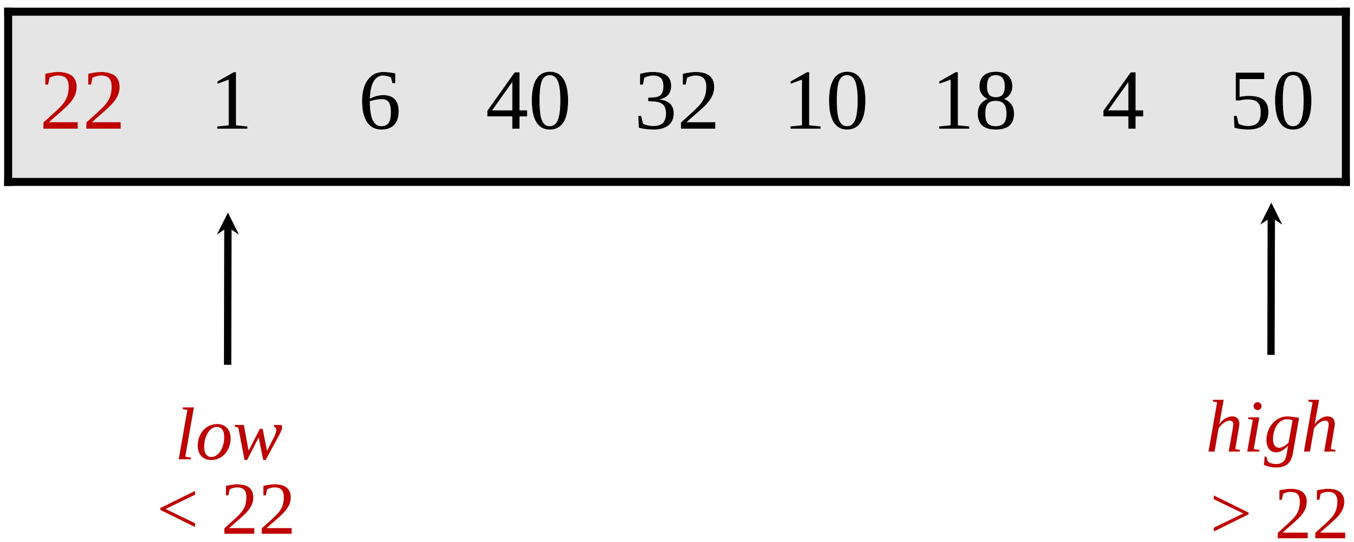
*high*
> 22

Move until it's bigger than the pivot

Move until it's less than the pivot

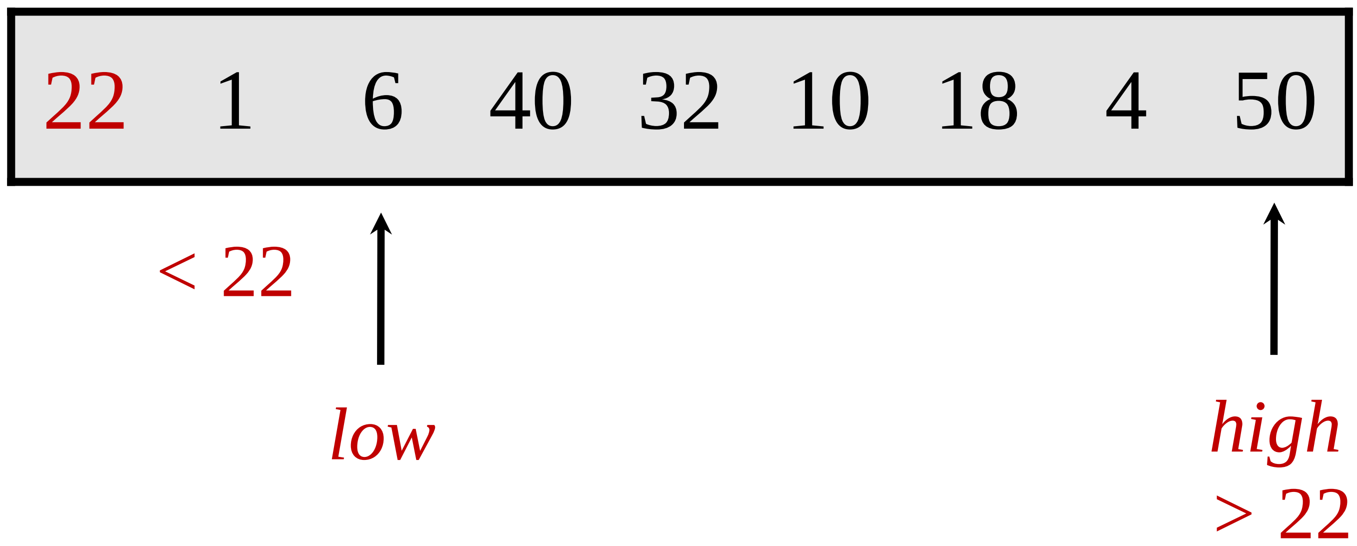
# Partitioning an Array

Example: partition around 22



# Partitioning an Array

Example: partition around 22



# Partitioning an Array

Example: partition around 22

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

< 22

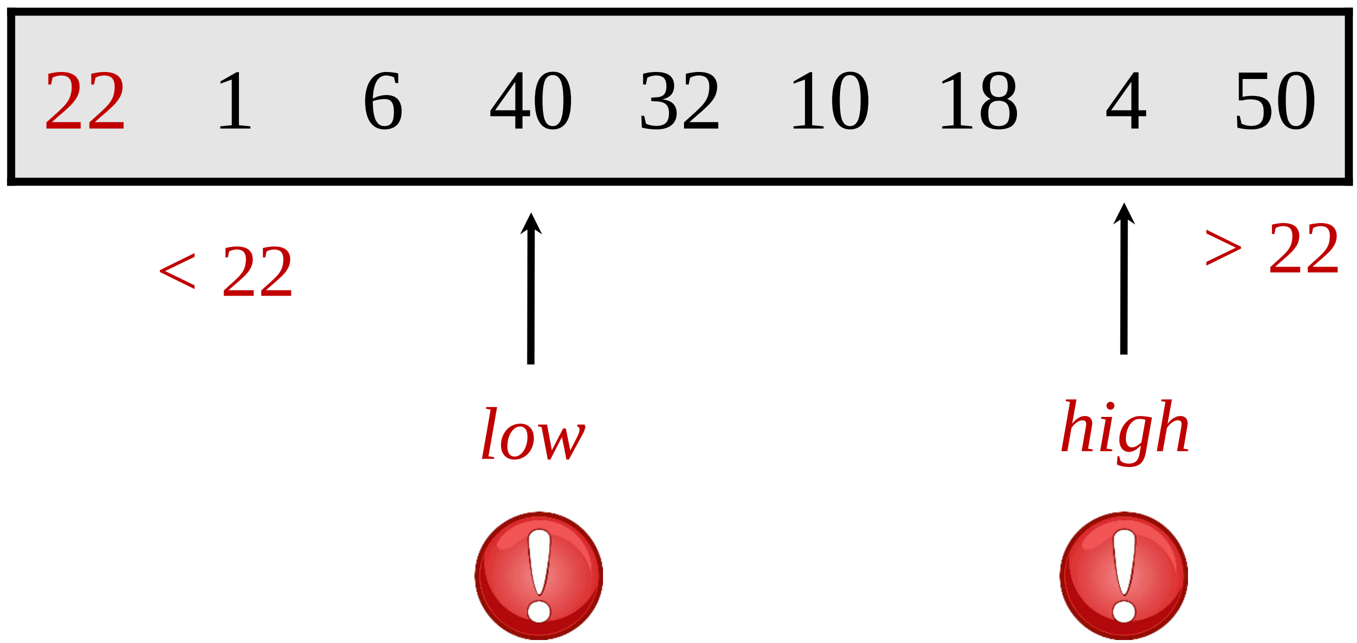
*low*

*high*

> 22

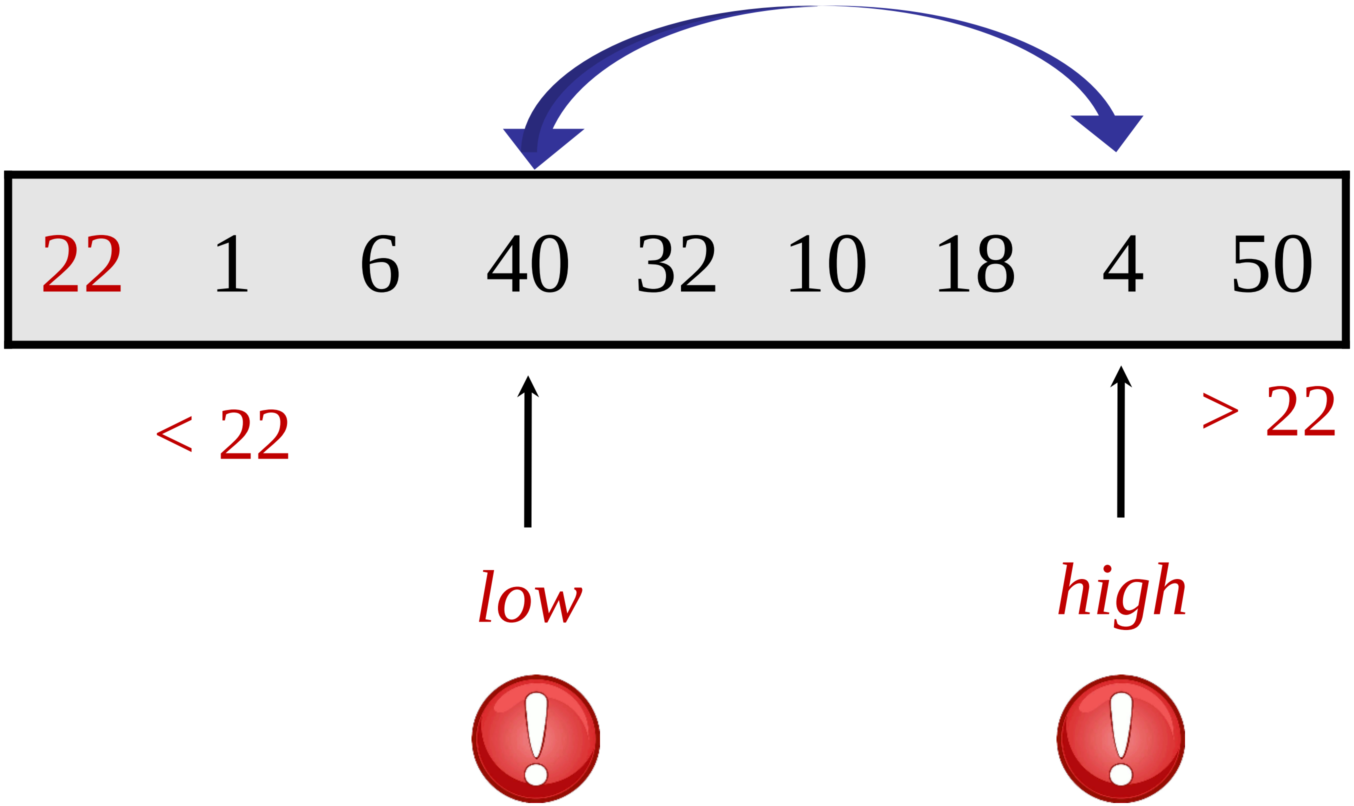
# Partitioning an Array

Example: partition around 22



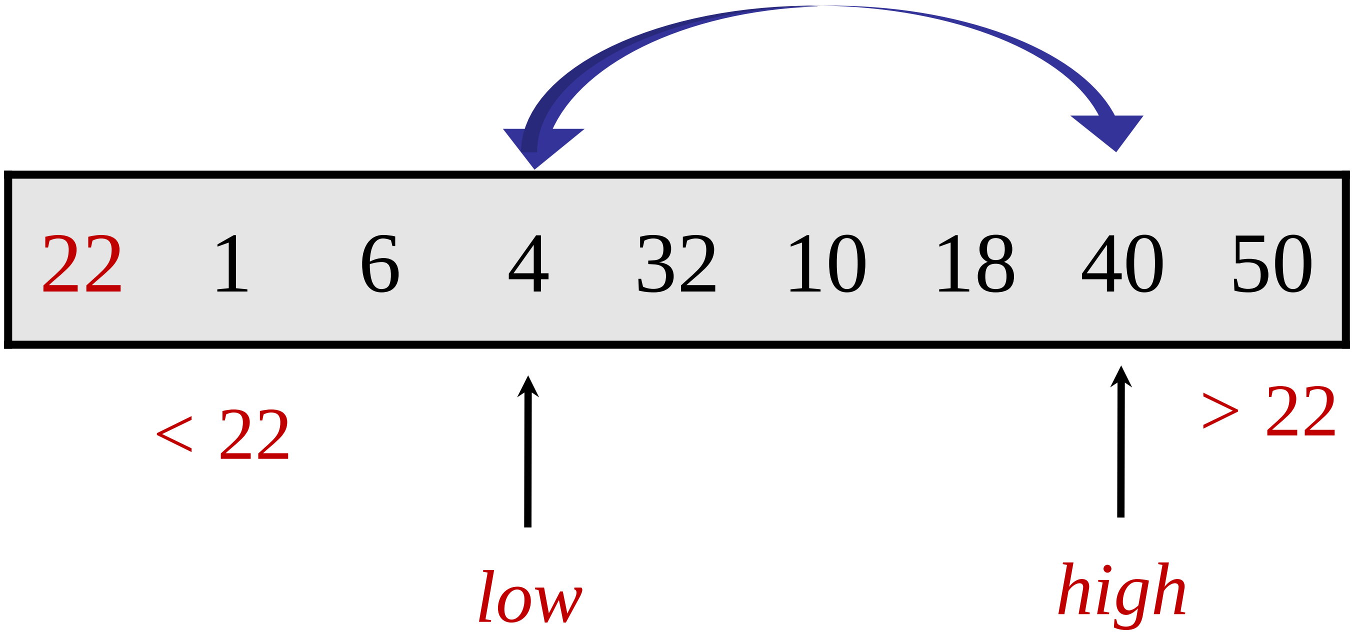
# Partitioning an Array

Example: partition around 22



# Partitioning an Array

Example: partition around 22



# Partitioning an Array

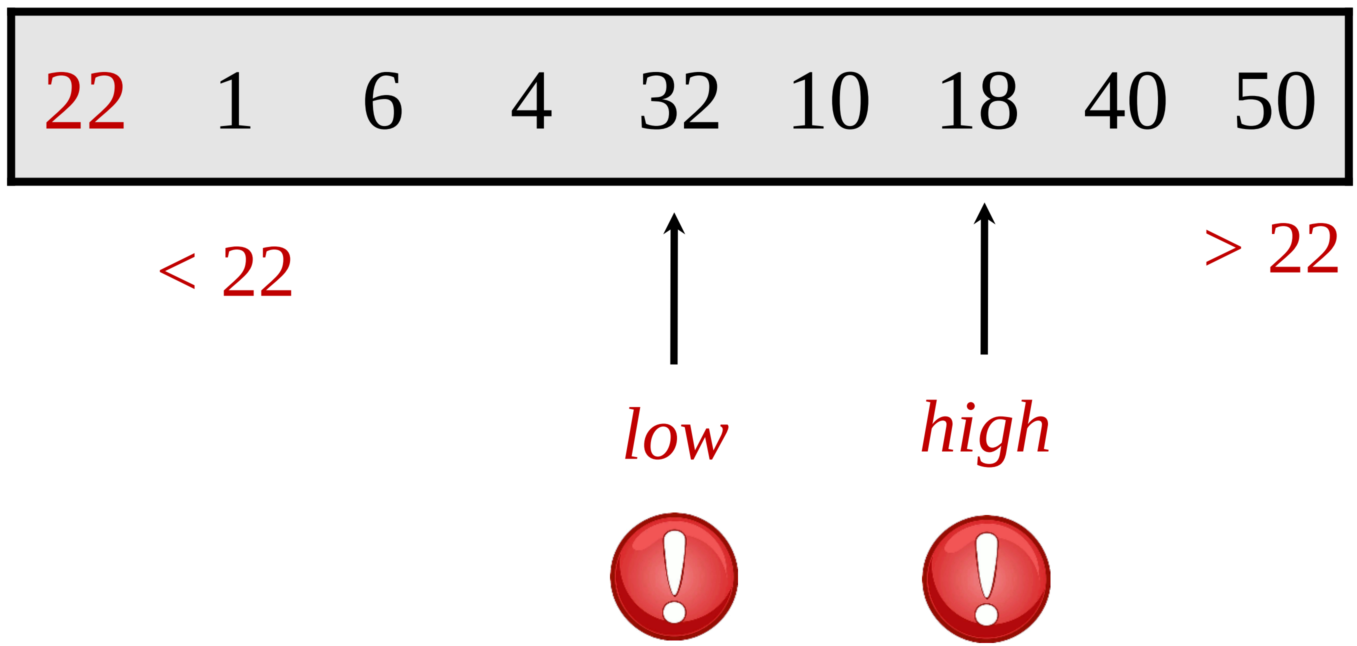
Example: partition around 22

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

< 22    ↑ *low*    ↑ *high*    > 22

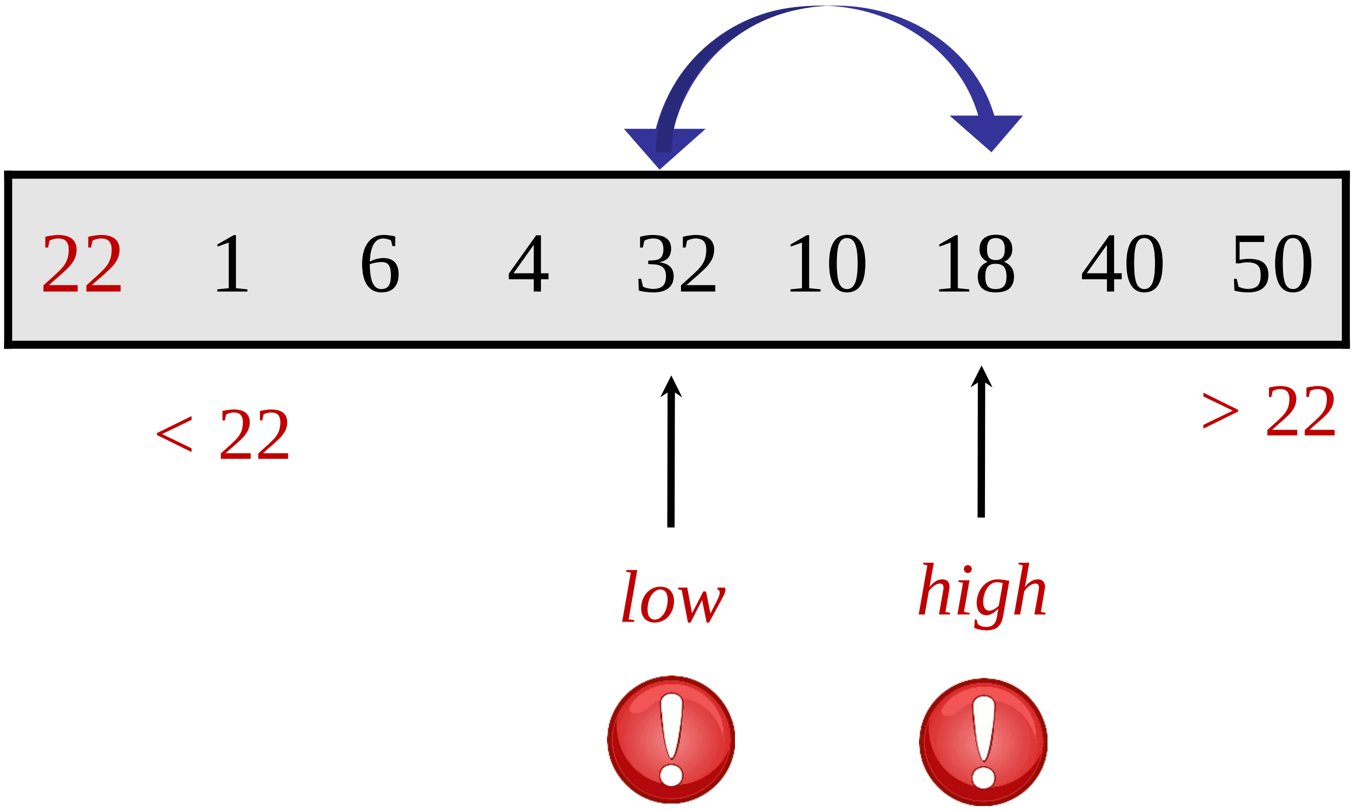
# Partitioning an Array

Example: partition around 22



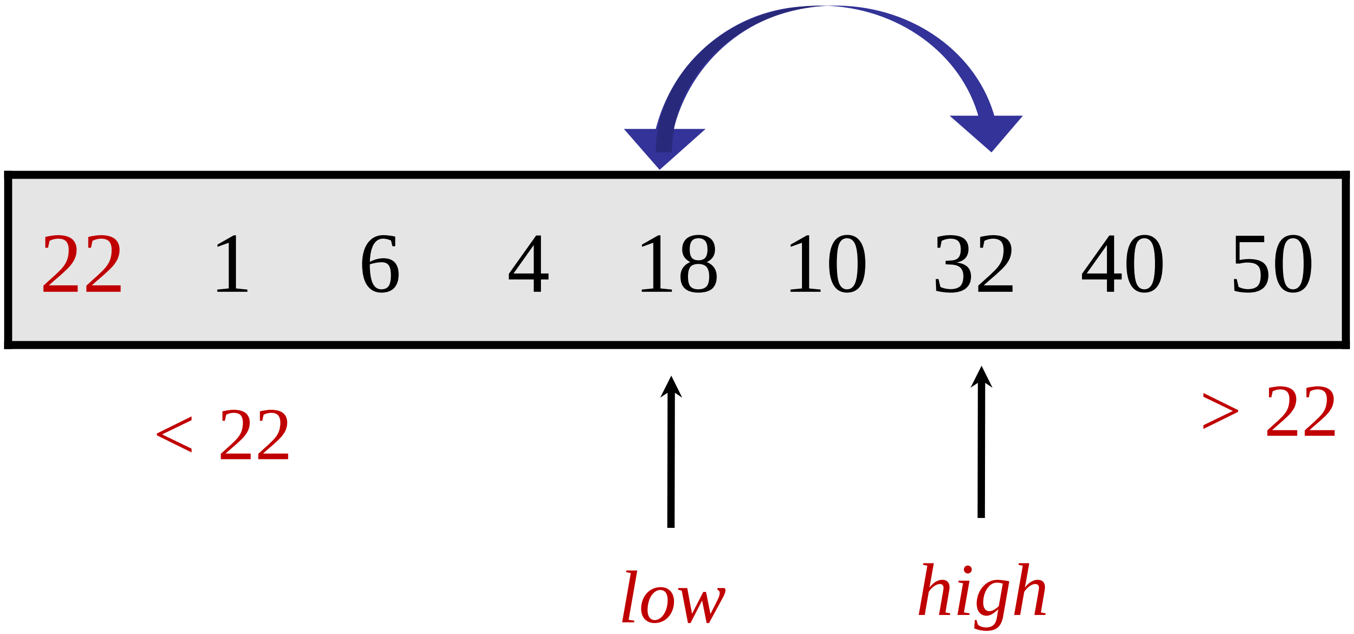
# Partitioning an Array

Example: partition around 22



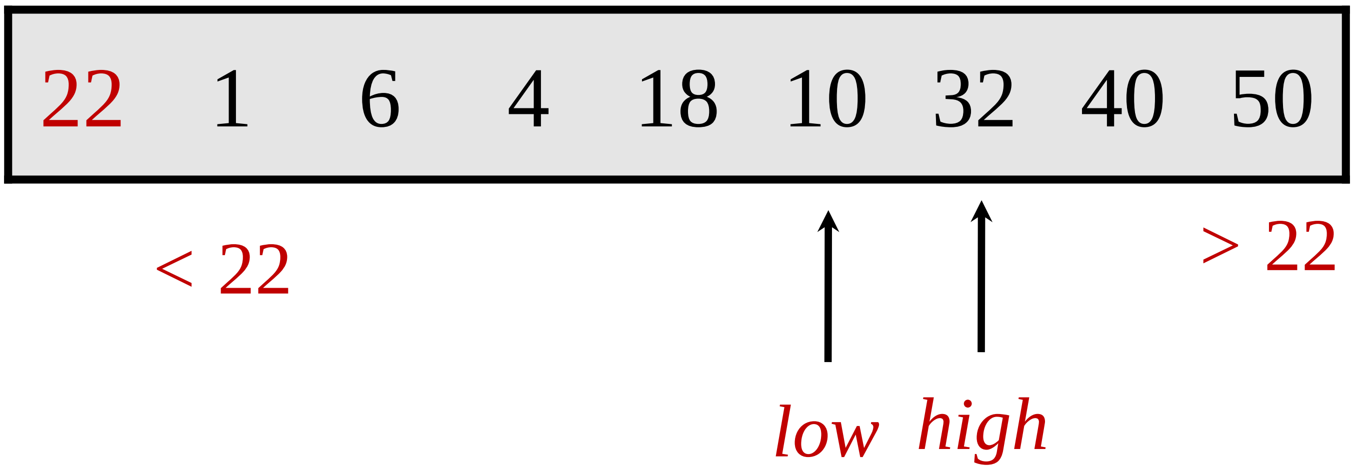
# Partitioning an Array

Example: partition around 22



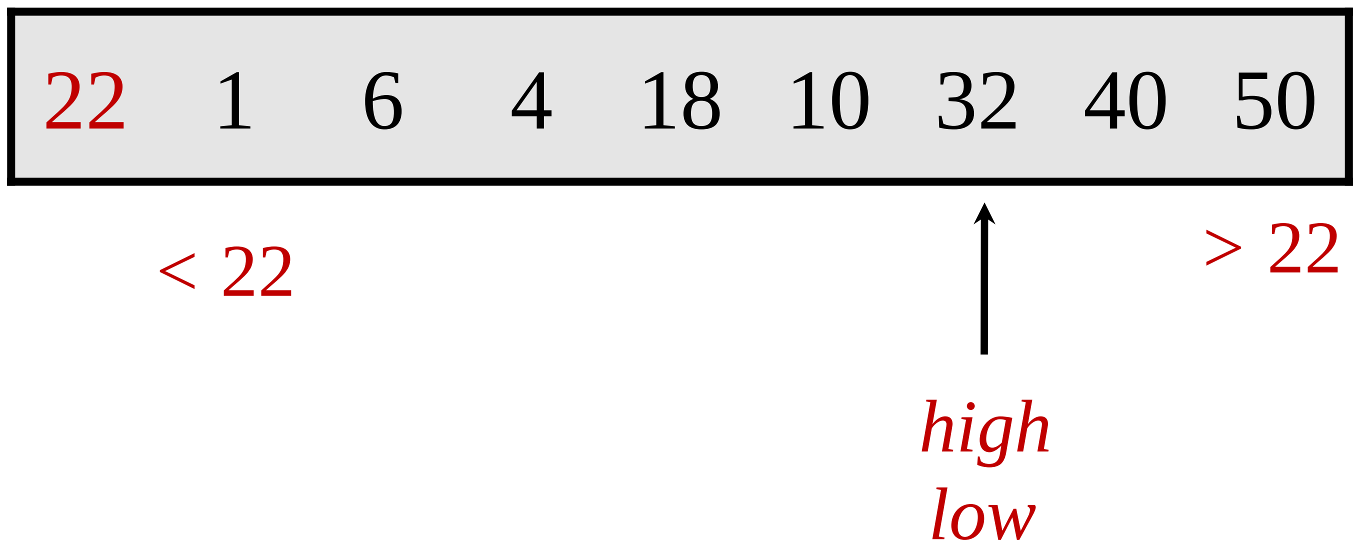
# Partitioning an Array

Example: partition around 22



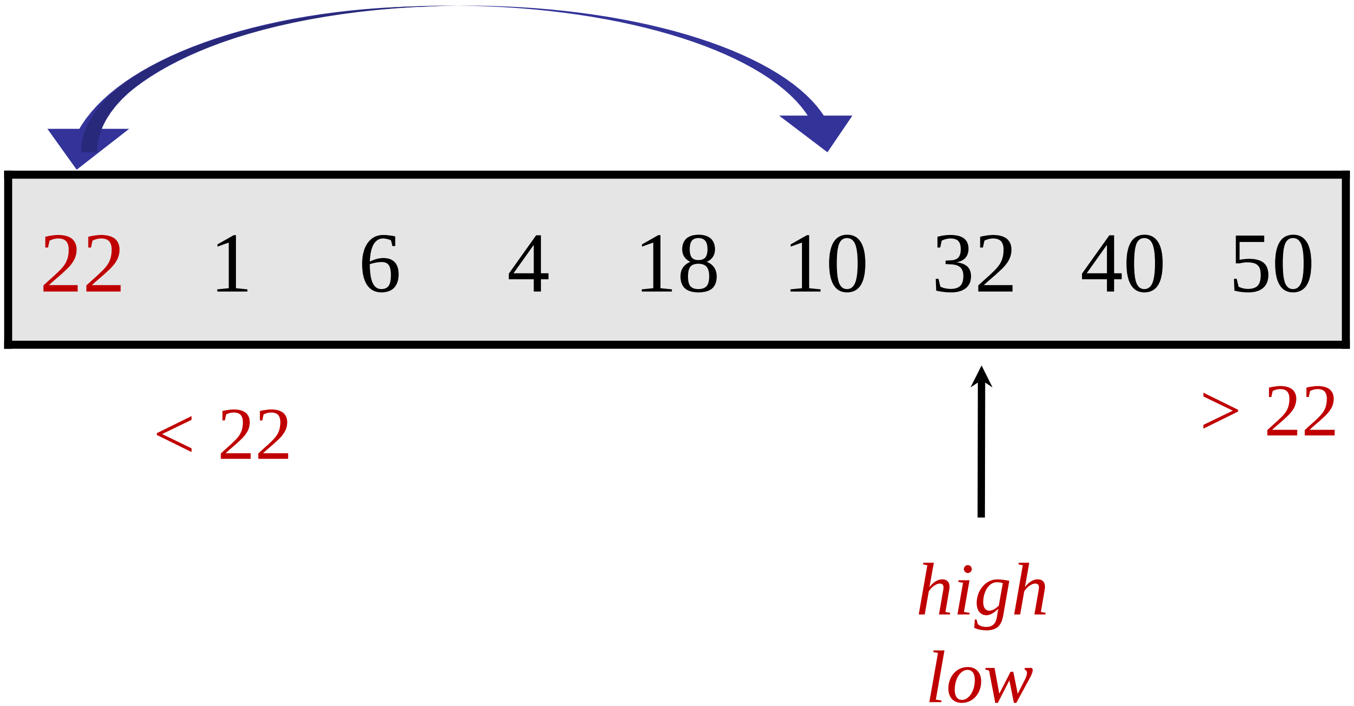
# Partitioning an Array

Example: partition around 22



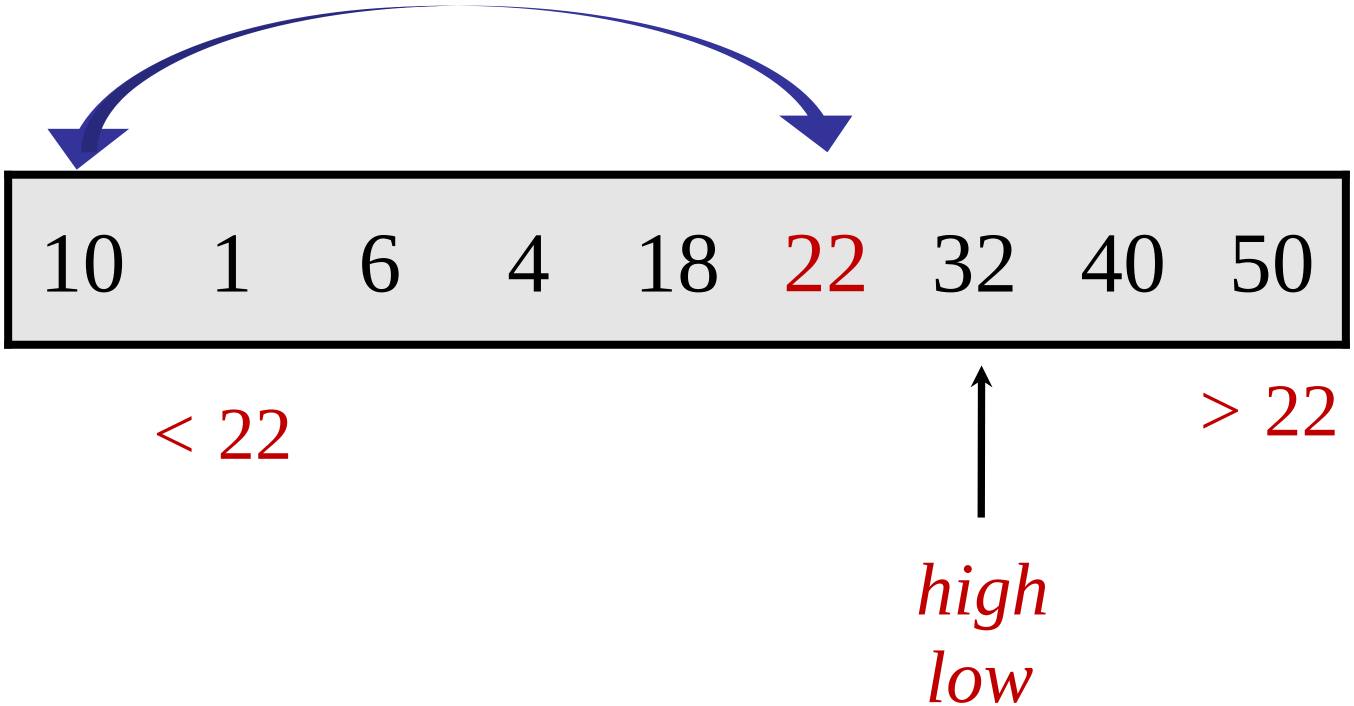
# Partitioning an Array

Example: partition around 22



# Partitioning an Array

Example: partition around 22



```
partition(A[1..n], n, pIndex)           // Assume no duplicates, n>1
    pivot = A[pIndex];                  // pIndex is the index of pivot
    swap(A[1], A[pIndex]);              // store pivot in A[1]
    low = 2;                            // start after pivot in A[1]
    high = n+1;                         // Define: A[n+1] = ∞
    while (low < high)
        while (A[low] <= pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high– – ;
        if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low–1]);
    return  low–1;
```

Pseudocode

vs.

Real Code

QuickSort is notorious for off-by-one errors…

# Partition

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

Proof:

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

# Partition

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

Proof: During loop:

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

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

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

- When exit loop decrementing high:

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

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

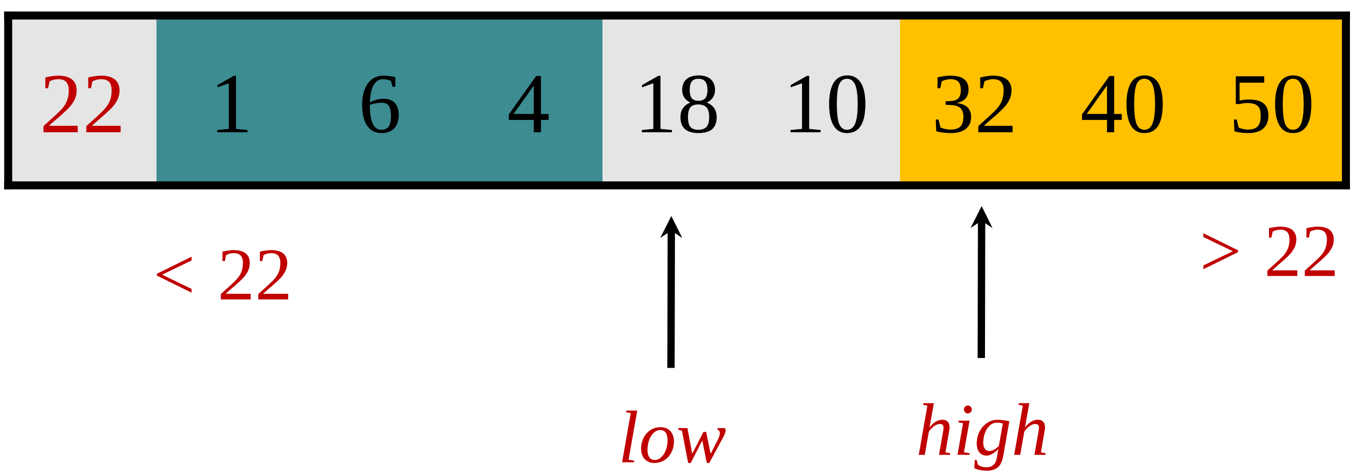
```
partition(A[1..n], n, pIndex)          // Assume no duplicates, n>1
    pivot = A[pIndex];                 // pIndex is the index of pivot
    swap(A[1], A[pIndex]);             // store pivot in A[1]
    low = 2;                           // start after pivot in A[1]
    high = n+1;                        // Define: A[n+1] = ∞
    while (low < high)
        while (A[low] < pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high– – ;
        if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low–1]);
    return  low–1;
```

# Partition

Invariant: At the end of every loop iteration:

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

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

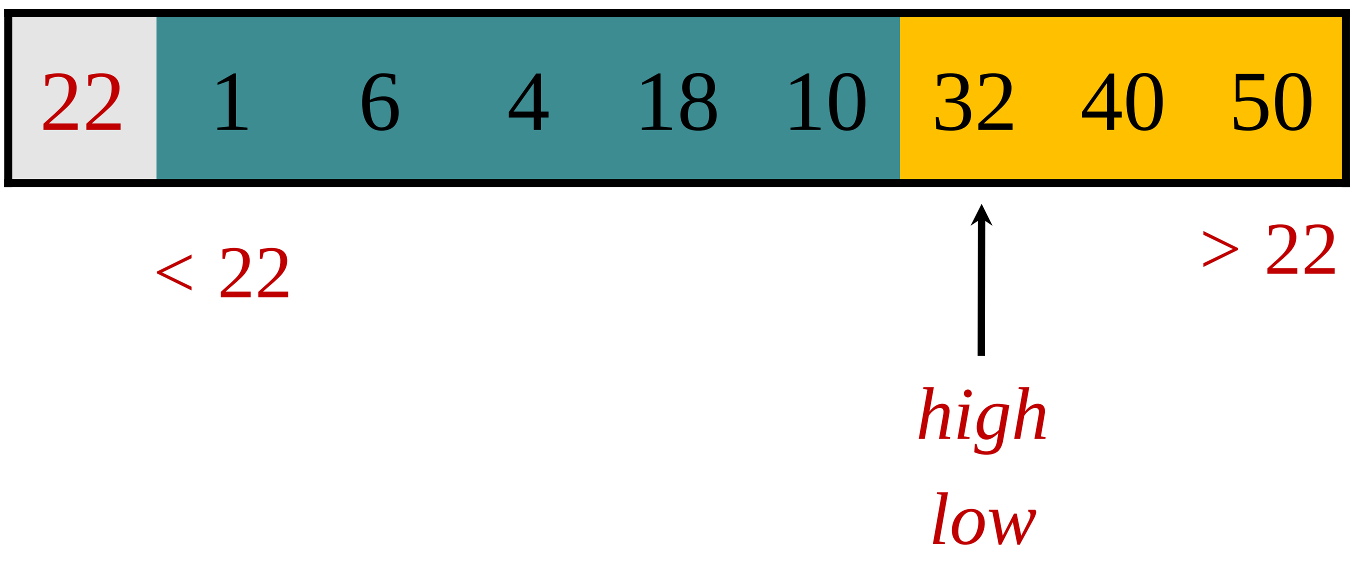


# Partition

Invariant: At the end of every loop iteration:

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

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



# Partition

Claim: At the end of every loop iteration:

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

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


Claim: Array $A$ is partitioned around the pivot

```
partition(A[1..n], n, pIndex)        // Assume no duplicates, n>1
    pivot = A[pIndex];               // pIndex is the index of pivot
    swap(A[1], A[pIndex]);           // store pivot in A[1]
    low = 2;                         // start after pivot in A[1]
    high = n+1;                      // Define: A[n+1] = ∞
    while (low < high)
        while (A[low] < pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high– – ;
        if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low–1]);
    return  low–1;
```

```
partition(A[1..n], n, pIndex)
    pivot = A[pIndex];
    swap(A[1], A[pIndex]);
    low = 2;
    high = n+1;
    while (low < high)
        while (A[low] < pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high– – ;
        if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low–1]);
    return  low–1;
```

Running time:

O(n)

# QuickSort

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

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

**else**

Choose pivot index *pIndex*.

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

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

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

# Sorting, continued

QuickSort

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

# QuickSort

What happens if there are duplicates?

# Duplicates

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

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

**else**

Choose pivot index *pIndex*.

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

$x$ = QuickSort($A$[1..$p$–1], $p$–1)

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

| | | | | |
|---|---|---|---|---|
| $< x$ | $x$ | $x$ | $x$ | $> x$ |

# Quicksort

Example:

6 6 6 6 6 6

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |

# Quicksort

Example:

| | | | | | |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |

# Quicksort

What is the running time on the all 6's array?

| | | | | | |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |

# Quicksort

Example:

Running time:

$O(n^2)$

| | | | | | |
|---|---|---|---|---|---|
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |
| 6 | 6 | 6 | 6 | 6 | 6 |

```
partition(A[1..n], n, pIndex)          // Assume no duplicates,  n>1
    pivot = A[pIndex];                 // pIndex is the index of pivot
    swap(A[1], A[pIndex]);             // store pivot in A[1]
    low = 2;                           // start after pivot in A[1]
    high = n+1;                        // Define: A[n+1] = ∞
    while (low < high)
        while (A[low] < pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high– – ;
        if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low–1]);
    return  low–1;
```

# Duplicates

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

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

**else**

Choose pivot index *pIndex*.

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

$x$ = QuickSort($A$[1..$p$–1], $p$–1)

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

# Duplicates

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

    **if** $(n{=}{=}1)$ **then** return;

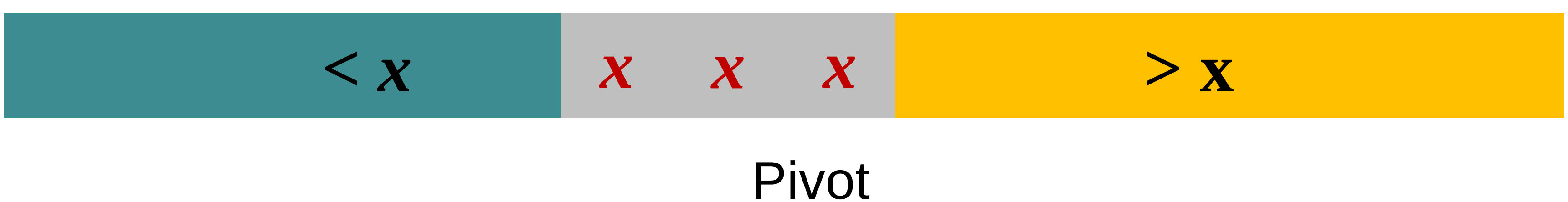
    **else**

        Choose pivot index *pIndex*.

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

        $x$ = QuickSort($A[1..p\text{–}1]$, $p\text{–}1$)

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



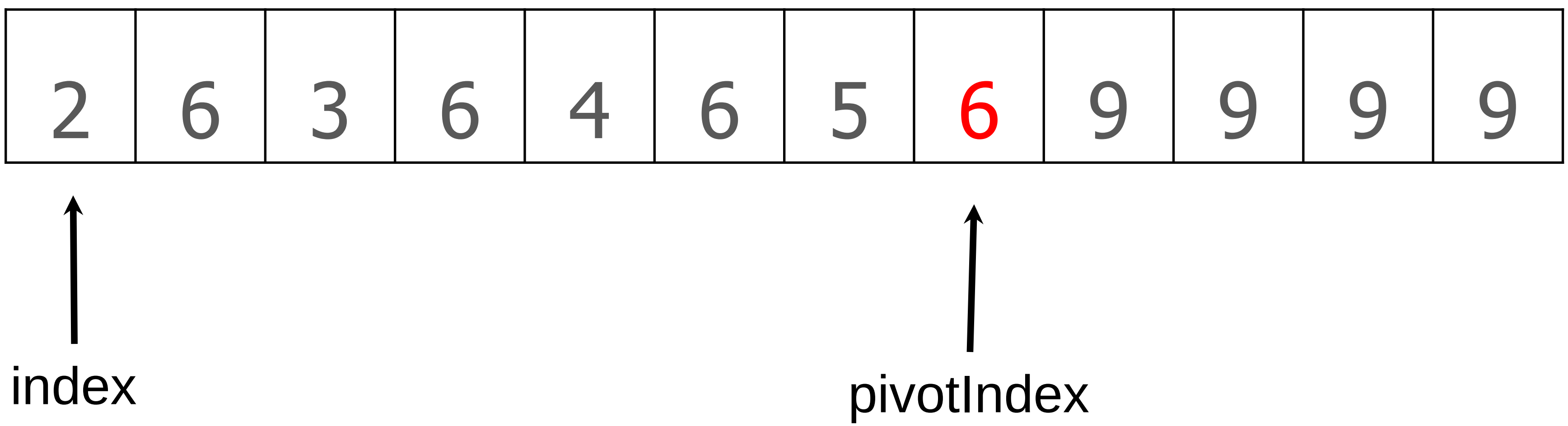
# Duplicates

3-Way Partitioning

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

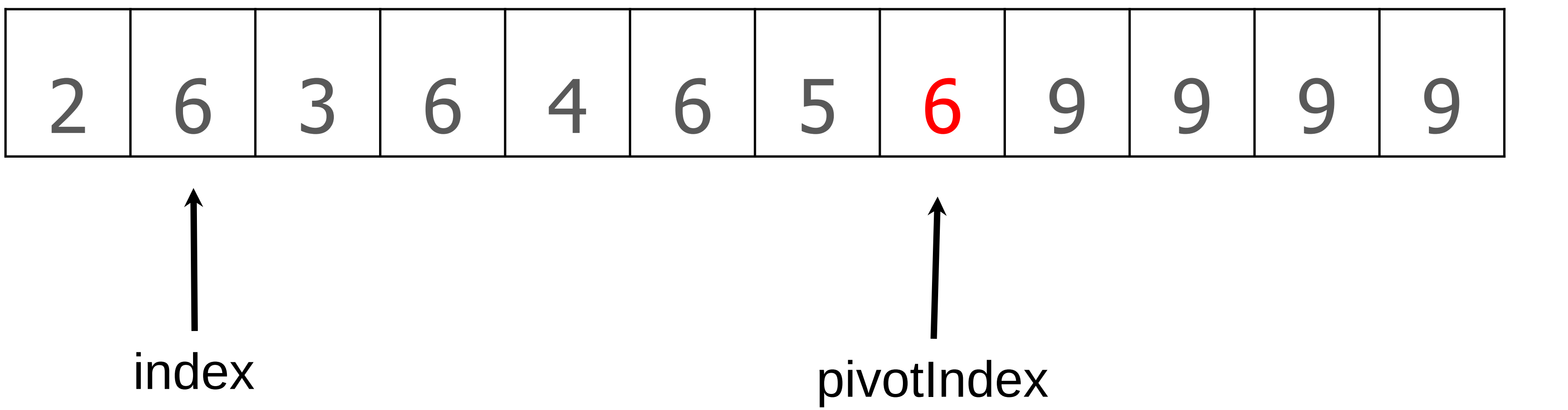
# Pack Duplicates

Example:



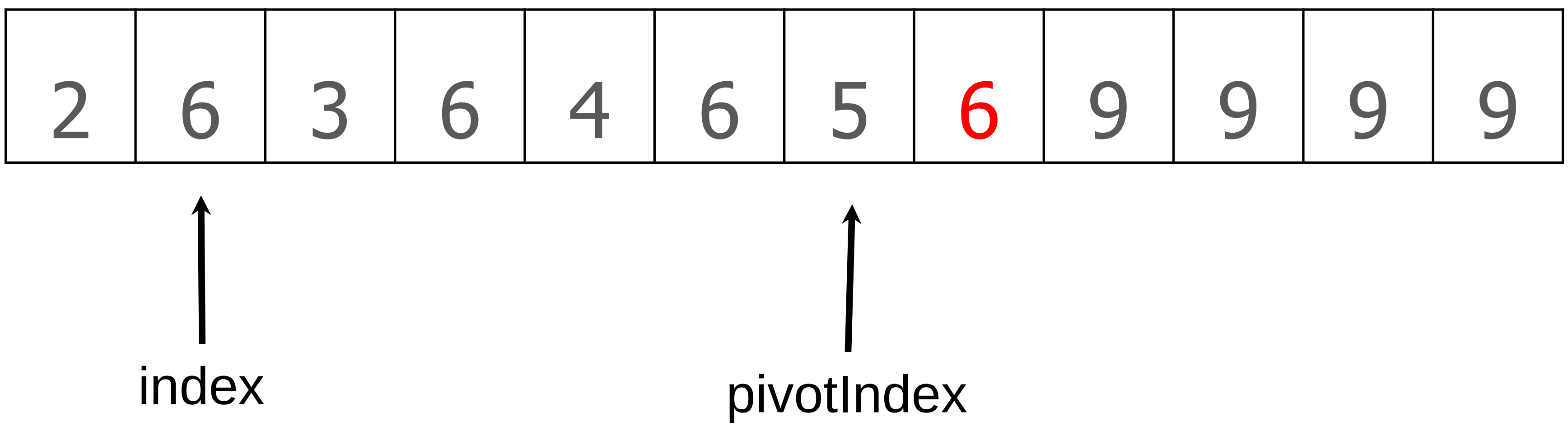
# Pack Duplicates

Example:



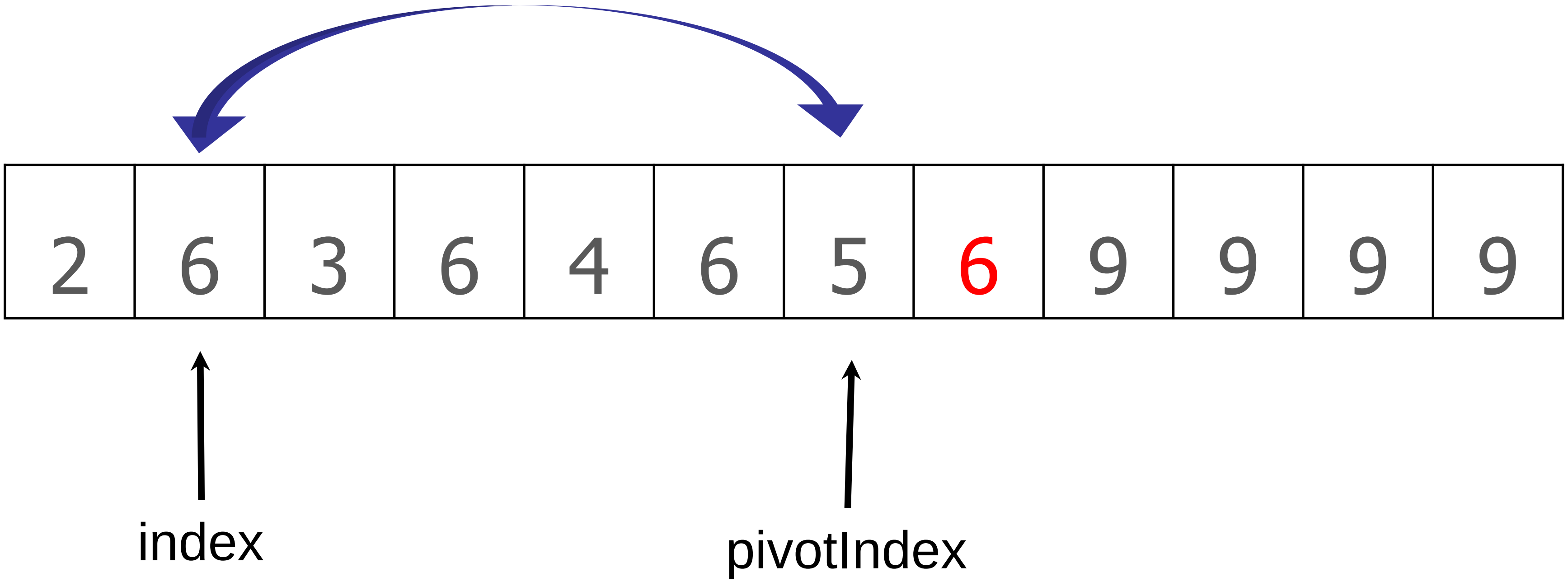
# Pack Duplicates

Example:



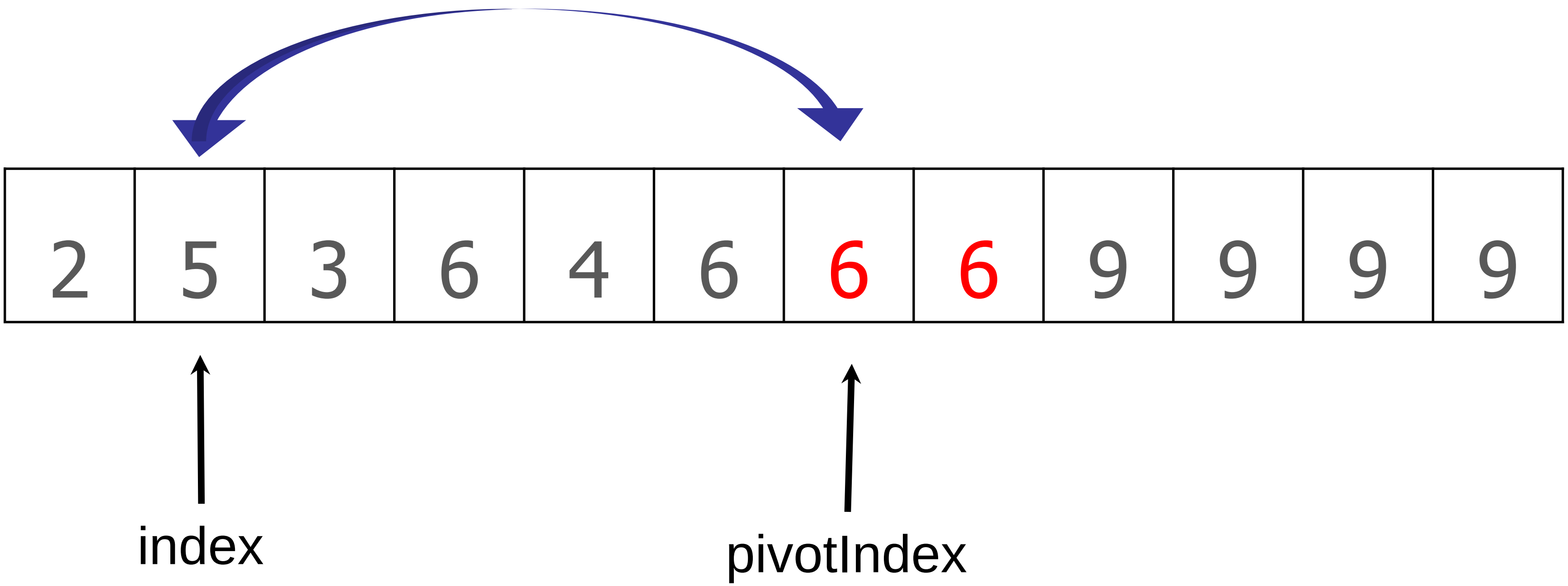
# Pack Duplicates

Example:



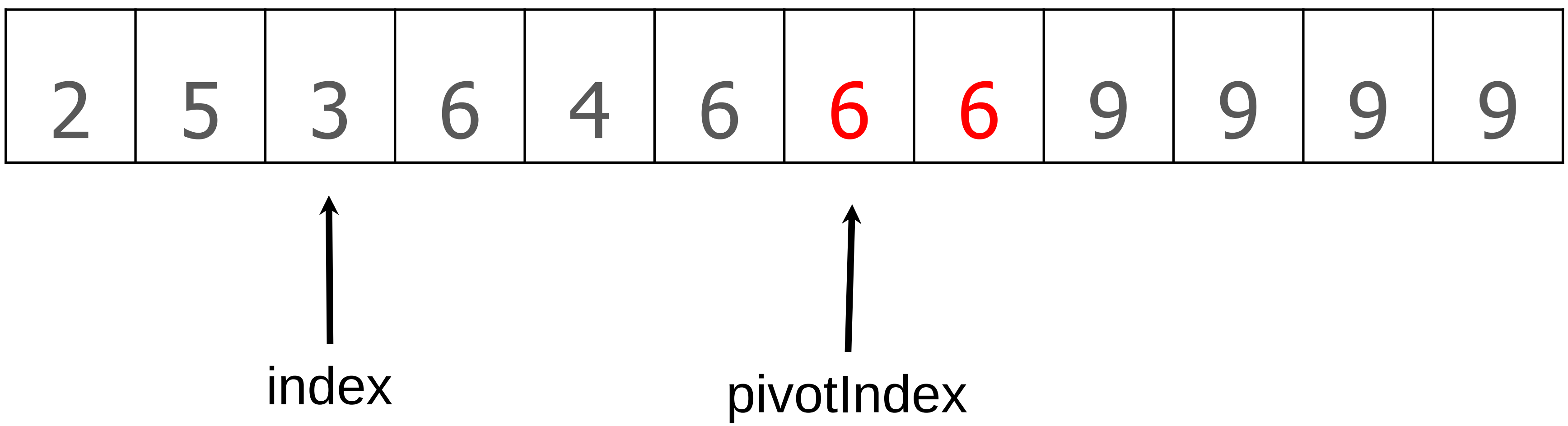
# Pack Duplicates

Example:



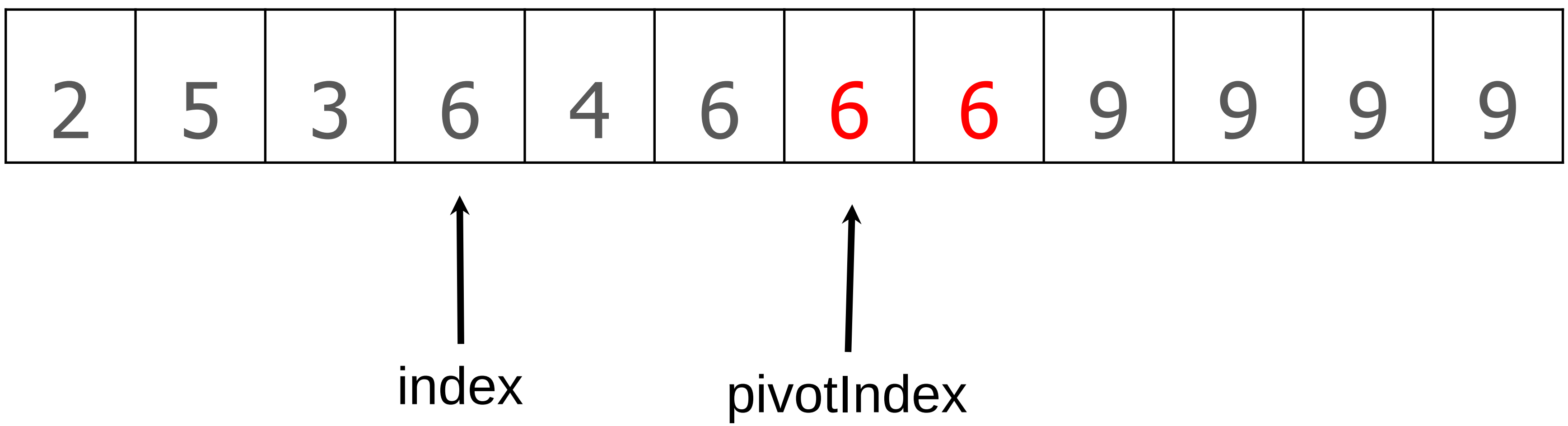
# Pack Duplicates

Example:



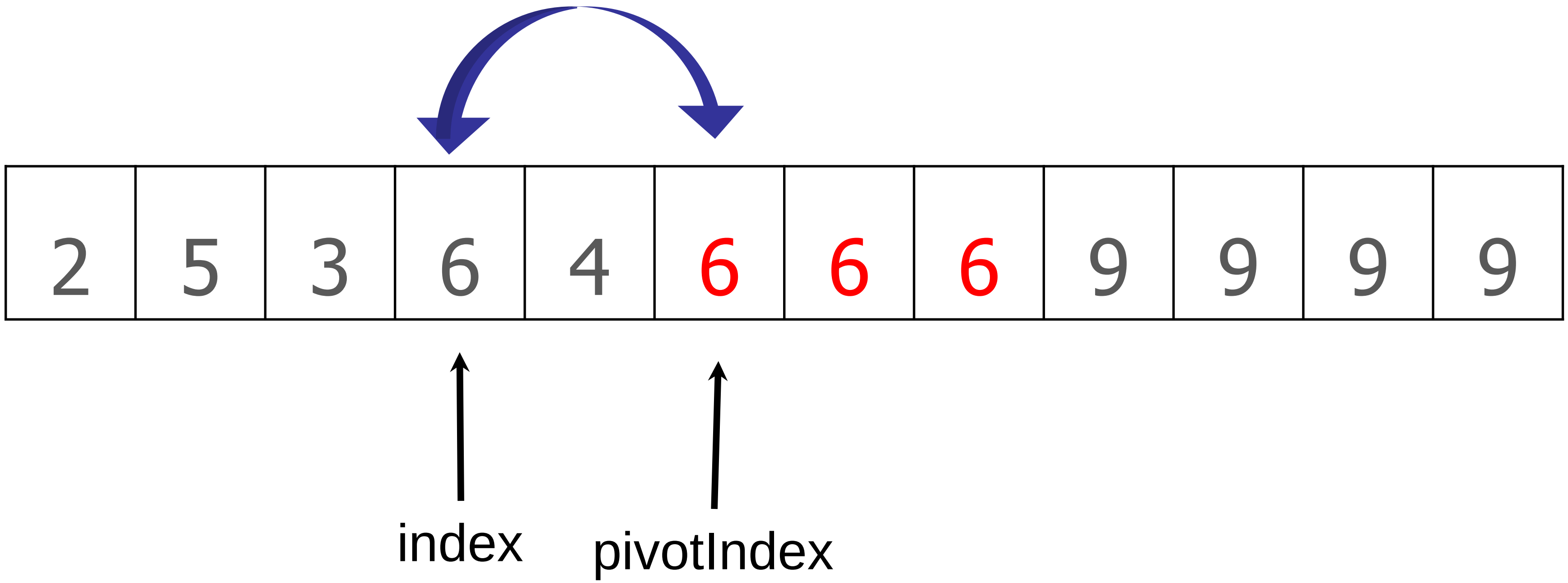
# Pack Duplicates

Example:



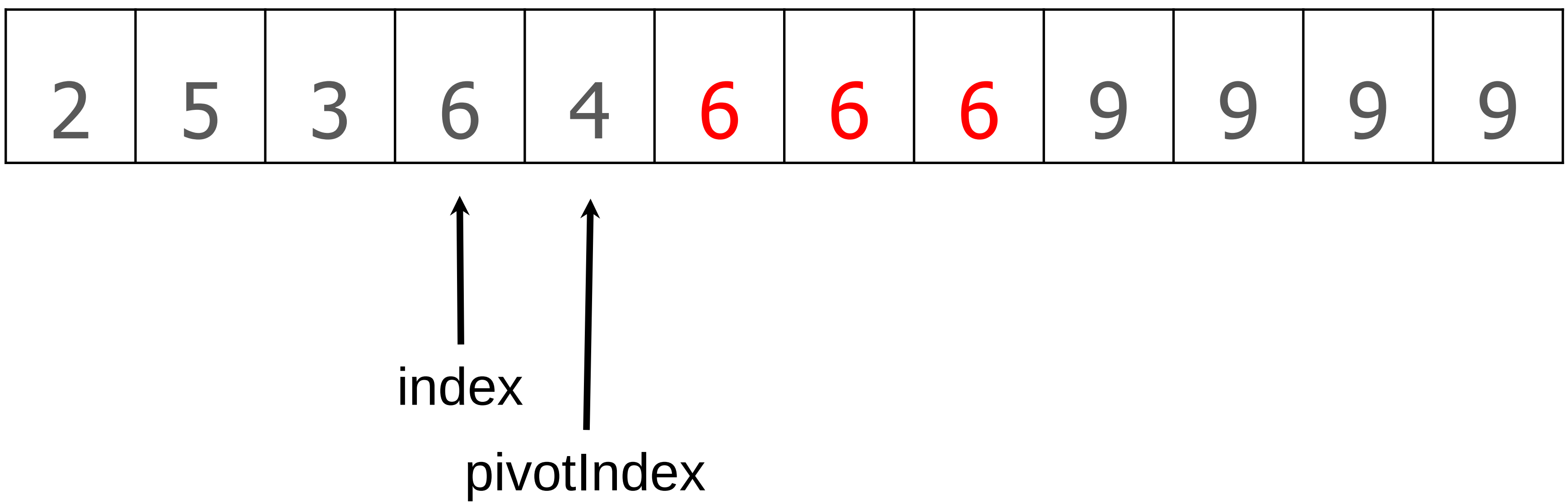
# Pack Duplicates

Example:



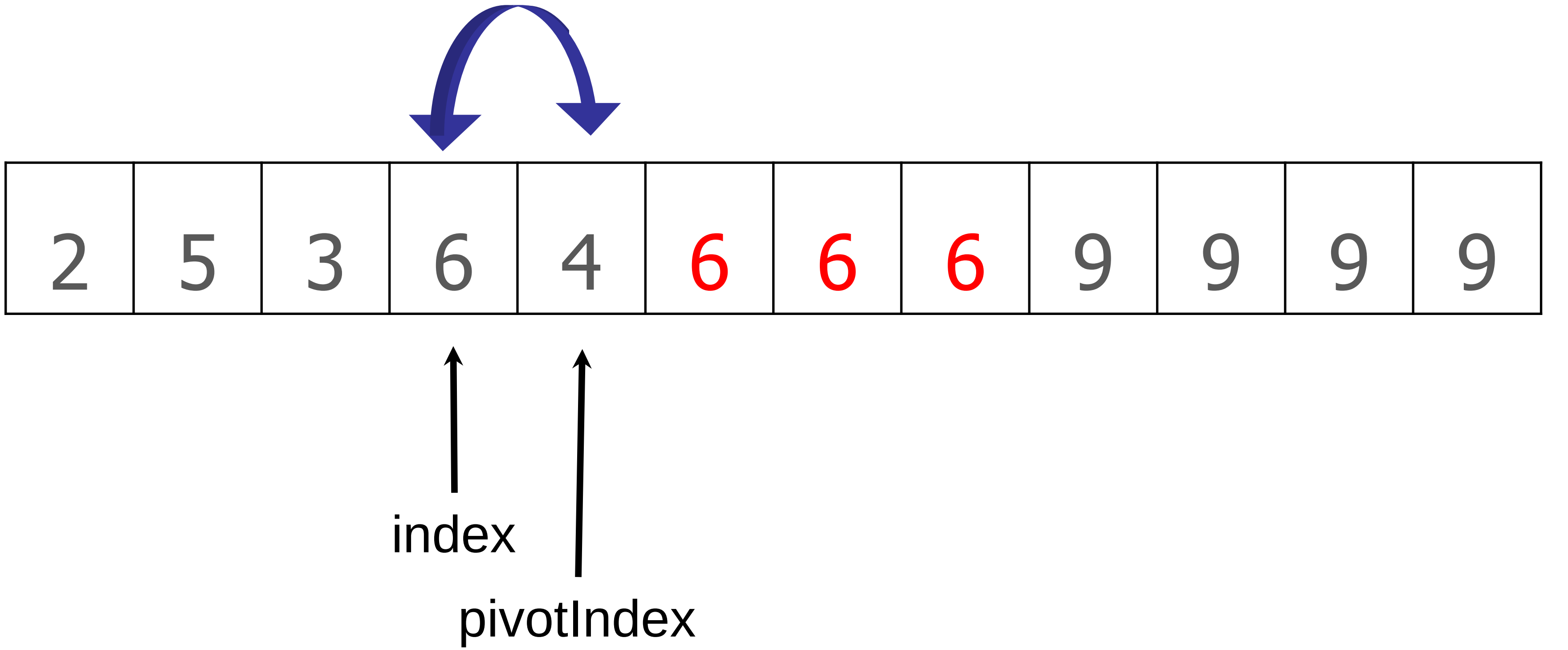
# Pack Duplicates

Example:



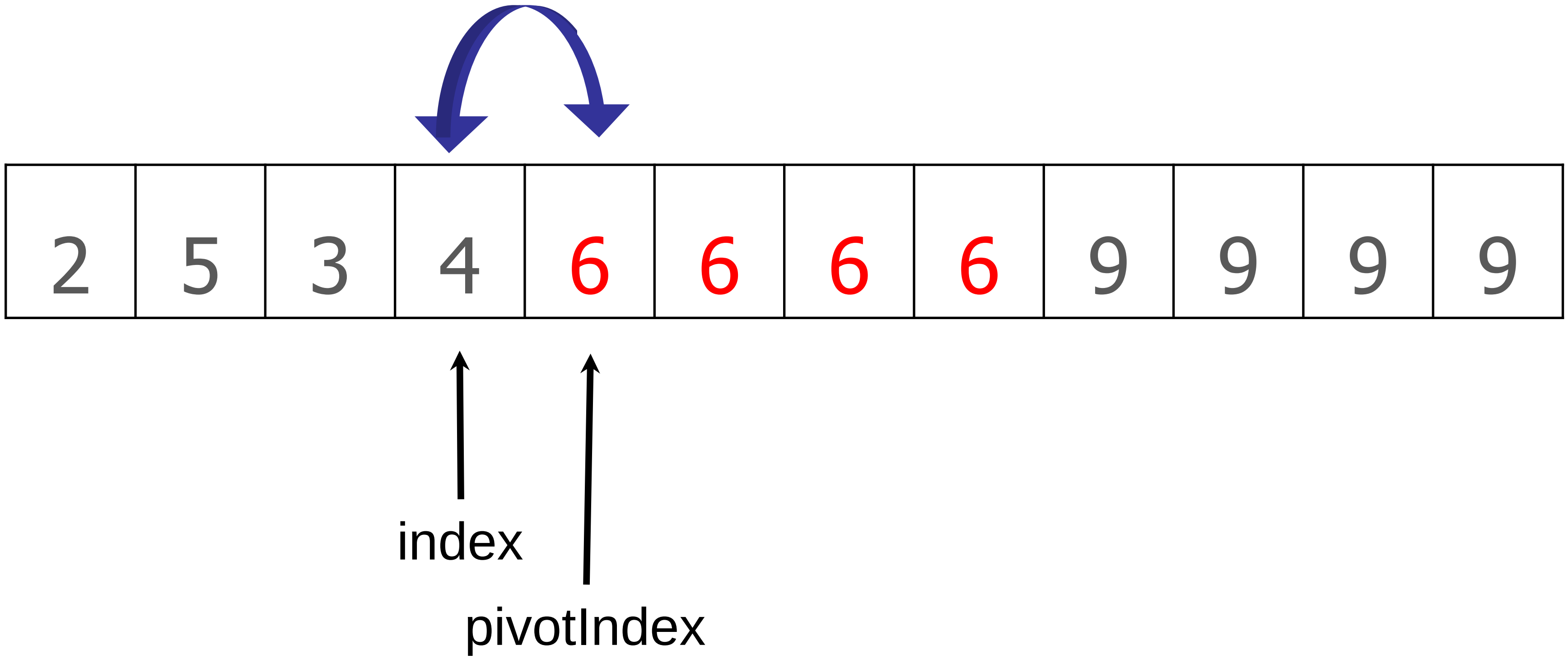
# Pack Duplicates

Example:



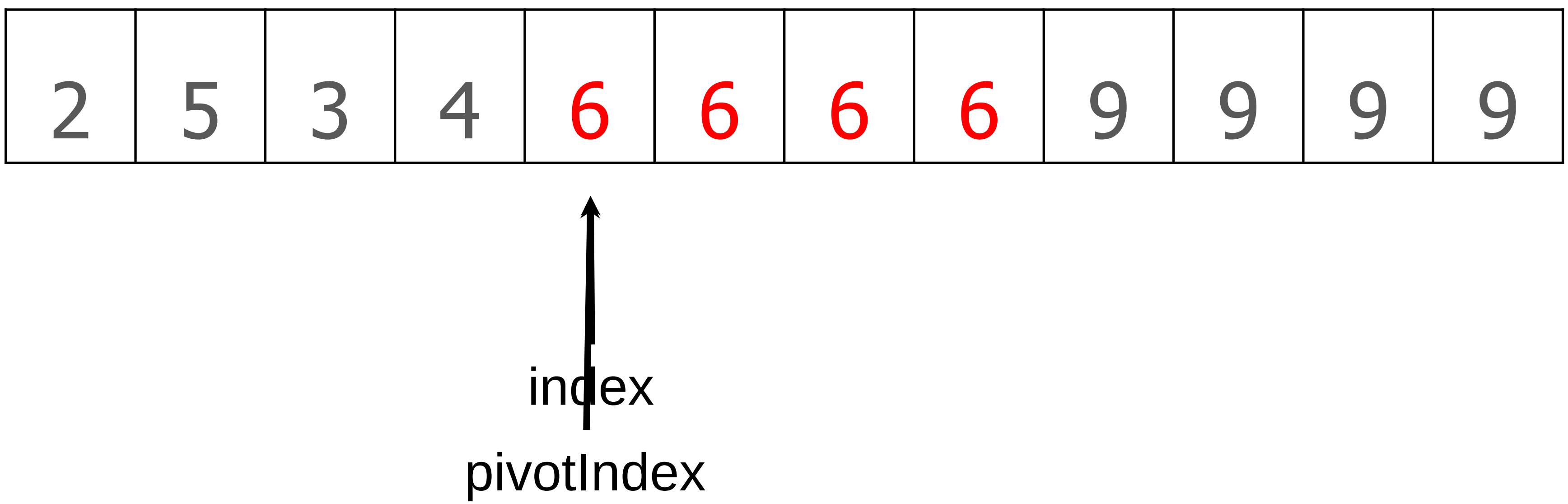
# Pack Duplicates

Example:



# Pack Duplicates

Example:



# Duplicates

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

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

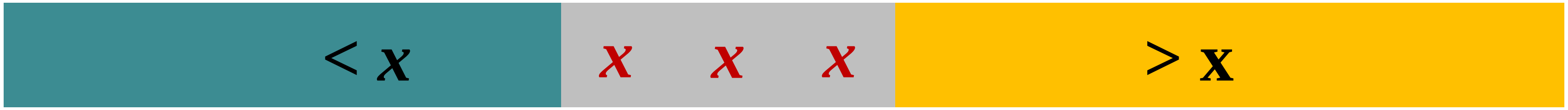
    **else**

        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], $n$–$p$)



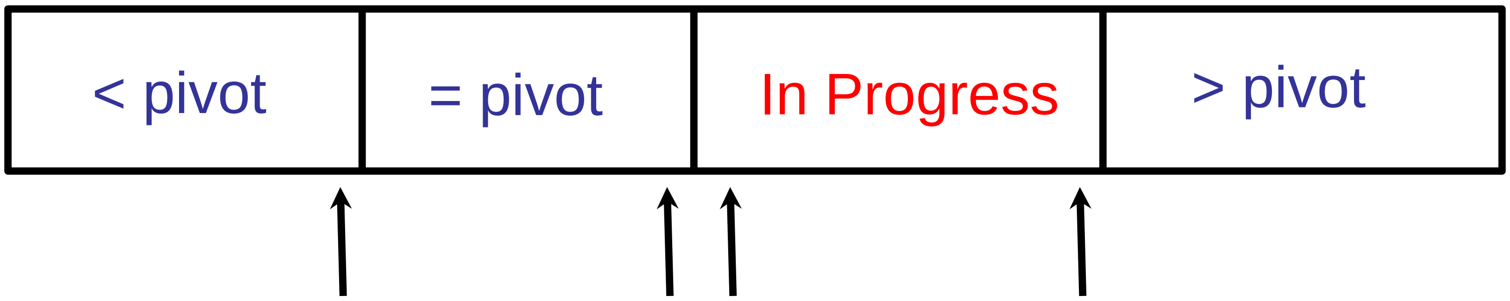
# Duplicates

3-Way Partitioning

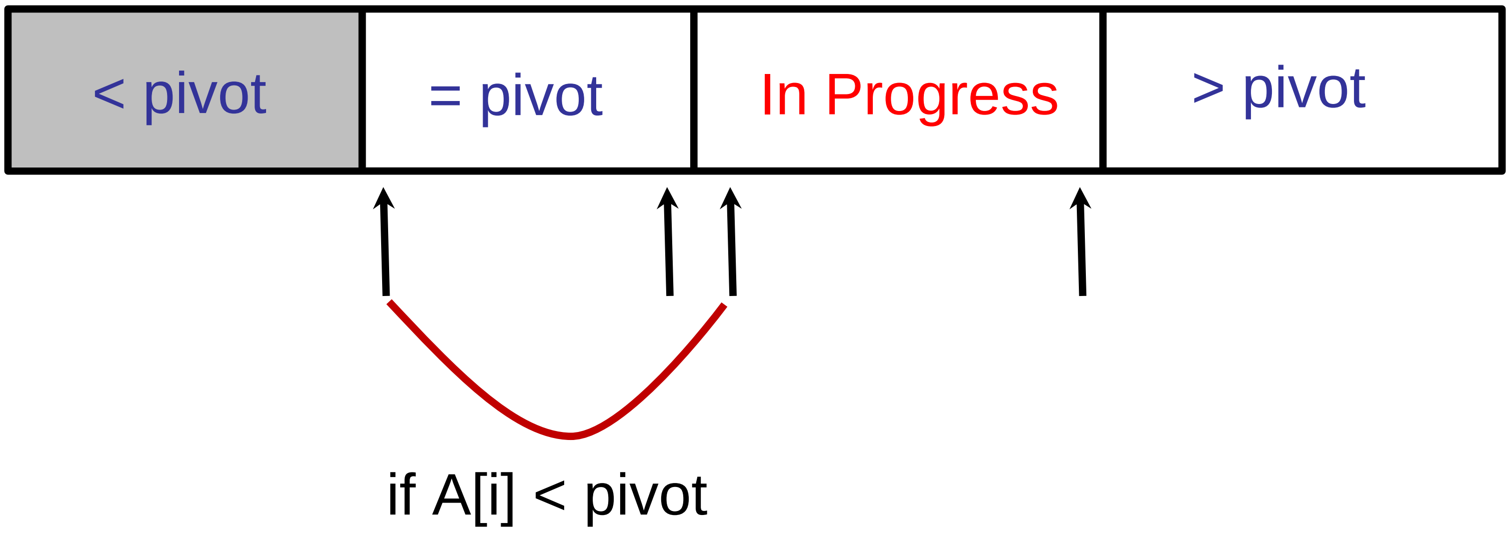
- Option 1: two pass partitioning
  1. Regular partition.
  2. Pack duplicates.

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

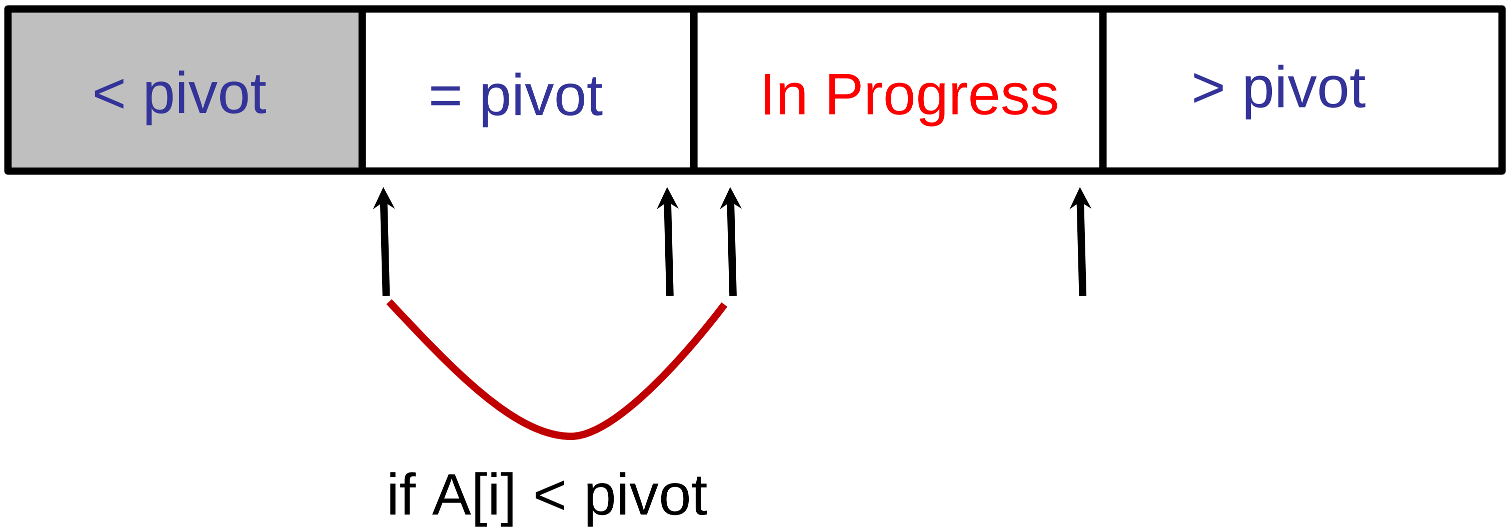
# 3-Way Partitioning



# 3-Way Partitioning

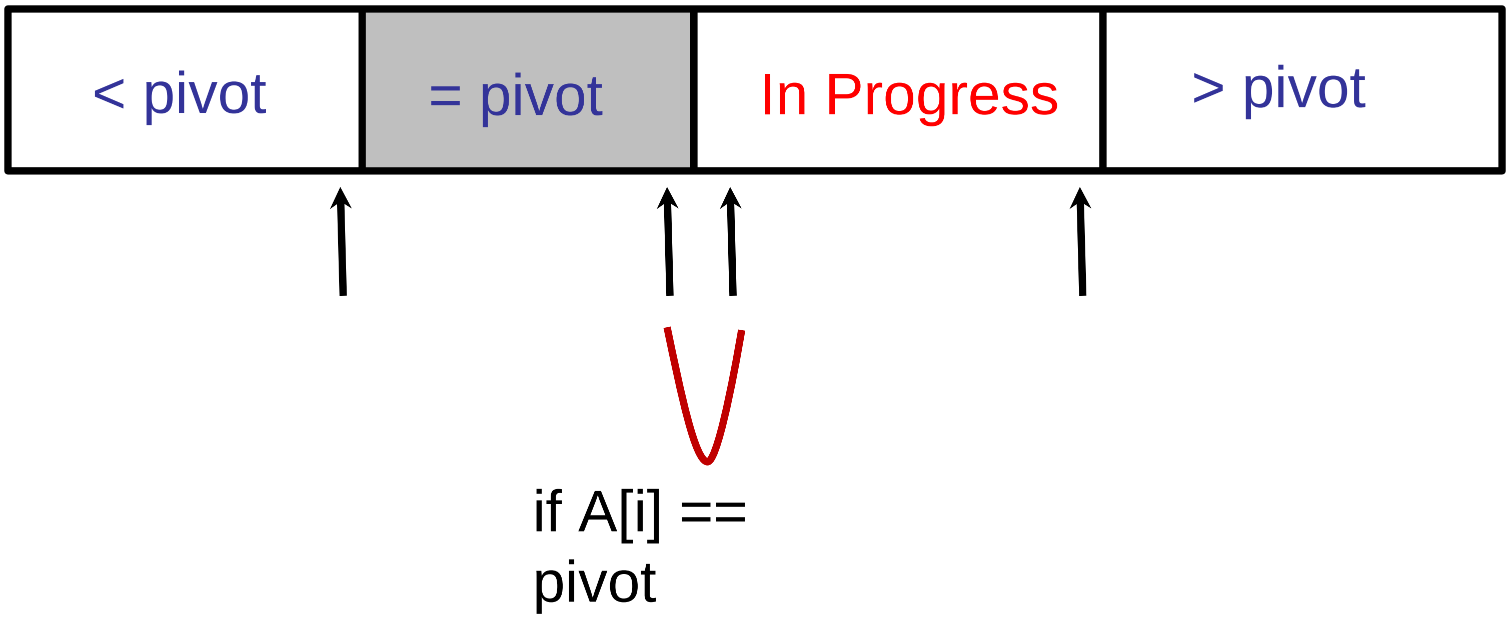


# 3-Way Partitioning

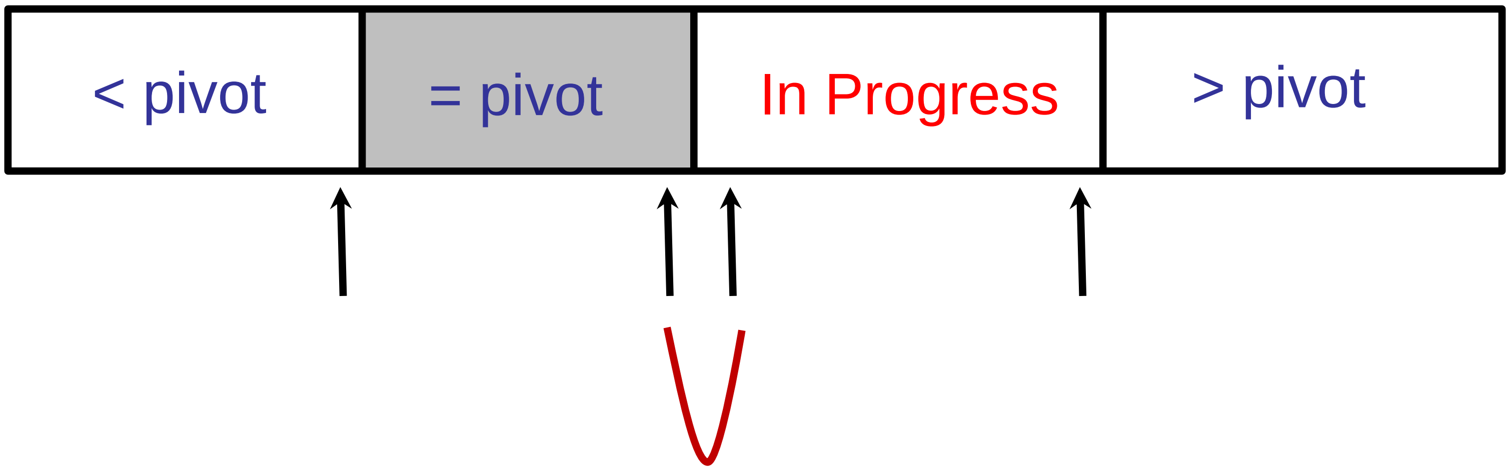



Just swap item at index i with first pivot element

# 3-Way Partitioning



# 3-Way Partitioning

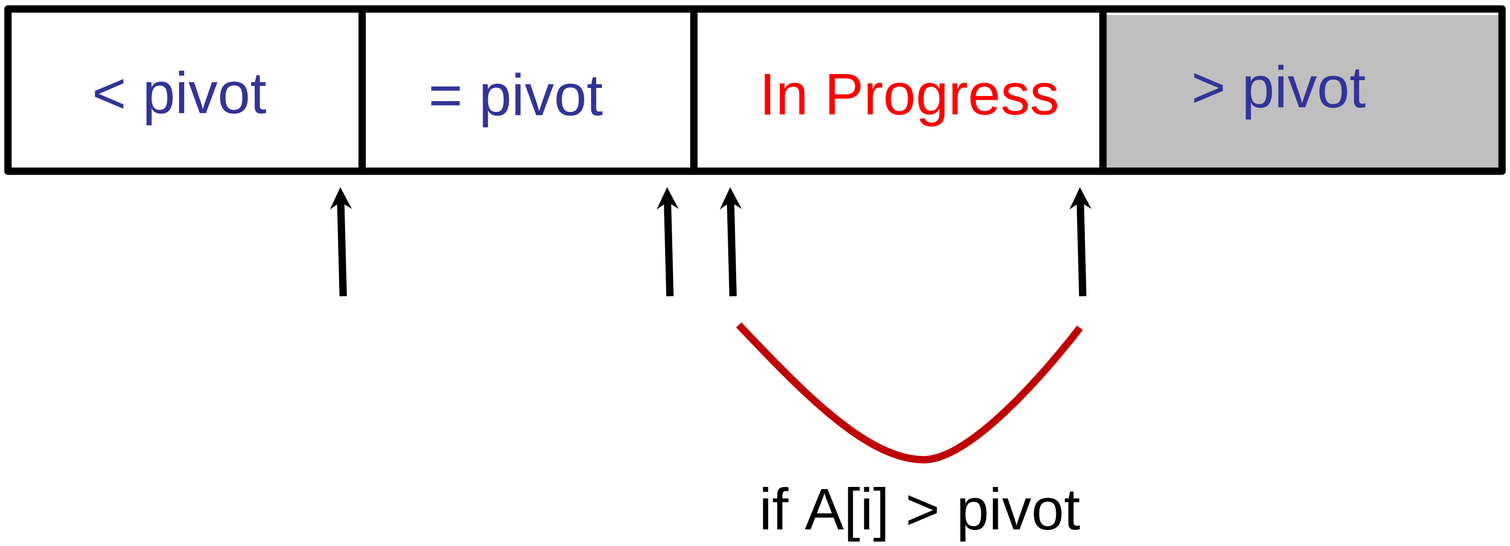



if A[i] ==
pivot

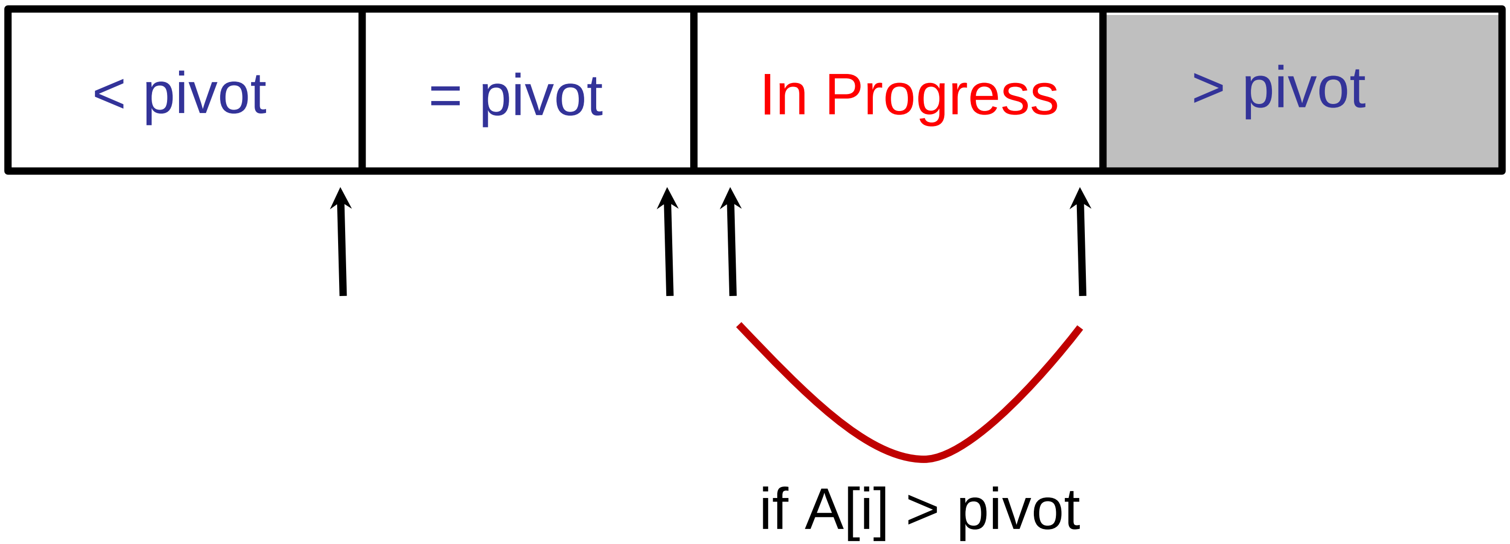
just increment the pivot end index by 1

this “grows” the pivot region

# 3-Way Partitioning

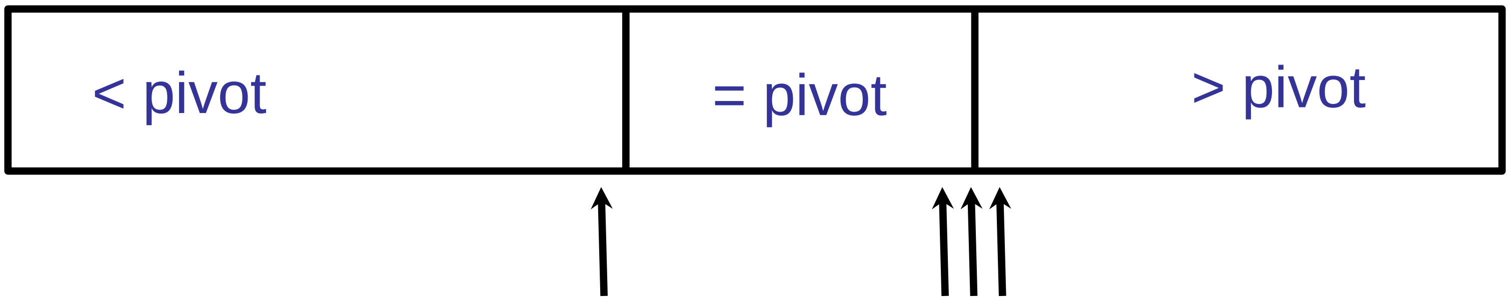


# 3-Way Partitioning




swap element at index i with beginning of the > pivot region

# 3-Way Partitioning



# Duplicates

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

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

**else**

Choose pivot index *pIndex*.

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

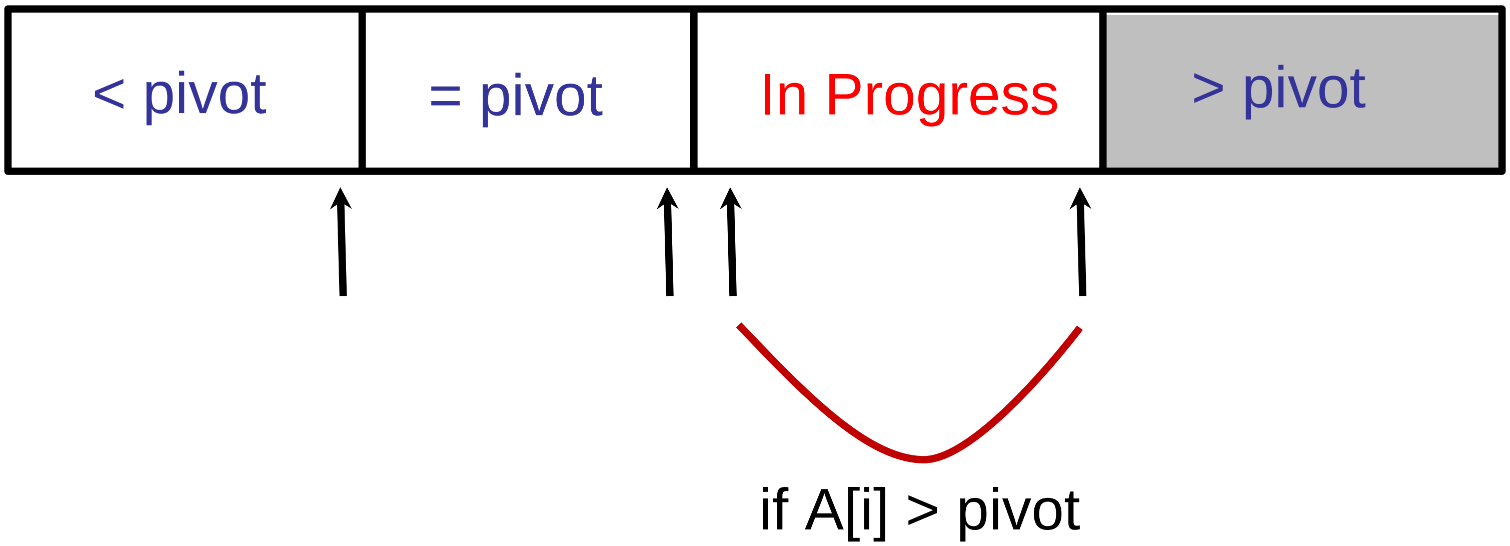
$x$ = QuickSort($A$[1..$p$–1], $p$–1)

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

| | | | | |
|---|---|---|---|---|
| $< x$ | $x$ | $x$ | $x$ | $> x$ |

# Is QuickSort stable?

# QuickSort is not stable



# Sorting, continued

QuickSort

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

# Choice of Pivot

Options:

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

# Choice of Pivot

Options:

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

In the worst case, it does not matter!

All options are equally bad.

# Choice of Pivot

Choose $A[1]$ for pivot:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 100 | 99 | 98 | 97 | 96 | 95 | 94 | 93 | 92 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 99 | 98 | 97 | 96 | 95 | 94 | 93 | 92 | 100 |

# Choice of Pivot

Choose $A[1]$ for pivot:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 99 | 98 | 97 | 96 | 95 | 94 | 93 | 92 | 100 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 98 | 97 | 96 | 95 | 94 | 93 | 92 | 99 | 100 |

# Choice of Pivot

Choose $A[1]$ for pivot:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 98 | 97 | 96 | 95 | 94 | 93 | 92 | 99 | 100 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 97 | 96 | 95 | 94 | 93 | 92 | 98 | 99 | 100 |

# Choice of Pivot

Choose $A[1]$ for pivot:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 98 | 97 | 96 | 95 | 94 | 93 | 92 | 99 | 100 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 97 | 96 | 95 | 94 | 93 | 92 | 98 | 99 | 100 |

# Choice of Pivot

Sorting the array takes *n* executions of partition.

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

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 98 | 97 | 96 | 95 | 94 | 93 | 92 | 99 | 100 |

⇩

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 97 | 96 | 95 | 94 | 93 | 92 | 98 | 99 | 100 |

# Deterministic QuickSort

QuickSort Recurrence:

$$T(n) = T(n-1) + T(1) + n$$

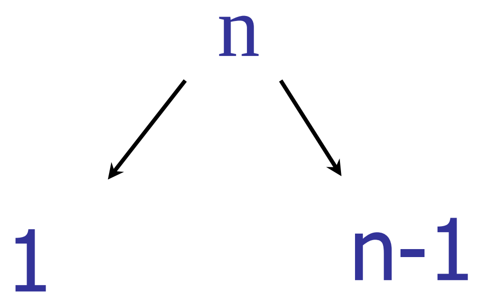
Cost of partition on n elements

Cost of QuickSort on 1 element

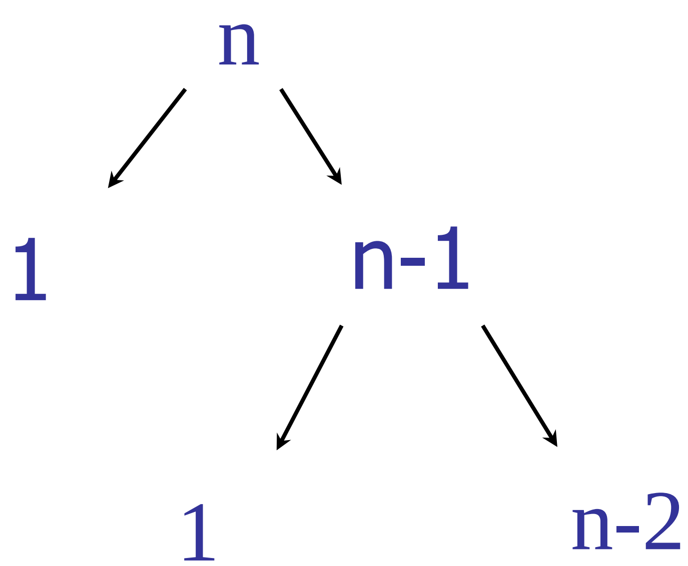
Cost of QuickSort on n – 1 elements

Cost of QuickSort on n elements

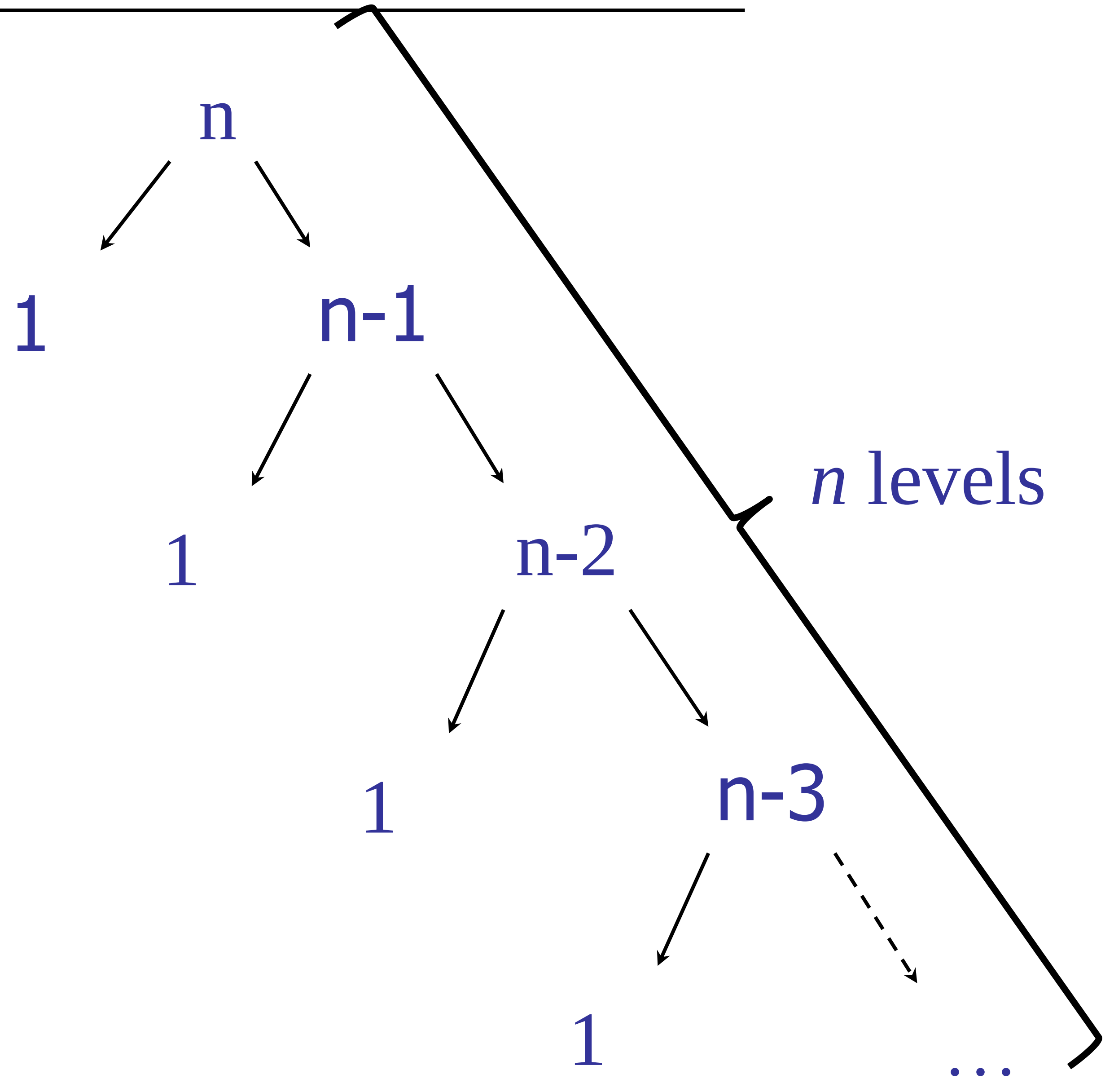
# Deterministic QuickSort



# Deterministic QuickSort

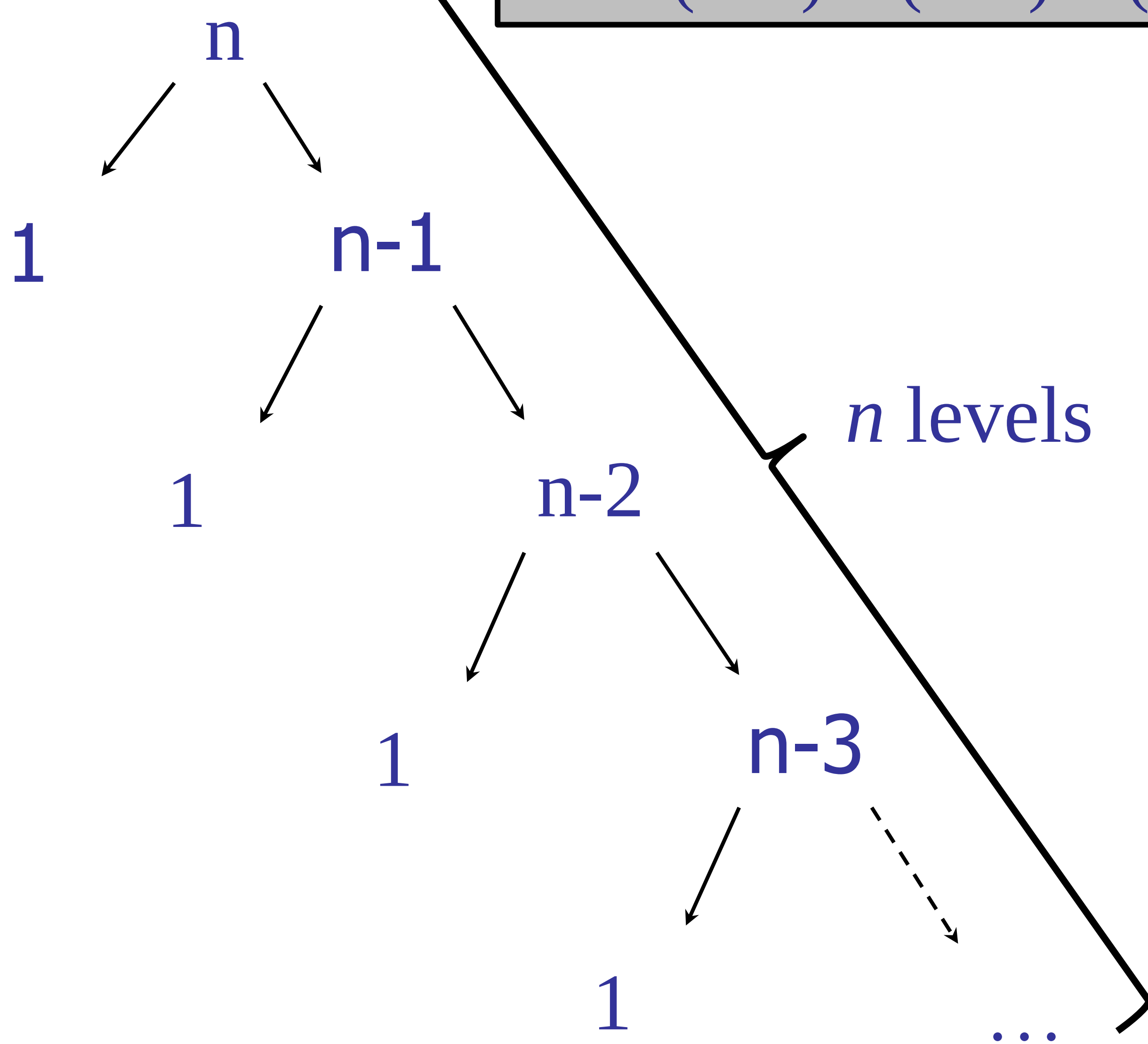


# Deterministic QuickSort



# Deterministic QuickSort

$n + (n-1) + (n-2) + (n-3) + \ldots = \mathbf{O(n^2)}$



# QuickSort

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

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

    **else**

        Choose pivot index *pIndex*.

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

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

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

| | | |
|---|---|---|
| $< x$ | $x$ | $> x$ |

# Better QuickSort

What if we chose the *median* element for the pivot?

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

Cost of partition on n elements

Cost of QuickSort on *high* elements

Cost of QuickSort on *low* elements

Cost of QuickSort on n elements

# Better QuickSort

If we split the array evenly:

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

# QuickSort Summary

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

# QuickSort Pivot Choice

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

- $L = \{A[i] : A[i] < pivot\}$
- $H = \{A[i] : A[i] > pivot\}$


What if the *pivot* is chosen so that:

1. L > n/10
2. H > n/10

# QuickSort

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

QuickSort with interesting *pivot* choice:

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

Cost of <u>partition</u> on n elements

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

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

Cost of <u>QuickSort</u> on $n$ elements

# QuickSort

Tempting solution:

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

What is wrong?

# QuickSort

Tempting solution:

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

Too loose an estimate. We didnt solve the last line properly.

# QuickSort

Tempting solution:

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

Intuition: Making 2 calls to problem sizes 9/10n leads to more work than O(nlog n)

# 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

$k = n/10$

$T(n)$

# QuickSort Analysis

$k = n/10$

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

# QuickSort Analysis

$k = n/10$

$T(n)$

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

$T(n/100)$ $T(9n/100)$ $T(9n/100)$ $T(18n/100)$

# QuickSort Analysis

$k = n/10$

$T(n)$

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

$T(n/100)$ $T(9n/100)$ $T(9n/100)$ $T(81n/100)$

$T(n/1000)$ $T(n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(81n/1000)$ $T(729n/1000)$

# QuickSort Analysis

$k = n/10$

$T(n)$ $= \text{n}$

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

$T(n/100)$ $T(9n/100)$ $T(9n/100)$ $T(81n/100)$

$T(n/1000)$ $T(n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(81n/1000)$ $T(729n/1000)$

# QuickSort Analysis

$k = n/10$

$T(n)$ $= n$

$T(n/10)$ $T(9n/10)$ $= n$

$T(n/100)$ $T(9n/100)$ $T(9n/100)$ $T(81n/100)$

$T(n/1000)$ $T(n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(81n/1000)$ $T(729n/1000)$

# QuickSort Analysis

$k = n/10$

$T(n)$ $= \mathrm{n}$

$T(n/10)$ $T(9n/10)$ $= \mathrm{n}$

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

$T(n/1000)$ $T(n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(81n/1000)$ $T(729n/1000)$

# QuickSort Analysis

$k = n/10$

$T(n)$ = n

$T(n/10)$ $T(9n/10)$ = n

$T(n/100)$ $T(9n/100)$ $T(9n/100)$ $T(81n/100)$ = n

$T(n/1000)$ $T(n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(81n/1000)$ $T(729n/1000)$ = n

# QuickSort Analysis

$k = n/10$

$T(n)$ $= n$

$T(n/10)$ $T(9n/10)$ $= n$

$T(n/100)$ $T(9n/100)$ $T(9n/100)$ $T(81n/100)$ $= n$

$T(n/1000)$ $T(n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(81n/1000)$ $T(729n/1000)$ $= n$

How many levels??

# QuickSort Analysis

$k = n/10$

$T(n)$ = n

$T(n/10)$ $T(9n/10)$ = n

$T(n/100)$ $T(9n/100)$ $T(9n/100)$ $T(81n/100)$ = n

$T(n/1000)$ $T(n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(9n/1000)$ $T(81n/1000)$ $T(81n/1000)$ $T(729n/1000)$ = n

How many levels??

# QuickSort Analysis

Maximum number of levels:

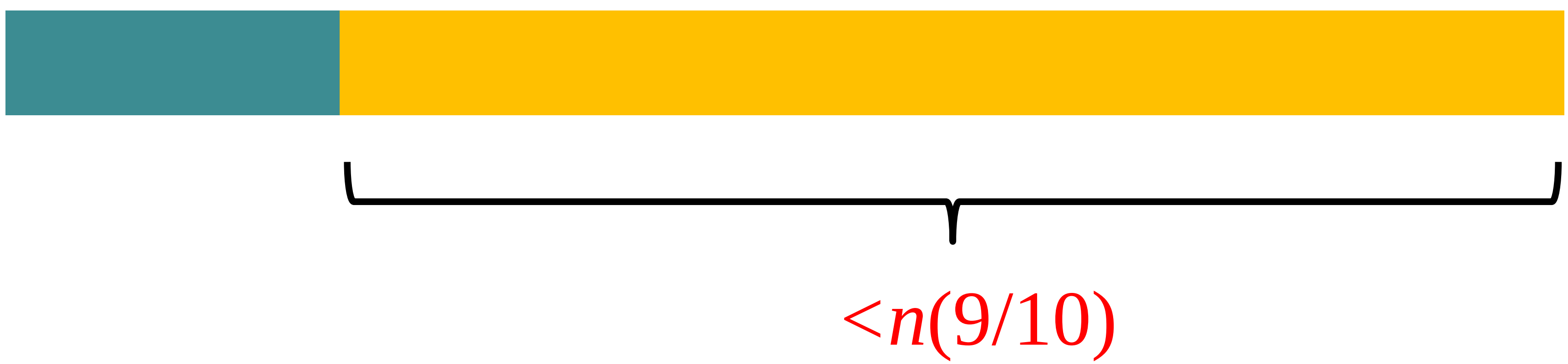
$$1 = n(9/10)^h$$

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

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

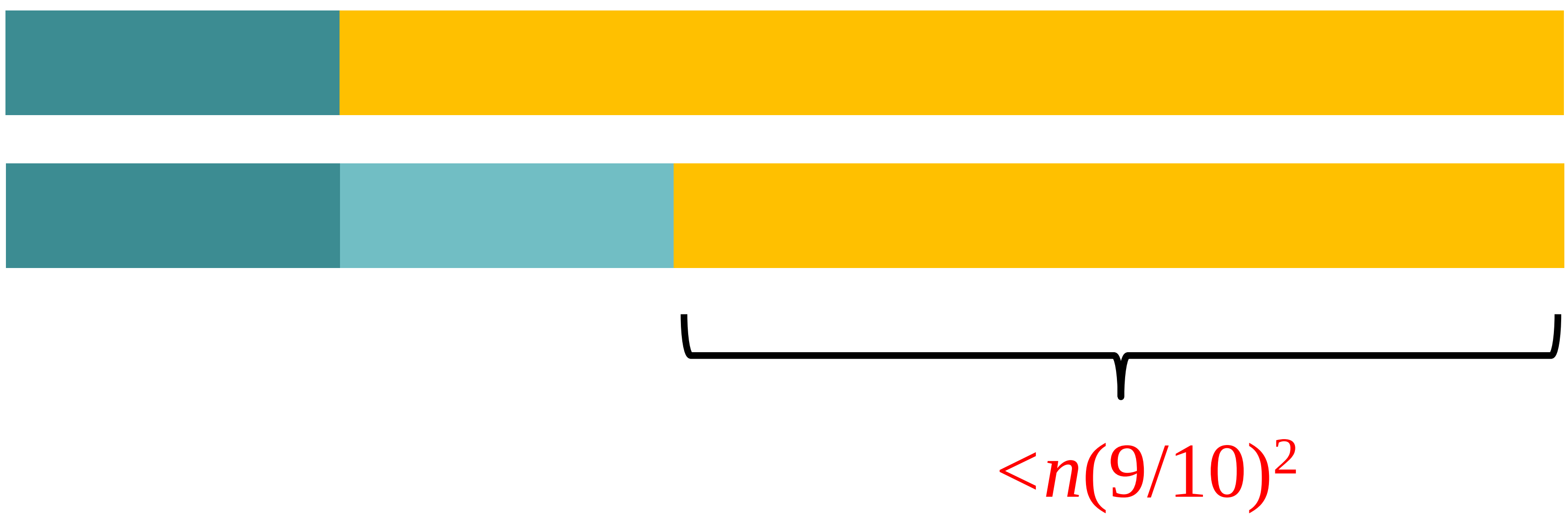
# QuickSort Analysis

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



# QuickSort Analysis

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



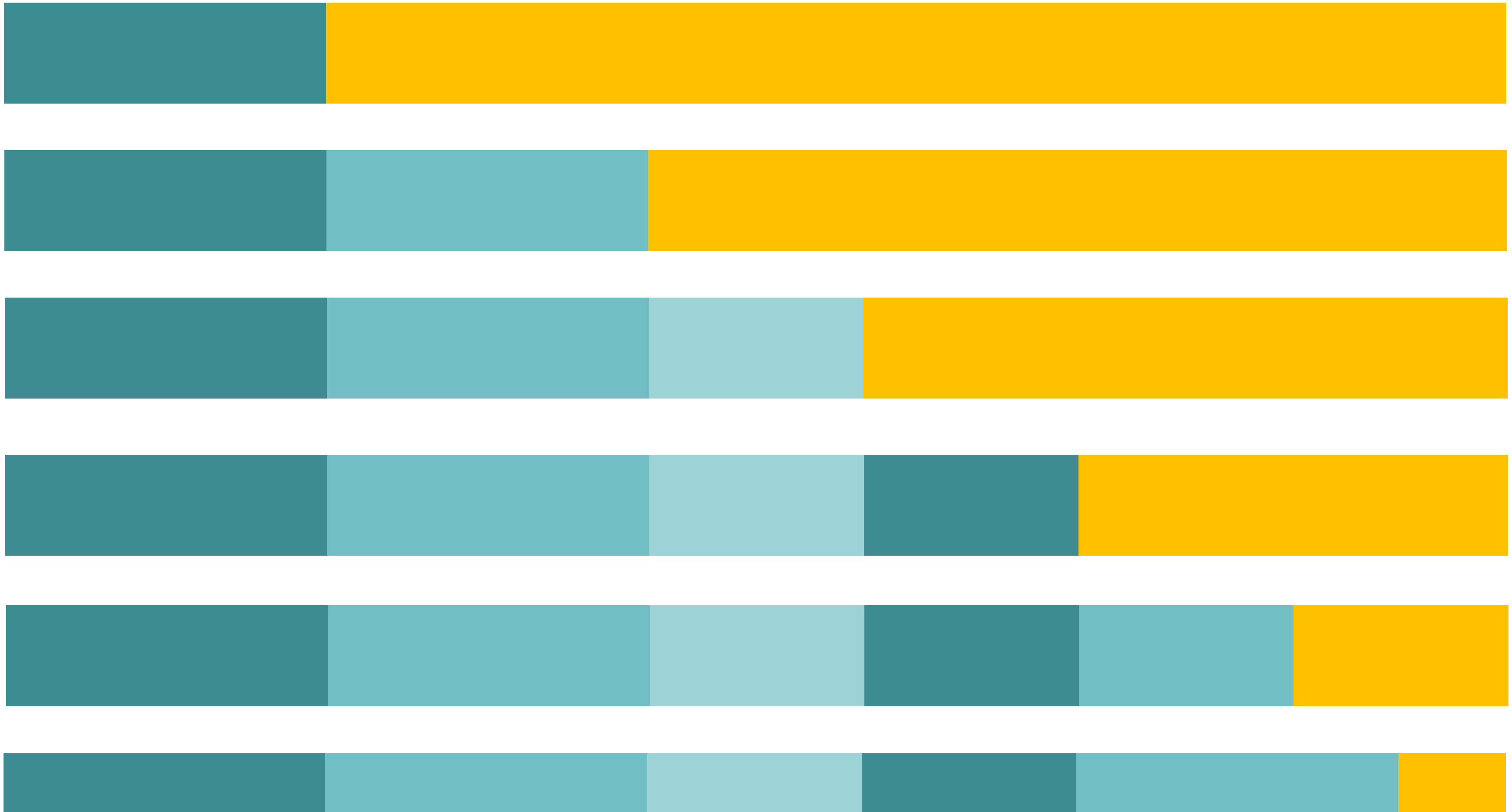
# QuickSort Analysis

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

$<n(9/10)^3$

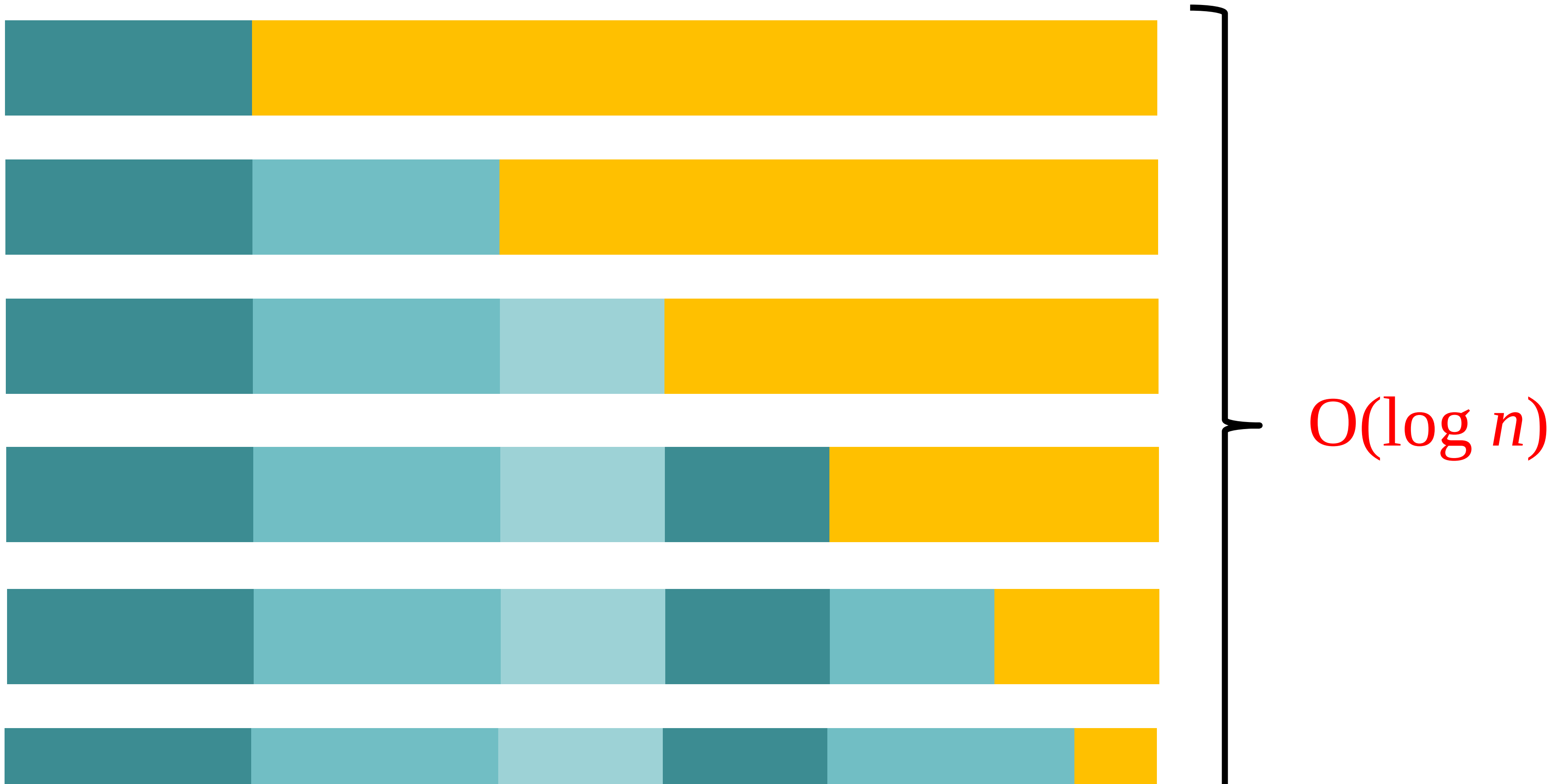
# QuickSort Analysis

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



# QuickSort Analysis

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



# QuickSort Summary

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

# QuickSort

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

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

**else**

Choose pivot index *pIndex*.

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

$x$ = QuickSort($A$[1..$p$–1], $p$–1)

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

# QuickSort

Key Idea:

- Choose the pivot at random.

Randomized Algorithms:

- Algorithm makes decision based on random coin flips.
- Can “fool” the adversary (who provides bad input)
- Running time is a *random variable*.

# Randomization

What is the difference between:

- Randomized algorithms
- Average-case analysis

# Randomization

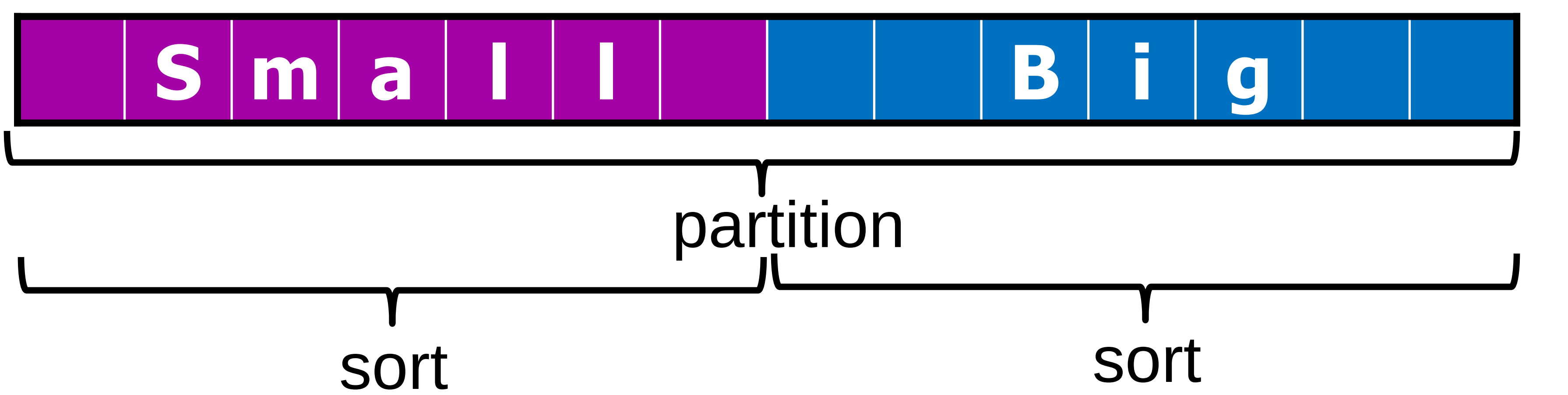
Randomized algorithm:

- Algorithm makes random choices
- For every input, there is a good probability of success.

Average-case analysis:

- Algorithm (may be) deterministic
- “Environment” chooses random input
- Some inputs are good, some inputs are bad
- For most inputs, the algorithm succeeds

```
QuickSort(A[1..n], n)
    if (n == 1) then return;
    else
      pIndex = random(1, n)
      p = 3WayPartition(A[1..n], n, pindex)
      x = QuickSort(A[1..p-1], p-1)
      y = QuickSort(A[p+1..n], n-p)
```



# Paranoid QuickSort

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

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

# Paranoid QuickSort

Easier to analyze:

- Every time we recurse, we reduce the problem size by at least (1/10).
- We have already analyzed that recurrence!

Note: non-paranoid QuickSort works too

- Analysis is a little trickier (but not much).

# Paranoid QuickSort

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

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

# Paranoid QuickSort

T(n) ParanoidQuickSort(A[1..n], n)

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

**else**

**repeat**

pIndex = **random**(1,

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

**until** p > (1/10)n **and** p < (

T(k) x = QuickSort(A[1..p–1], p

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

*9n/10 > k > n/10*

# Paranoid QuickSort

T(n)

1

O(n)
(in expectation)

T(k)
T(n-k)

*9n/10 > k > n/10*

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

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

# Paranoid QuickSort

Key claim:

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

Then we know:

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

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

# Paranoid QuickSort

T(n) → **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*

# Summary

QuickSort:

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

Next Week:

- Probability Theory
- Randomized Analysis
- Ordered Statistics