CS2040S
Data Structures and Algorithms

Welcome!

Last Time: Sorting, Part I

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

Problem Set 3

Sorting Detective

- Six suspicious sorting algorithms
  - Investigate the mysterious sorting code.
  - Identify each sorting algorithm.
  - Find the criminal: Dr. Evil!

- Focus on the properties:
  - Asymptotic performance
  - Stability
  - Performance on special inputs

- Absolute speed is not a good reason…

Problem Set 3

Sorting Detective

– Six suspicious sorting algorithms
  - Investigate the mysterious sorting
  - Identify each sorting algorithm
  - Find the criminal: Dr. Evil!

I compared the speed of A and B, and B was much faster so it must be InsertionSort.

It ran the fastest so it must be QuickSort.

operties:

mance

ability

erformance on special inputs

– Absolute speed is not a good reason…

Problem Set 3

Sorting Detective

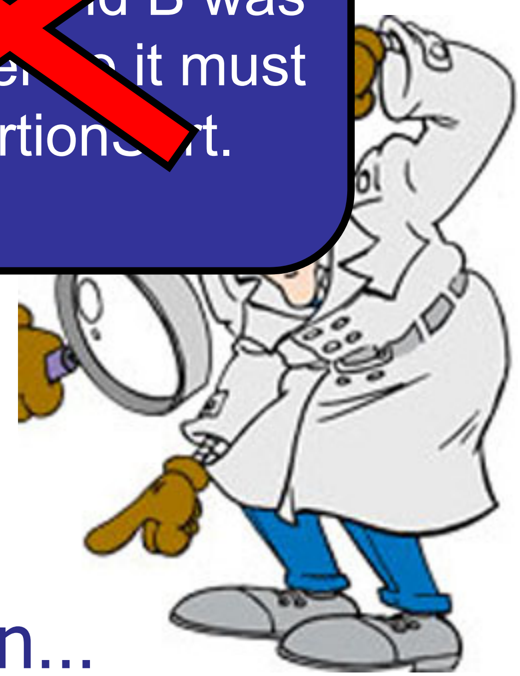
- Six suspicious sorting algorithms
  - Investigate the mysterious sorting
  - Identify each sorting algorithm
  - Find the criminal: Dr. Evil!

It ran t… f…est so it must b… …ickSort.

I co…ared the …peed of A an… …d B was much f… …it must b… insertion…t.

…operties:

- …mance
- …ability
- …erformance on special inputs



- Absolute speed is not a good reason…

Today: Sorting, Part II

MergeSort

- Space Analysis

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:

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

MergeSort

Step 1:
Divide array into two pieces.

MergeSort(A, n)

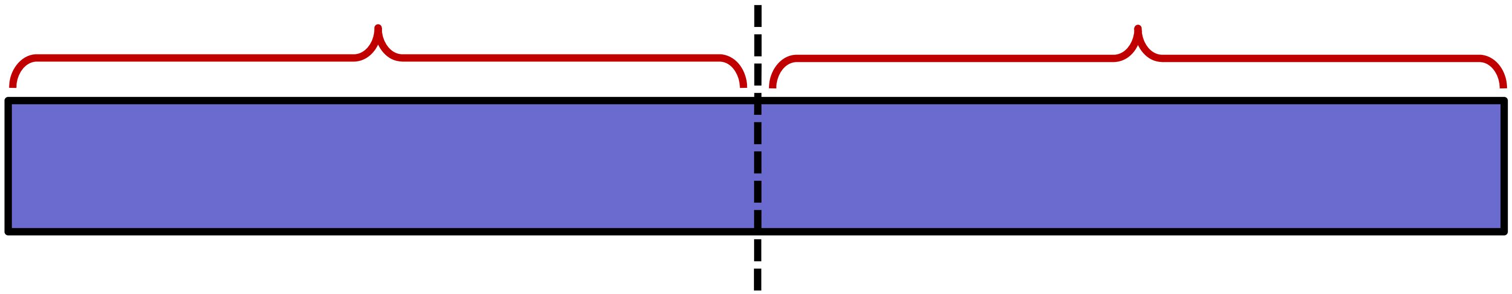
if (n=1) **then return;**

else:

X ←MergeSort**(**A[1..n/2], n/2**);**

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

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



MergeSort

Step 2:
Recursively sort the two halves.

MergeSort(A, n)

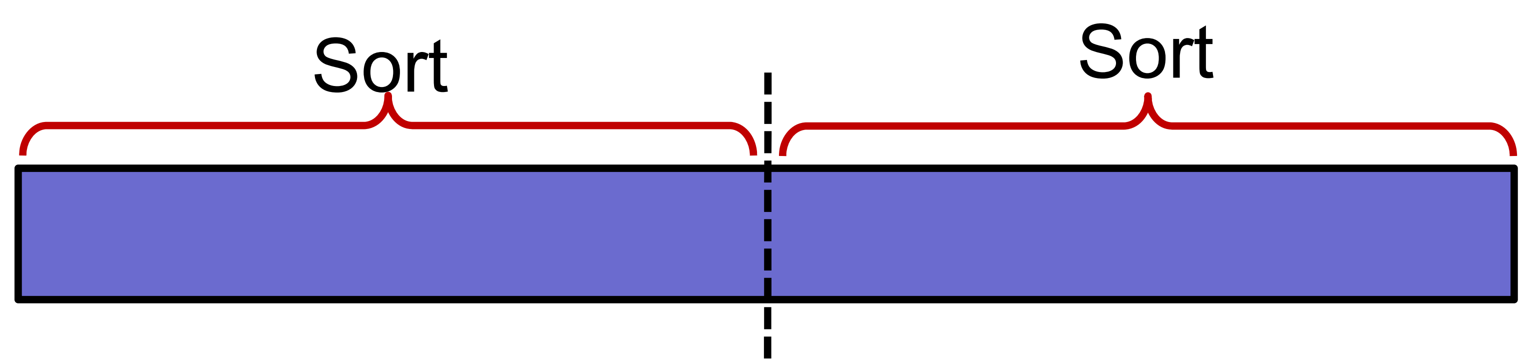
if (n=1) **then return;**

else:

X ←MergeSort**(**A[1..n/2], n/2**);**

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

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



MergeSort

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

MergeSort(A, n)

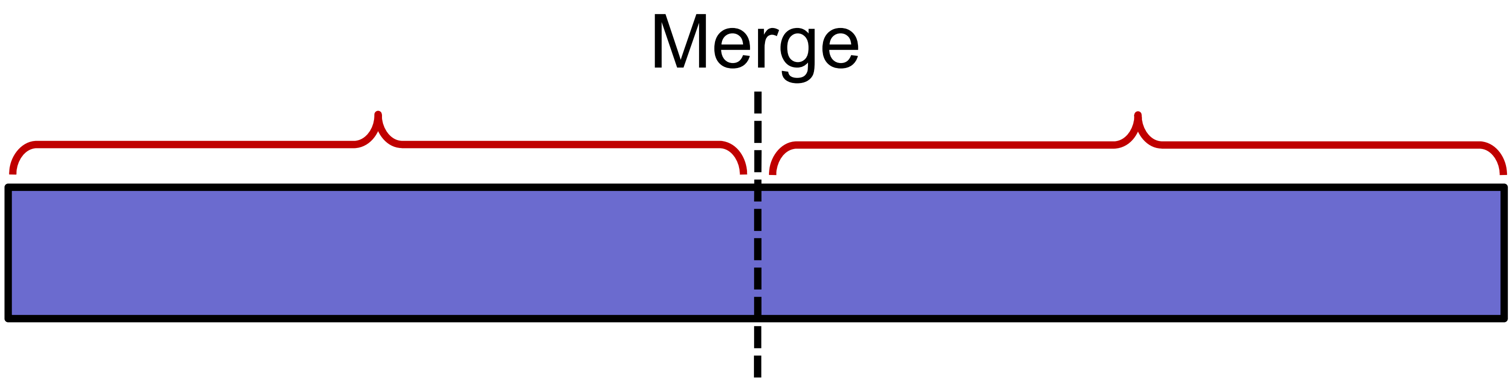
if (n=1) **then return;**

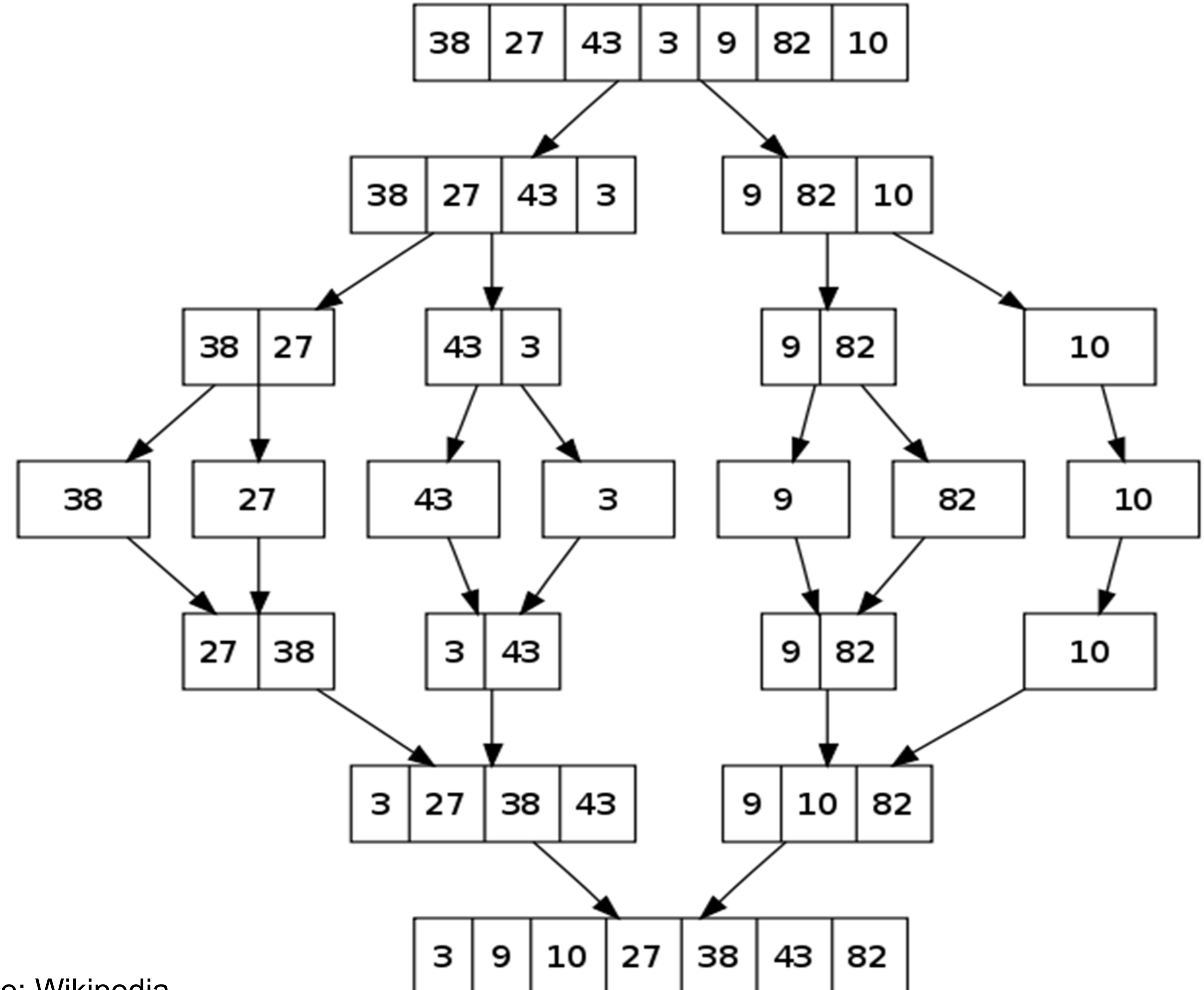
else:

X $\leftarrow$MergeSort**(**A[1..n/2], n/2**);**

Y $\leftarrow$MergeSort**(**A[n/2+1, n], n/2**);**

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





Source: Wikipedia

MergeSort, Bottom Up

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

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.

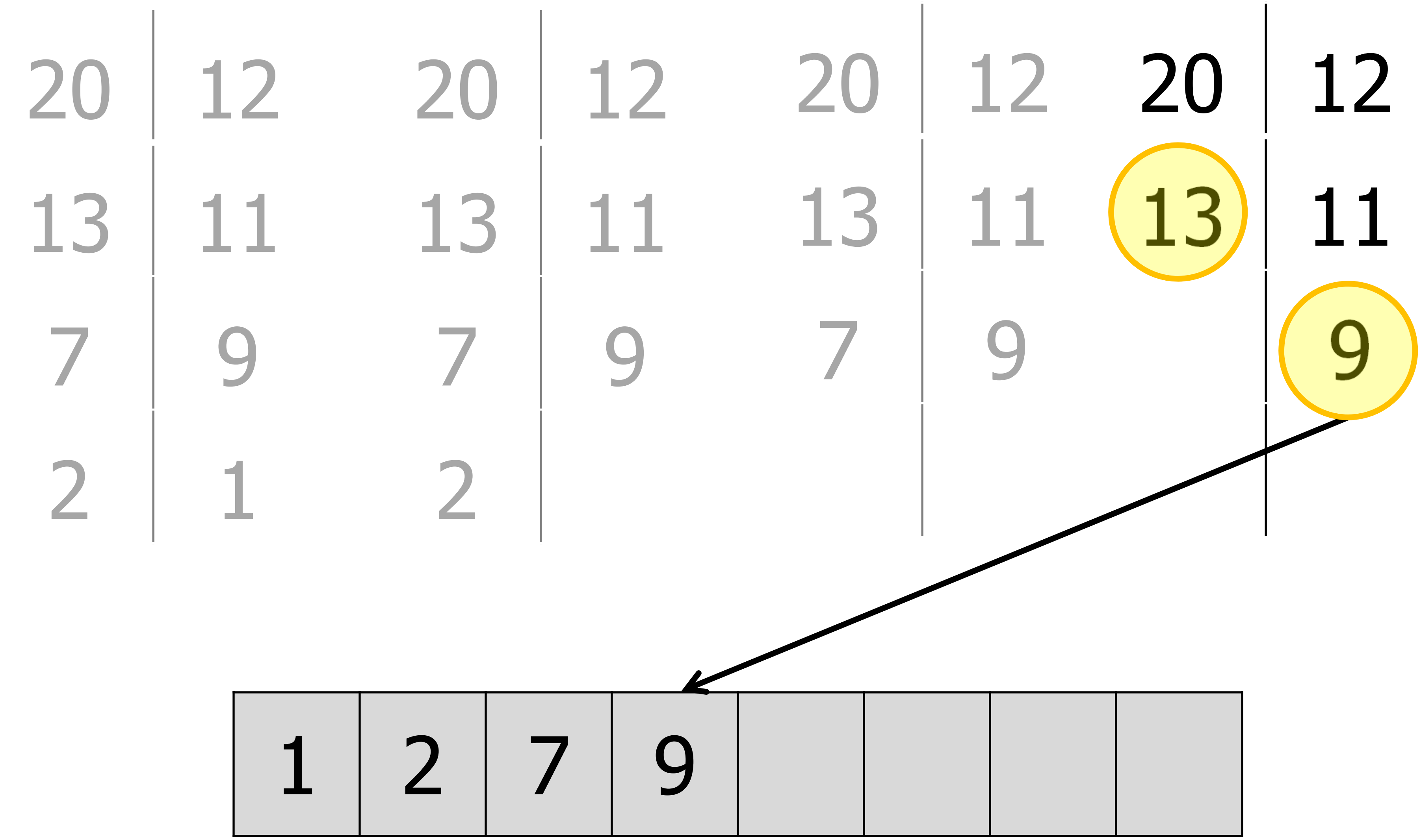
Space Complexity

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. $O(2^n)$

ARCHIPELAGO

is open

Space Analysis

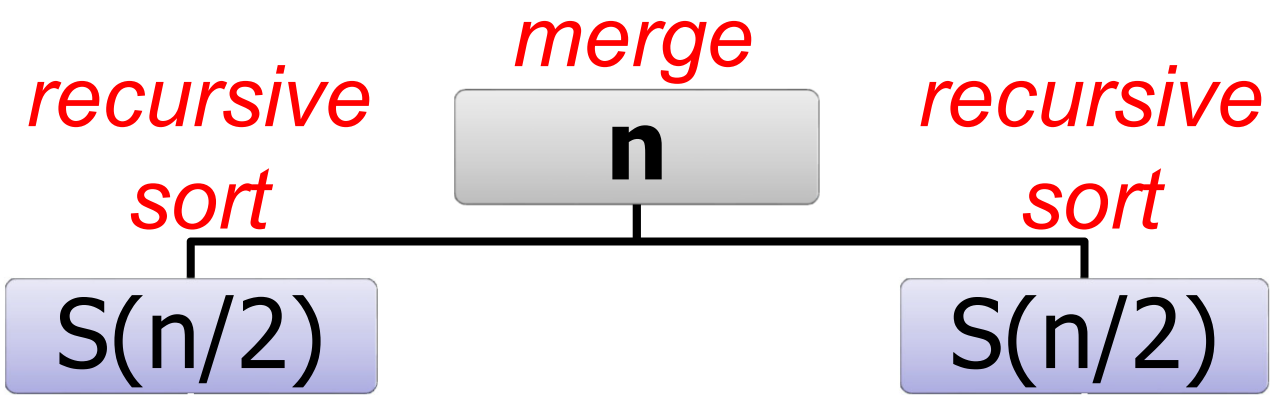
Let $S(n)$ be the worst-case space for an array of n elements.

$$S(n) = \theta(1) \quad \textbf{if } (n=1)$$

$$= 2S(n/2) + n \quad \textbf{if } (n>1)$$

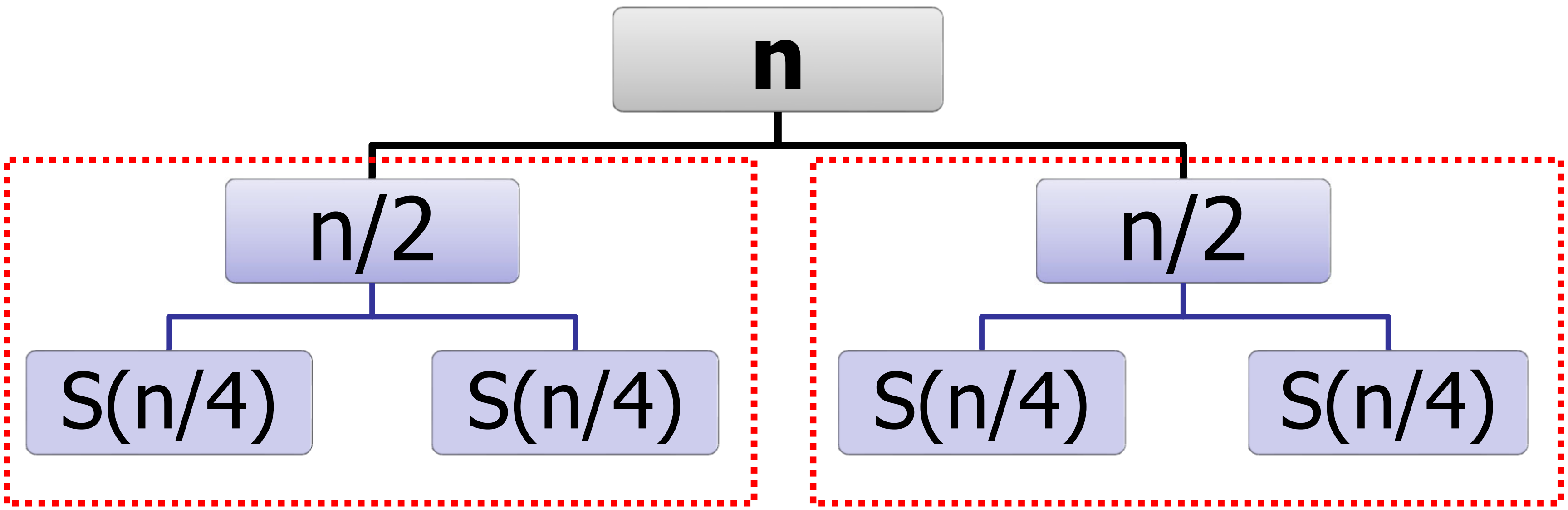
Space Analysis

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



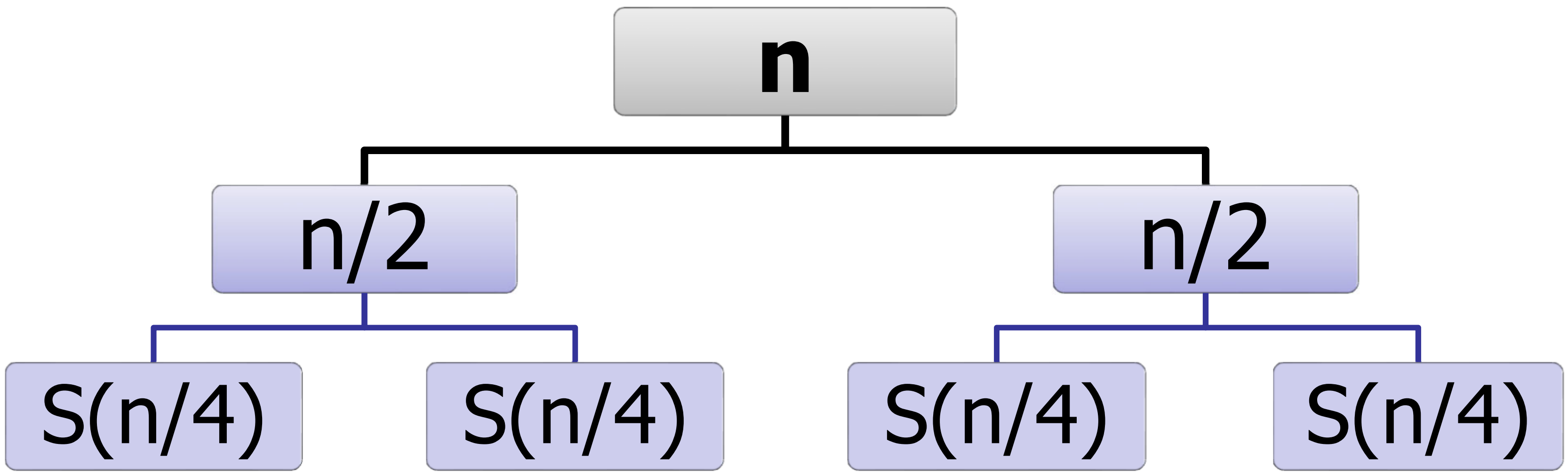
Space Analysis

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



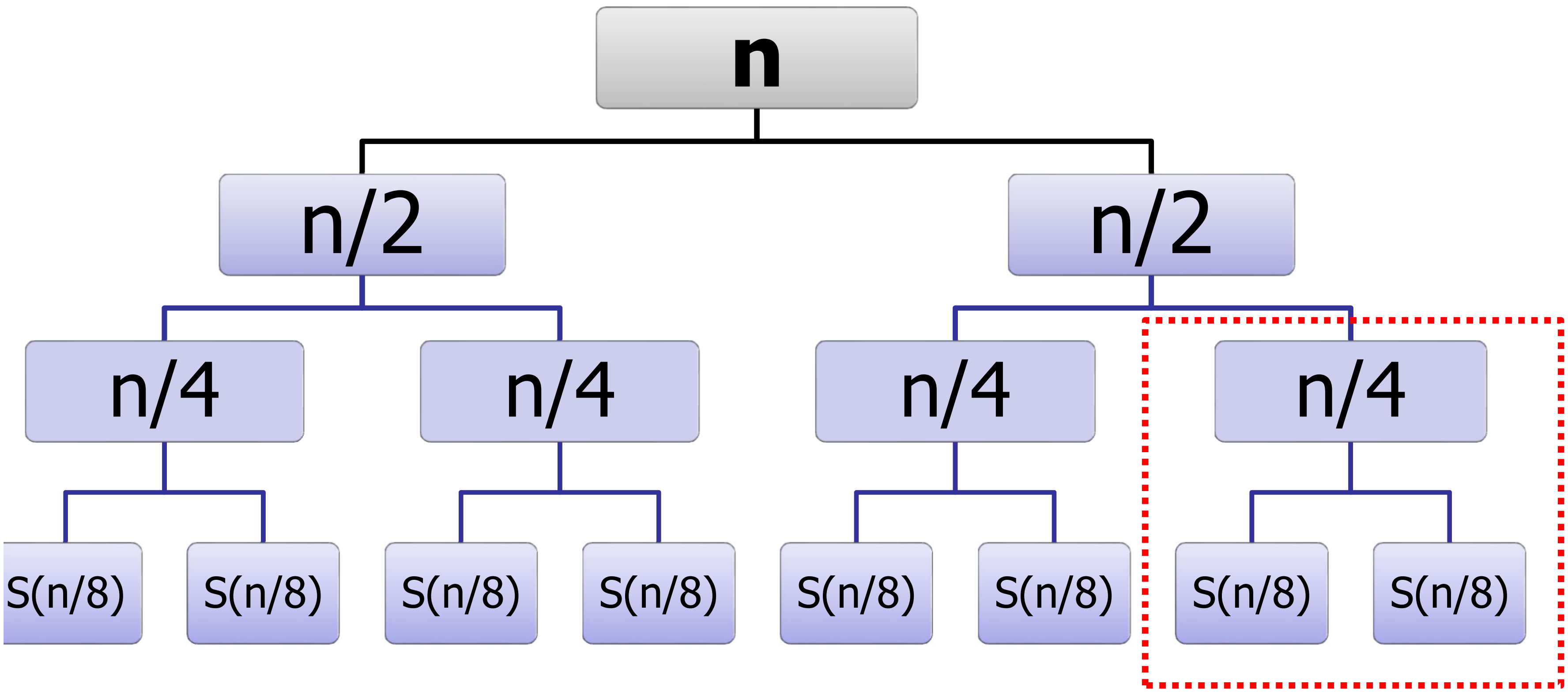
Space Analysis

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



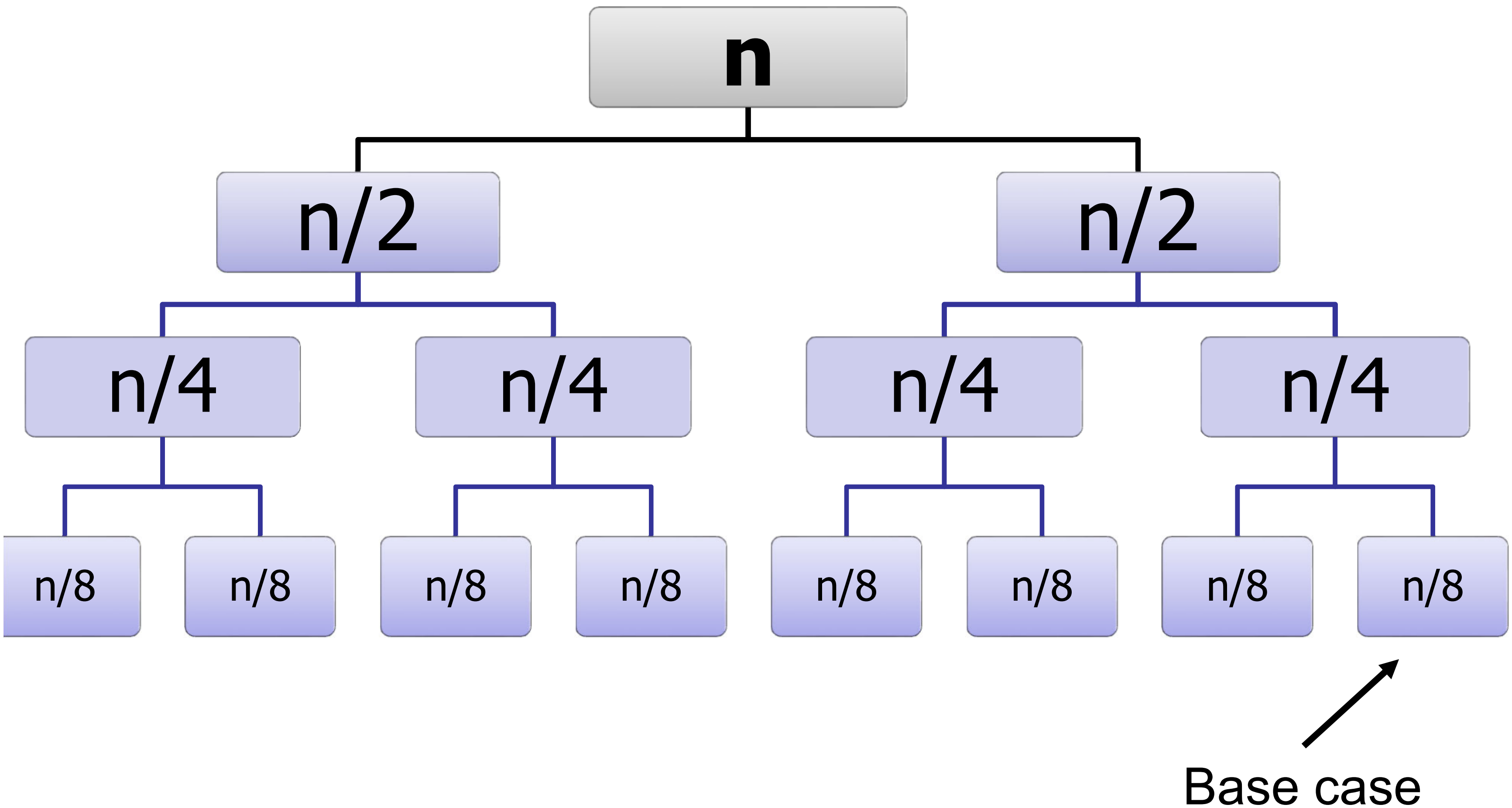
Space Analysis

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



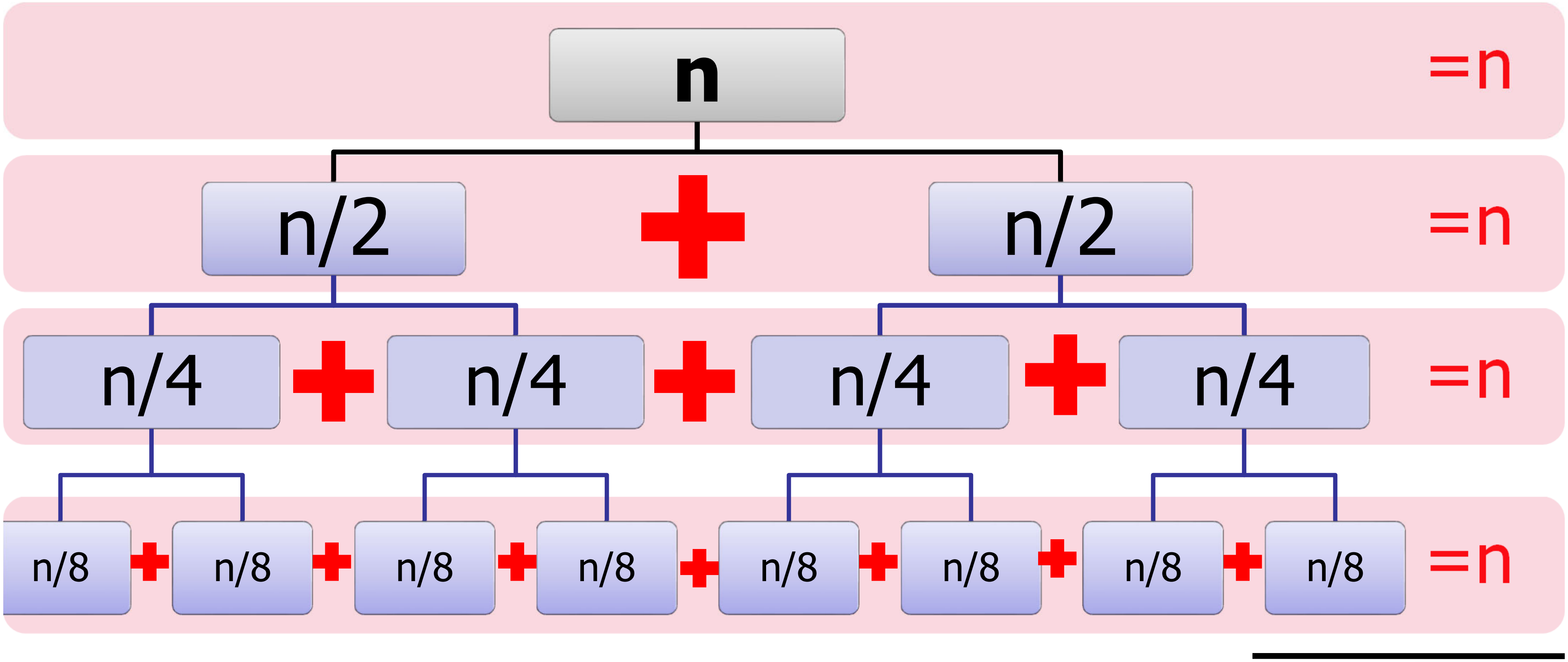
Space Analysis

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



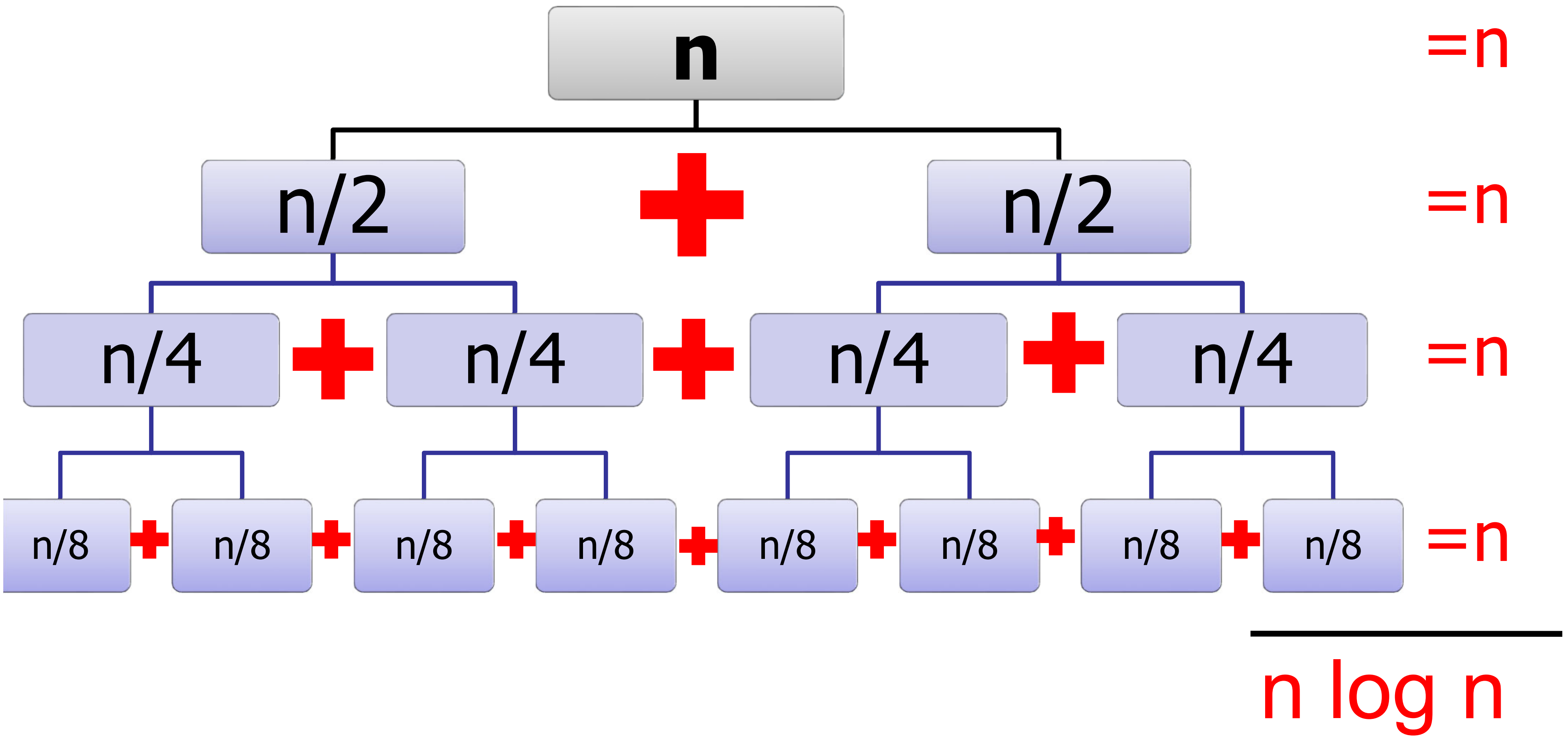
Space Analysis

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



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 $\leftarrow$ MergeSort**(**…**);**

Y $\leftarrow$ MergeSort**(**…**);** $S(n/2)$

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

$\theta(n)$

Better Space Usage

Implement MergeSort where:

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

  MergeSort (int[] A, int[] tempArray)

- No new arrays are allocated during the sort.

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

        n/2

        MergeSort(A, begin, mid, tempArray);

        n/2

        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

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

```
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);
```

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

$S(n) = ?$

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

ARCHIPELAGO
is open

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);~~
```

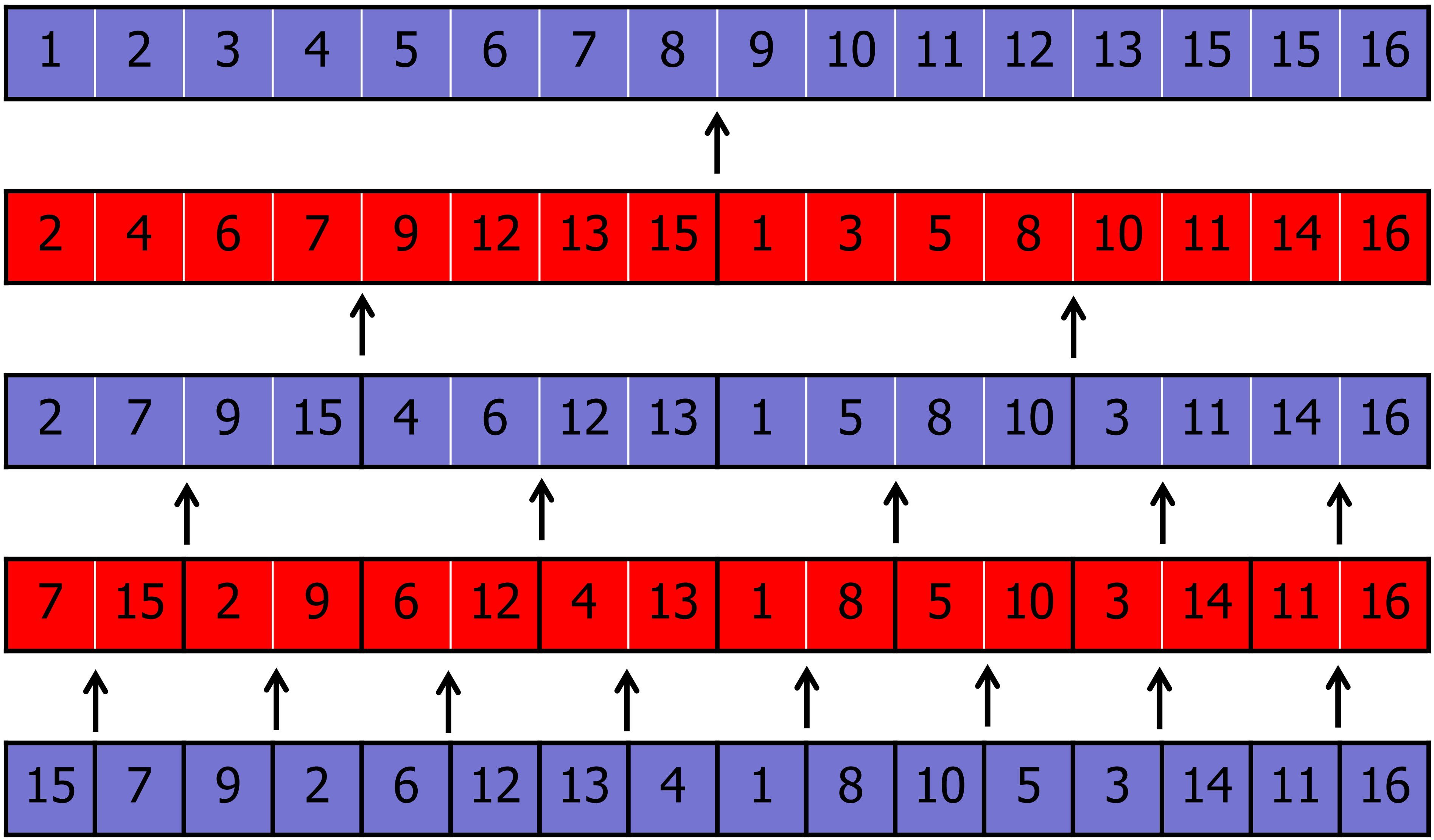
Initially, both A and B have copies of the unsorted array.

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

MergeSort, Bottom Up

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

MergeSort, Bottom Up



Summary

Name	Best Case	Average Case	Worst Case	Extra Memory	Stable?
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes

Today: Sorting, Part II

QuickSort

- Divide-and-Conquer
- Paranoid QuickSort
- Randomized 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)

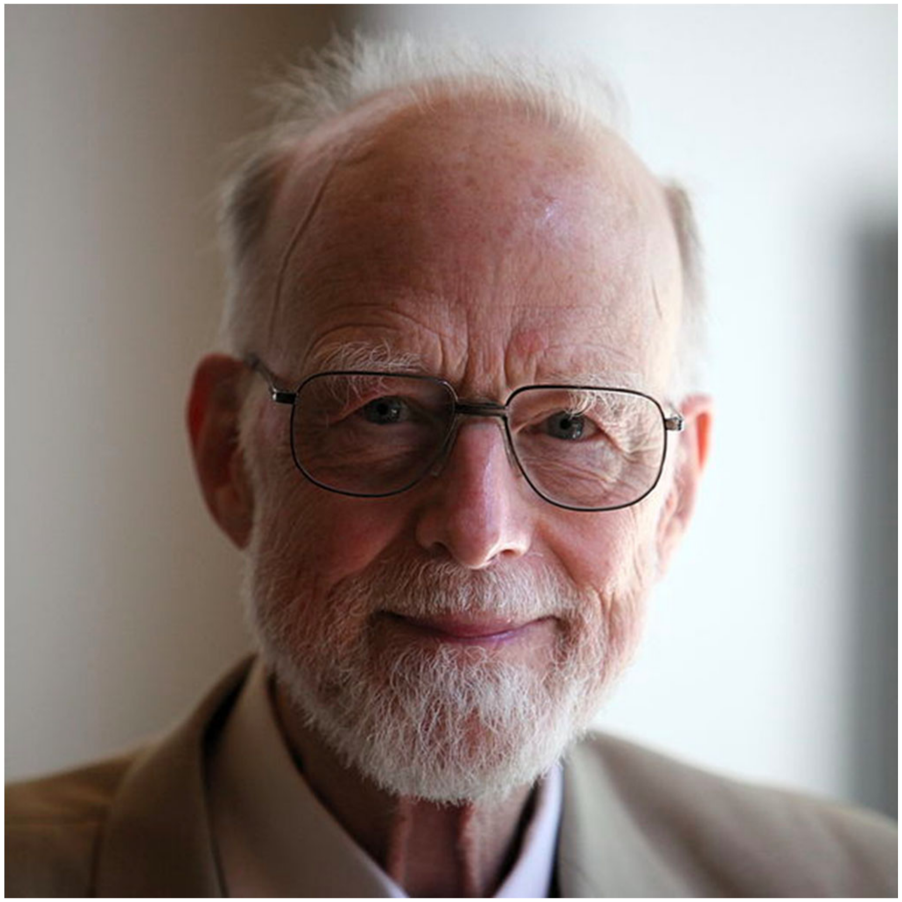


Photo: Wikimedia Commons (Rama)

Hoare

Quote:

“There are two ways of constructing a software design:

One way is to make it <u>so simple </u>that there are obviously no deficiencies, and the other way is to make it <u>so complicated </u>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!

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

QuickSort

In class:

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

Recall: MergeSort

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

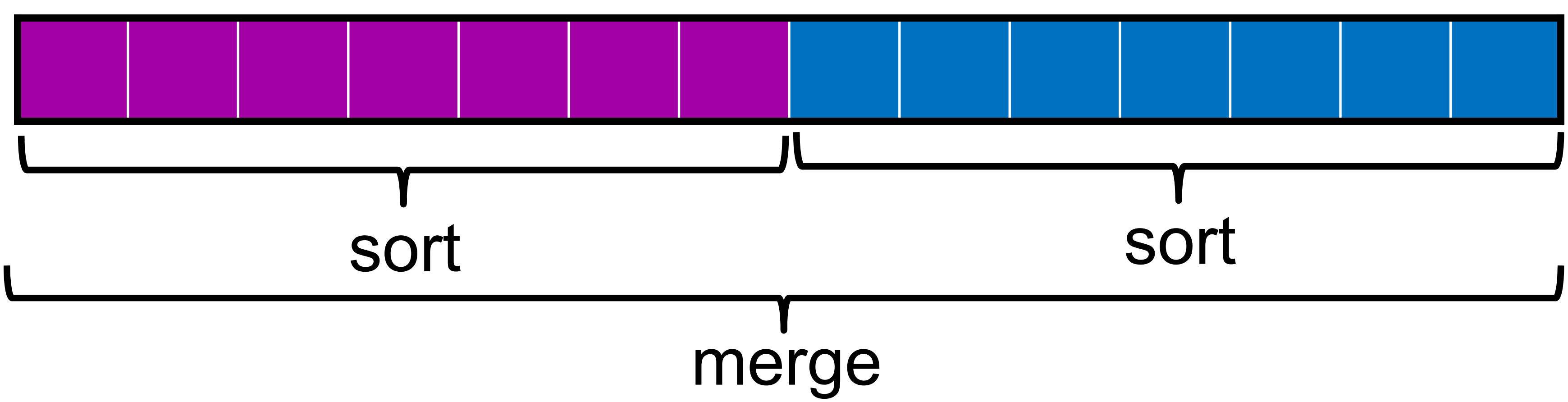
    if (n==1) **then** return;

    else

        x = MergeSort(A[1..n/2], n/2)

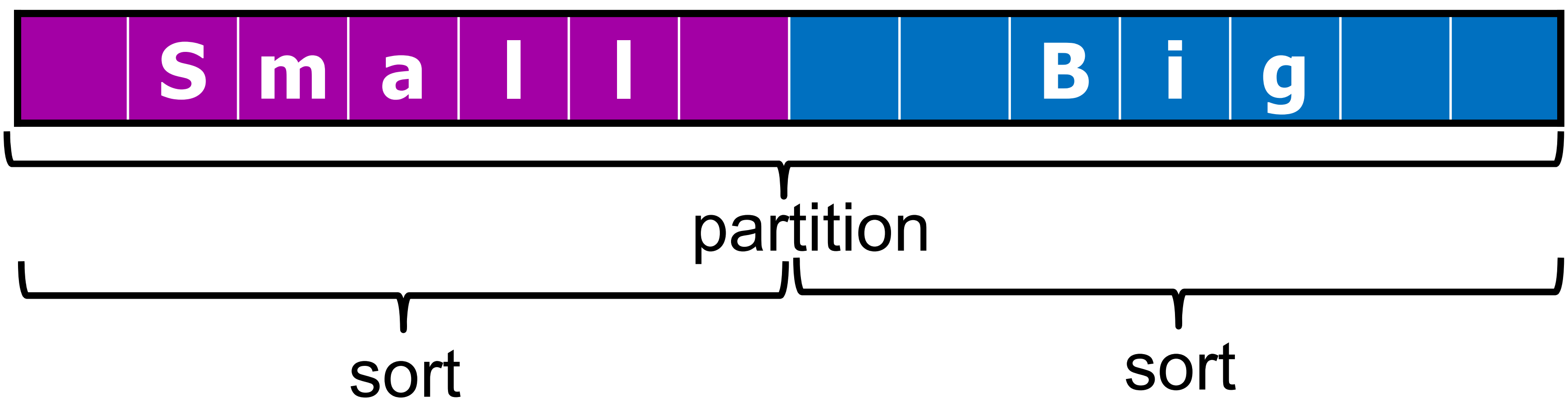
        y = MergeSort(A[n/2+1..n], n/2)

        return merge(x, y, n/2)



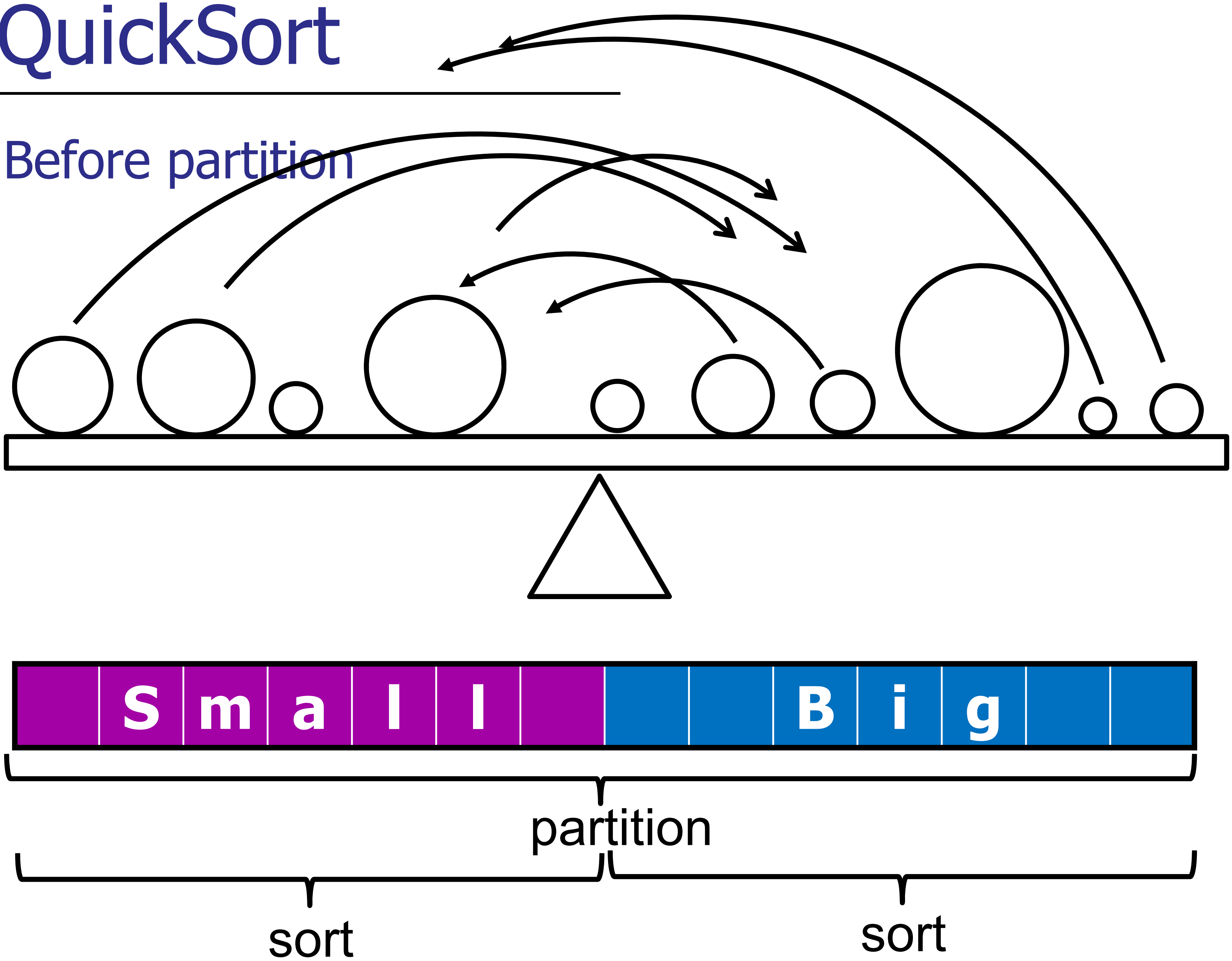
QuickSort

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



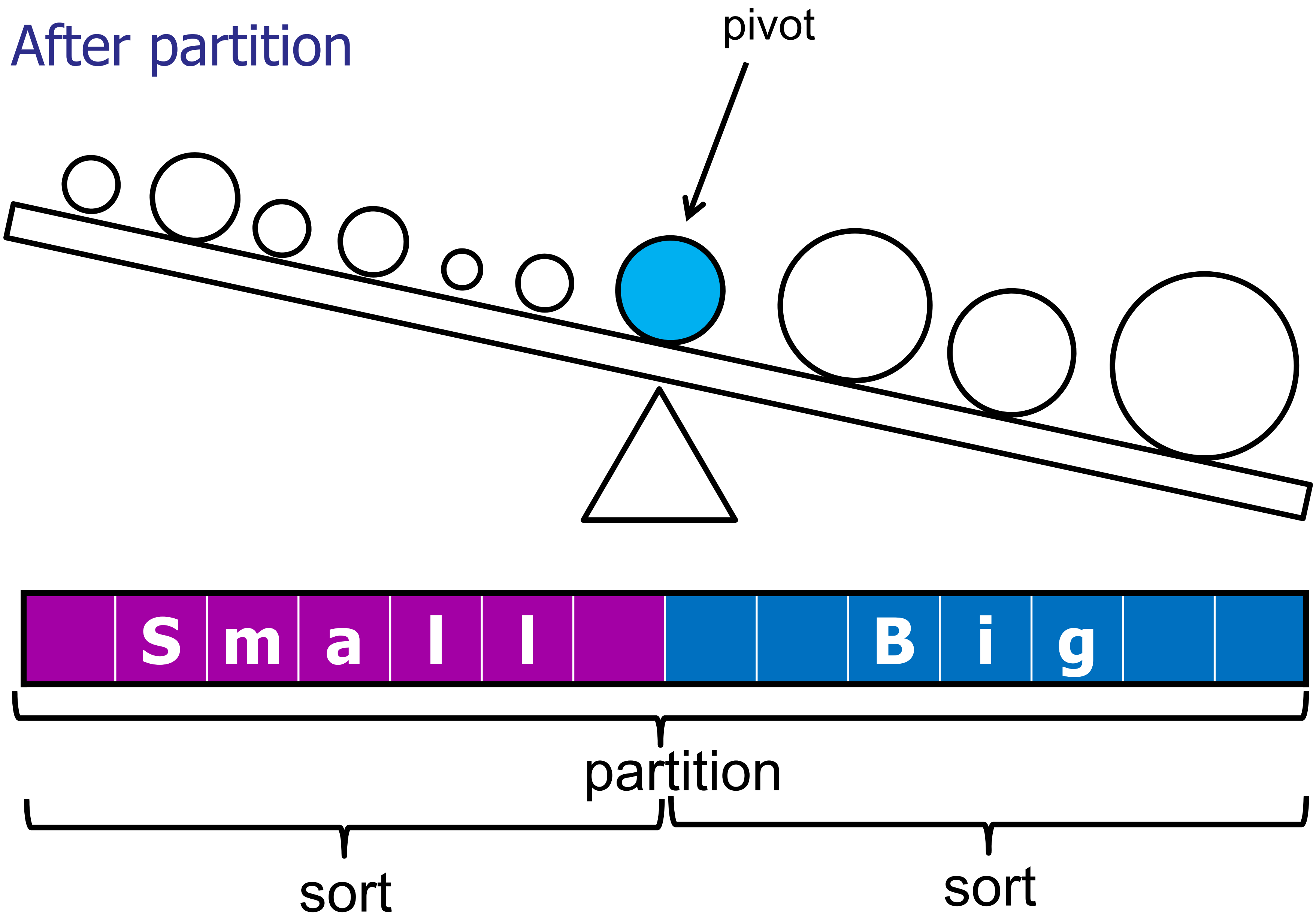
QuickSort

Before partition

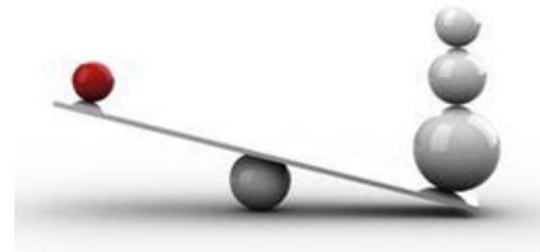


QuickSort

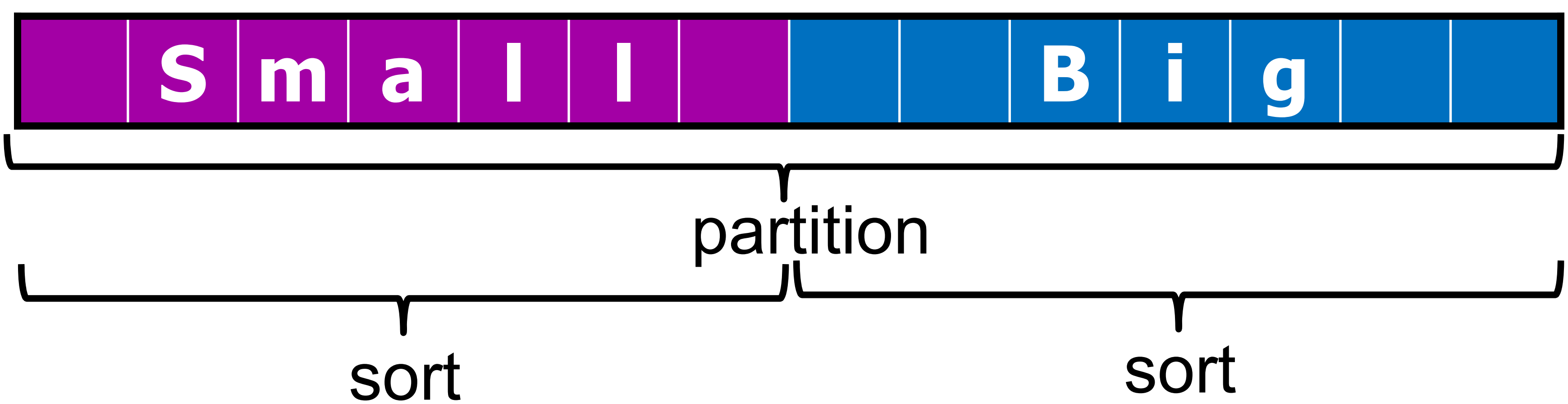
After partition



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.
2. Find all elements smaller than the pivot.
3. Find all elements larger than the pivot.

$< x$	x	$> \mathbf{x}$

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
$\boxed{3 \quad 4 \quad 2}$			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	50

a. 6

b. 10

c. 22

d. 40

e. 32

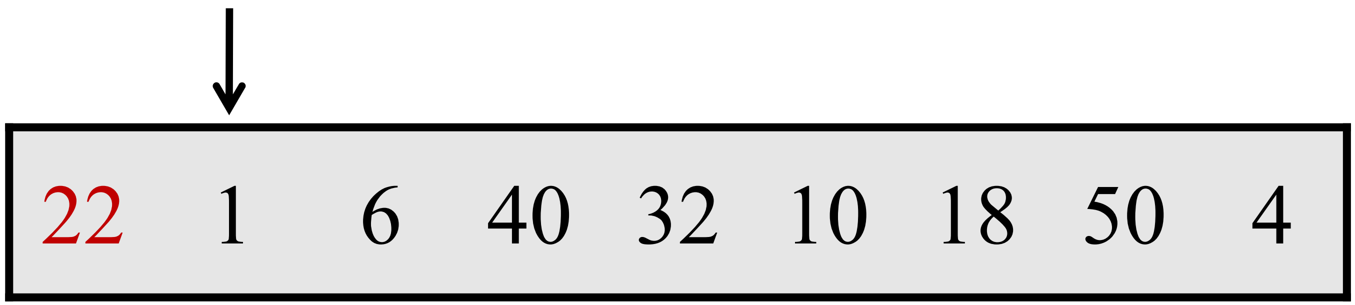
f. I don’t know.

ARCHIPELAGO

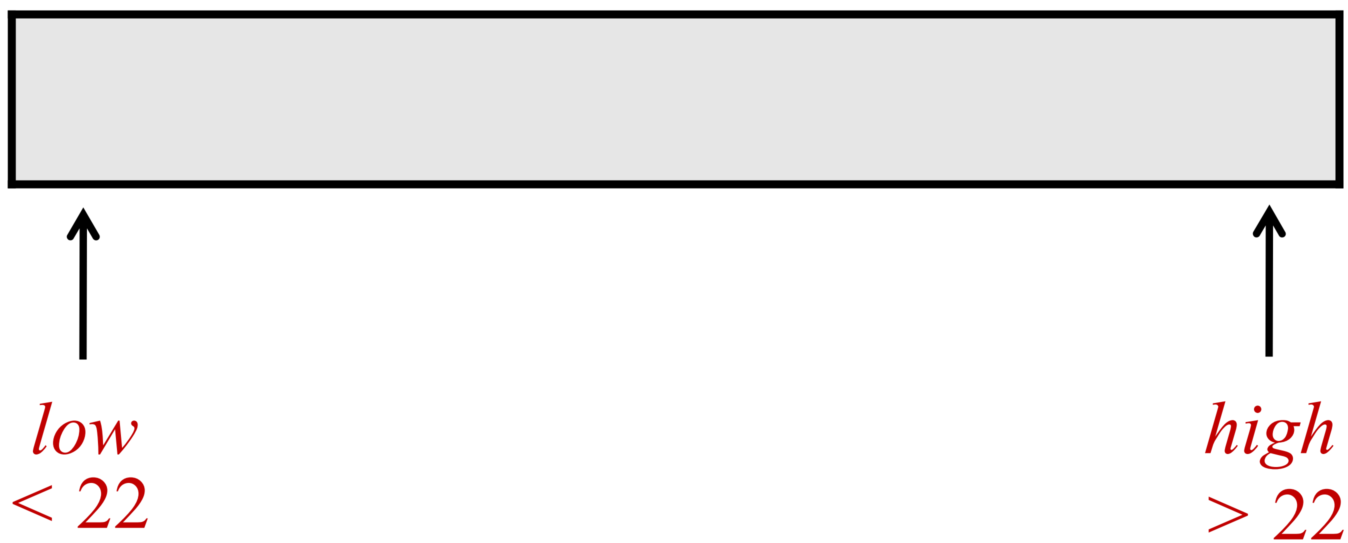
is open

Partitioning an Array

Example: partition around 22

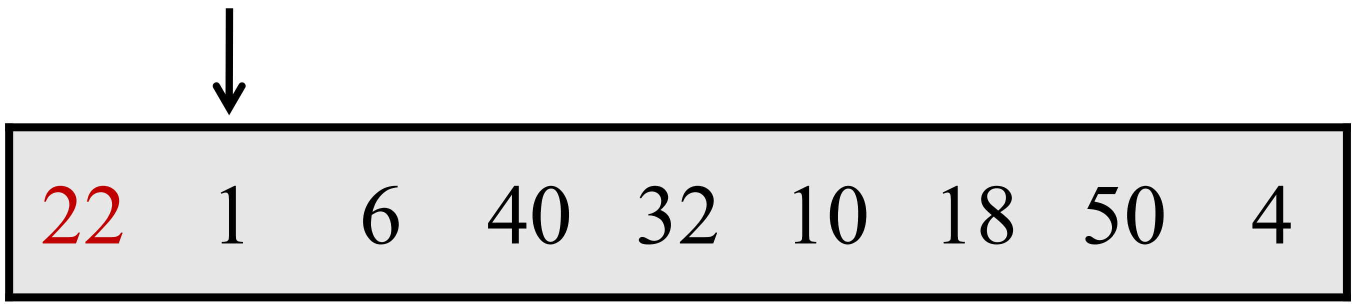


Output array:

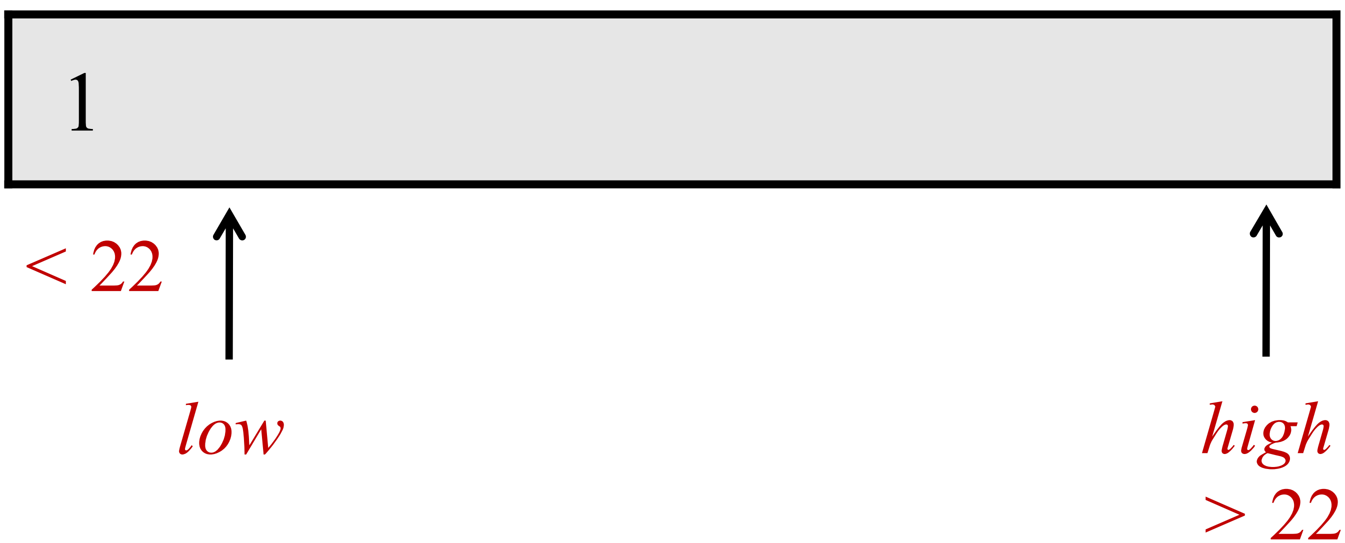


Partitioning an Array

Example: partition around 22

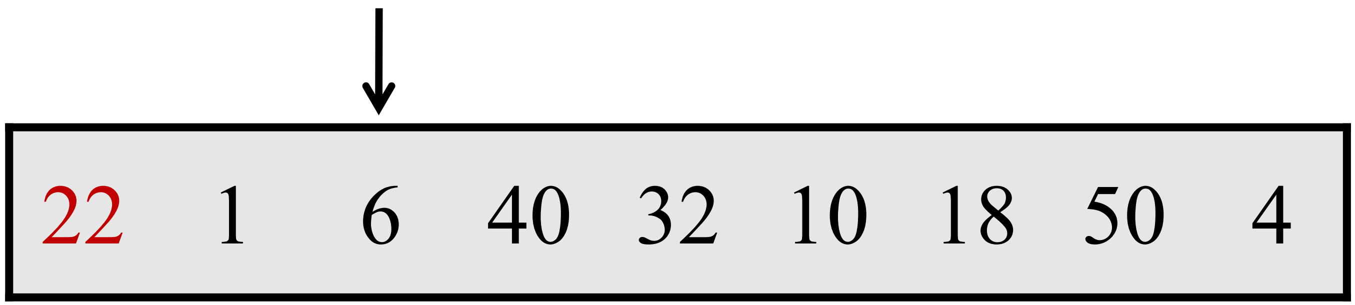


Output array:

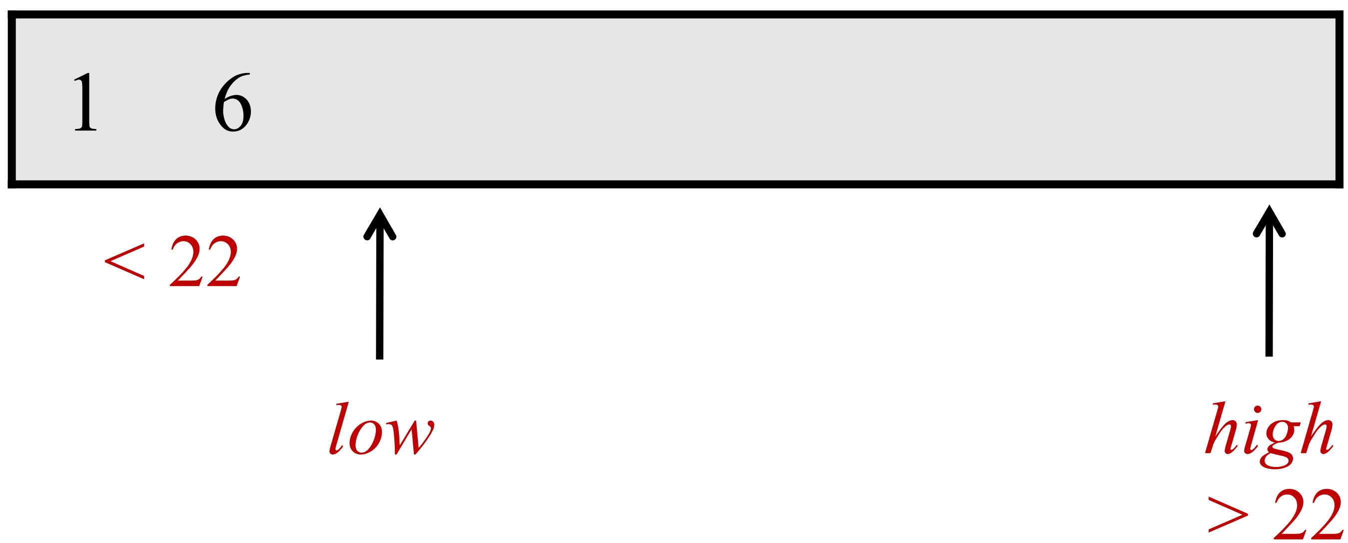


Partitioning an Array

Example: partition around 22

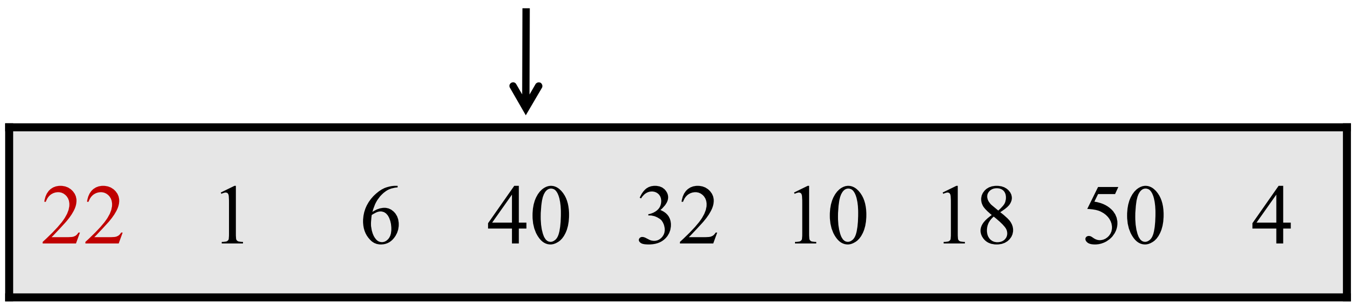


Output array:

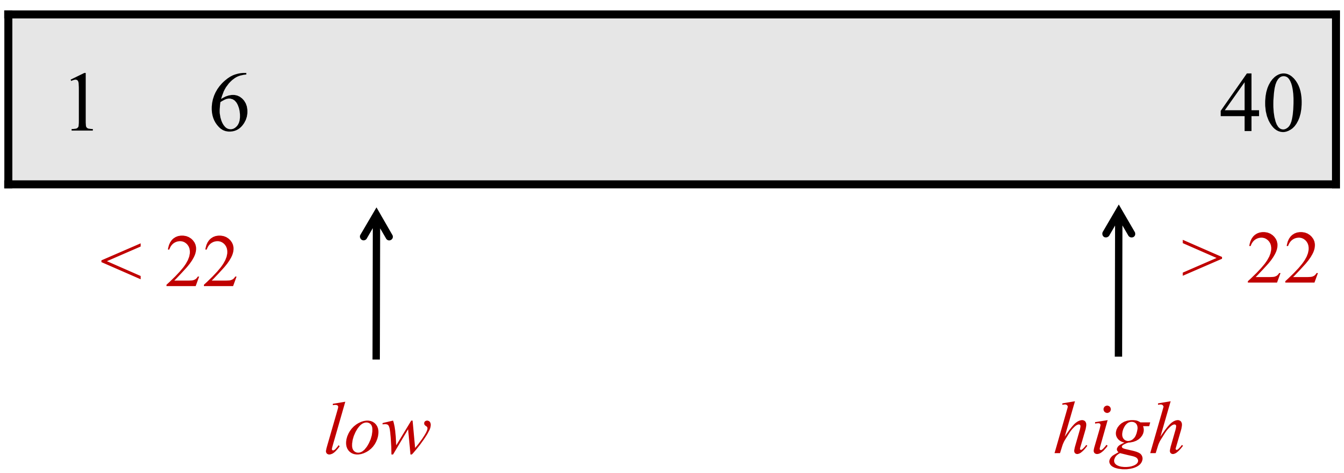


Partitioning an Array

Example: partition around 22

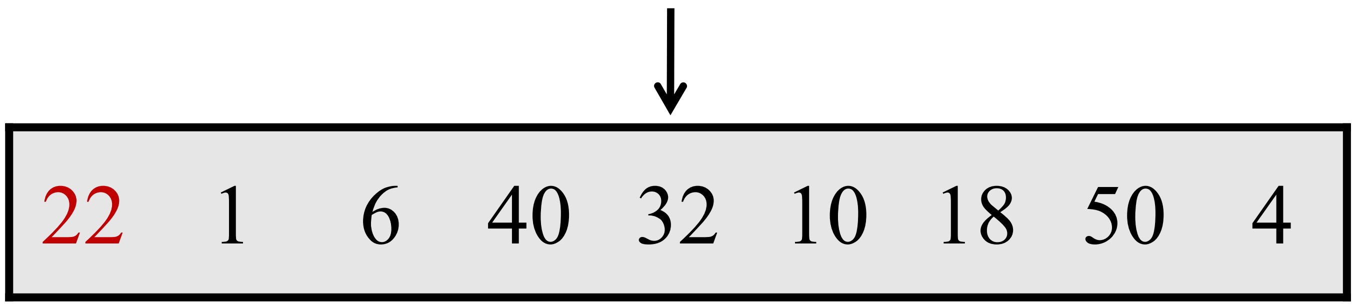


Output array:

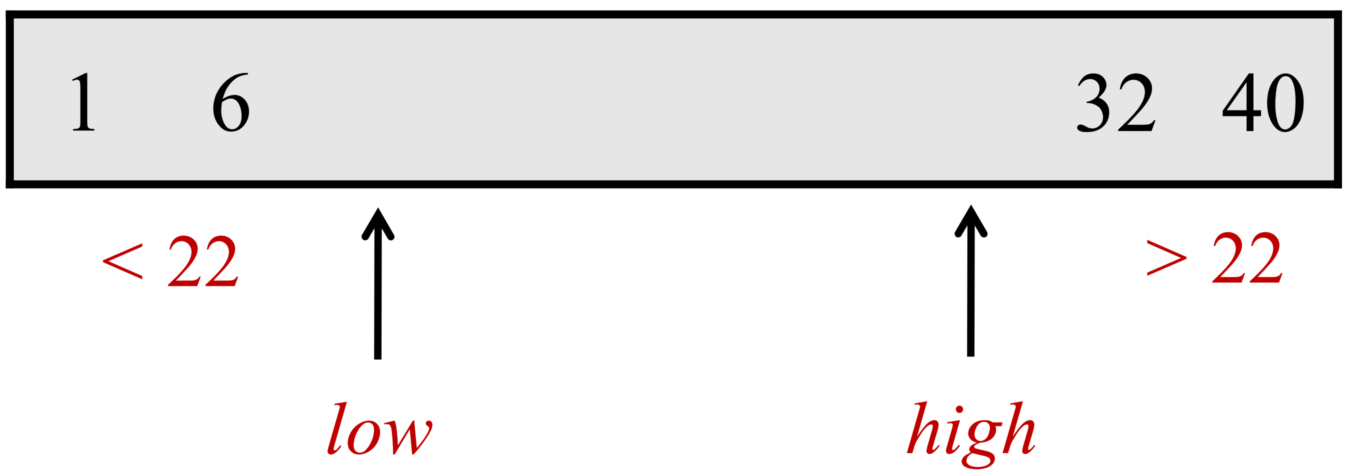


Partitioning an Array

Example: partition around 22



Output array:



Partitioning an Array

Example: partition around 22

22	1	6	40	32	10	18	50	4

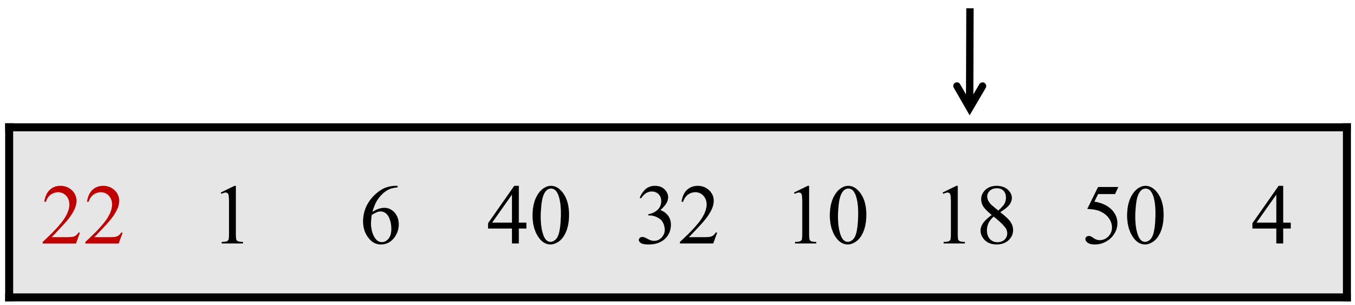
Output array:

1	6	10					32	40

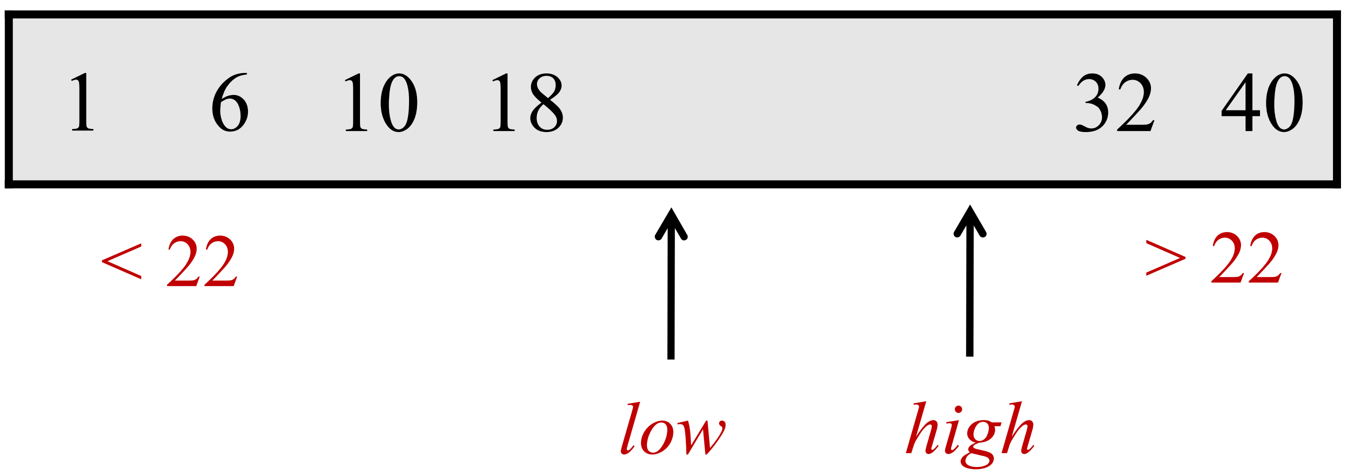
< 22   *low*   *high*   > 22

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:

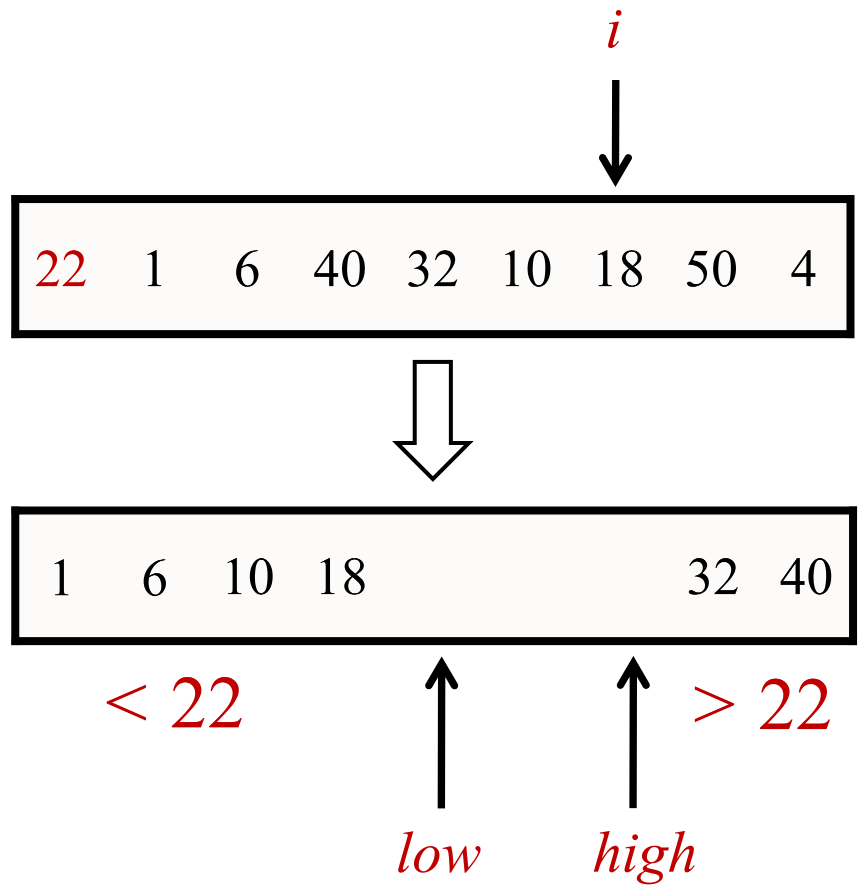
1	6	10	18	4	22	50	32	40

< 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?

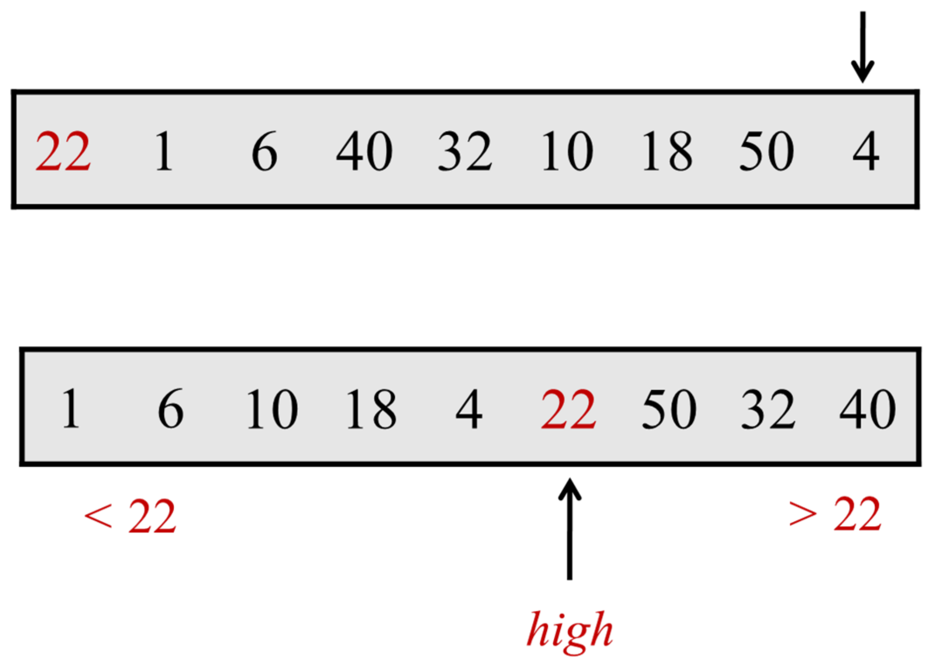
1. O(log n)
2. O(n) ✓
3. O(n log n)
4. O(n^2)
5. I have no idea.

ARCHIPELAGO

is open

Any bugs?

Anything that can be improved?

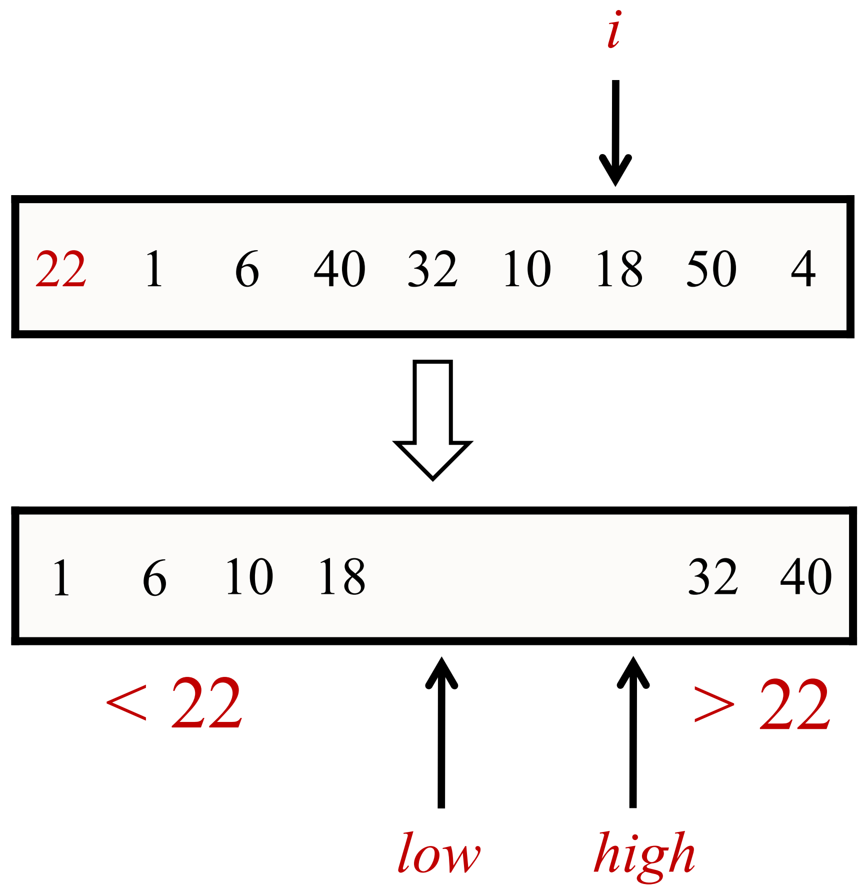


ARCHIPELAGO

is open

```
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

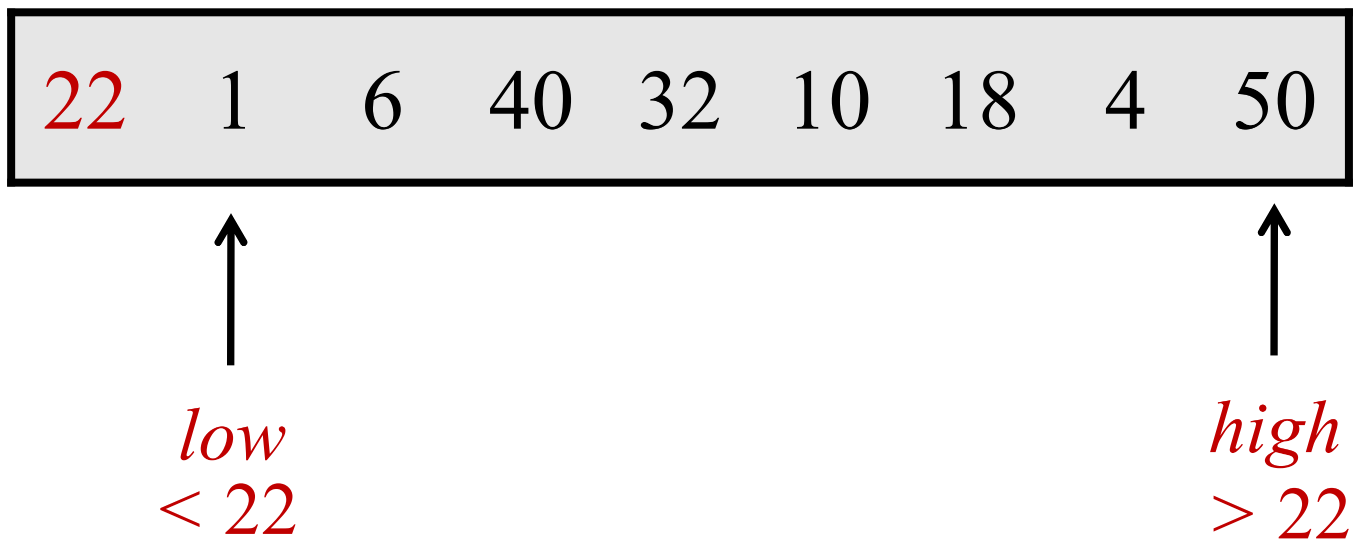
high
> 22

Move until it's bigger than the pivot

Move until it's less than the pivot

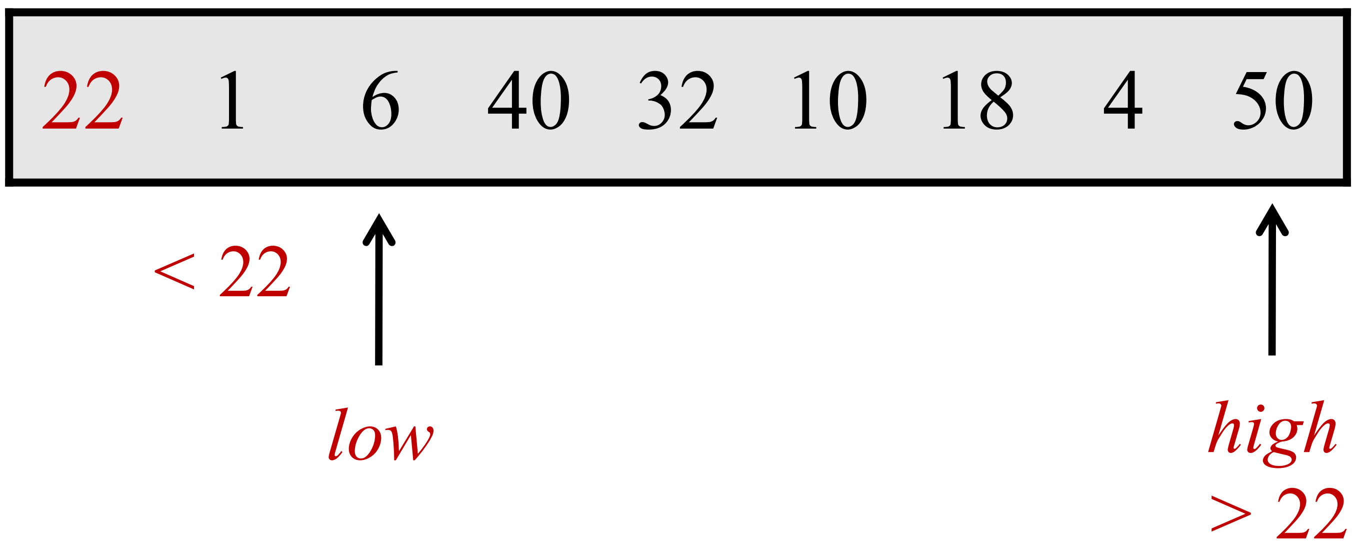
Partitioning an Array

Example: partition around 22



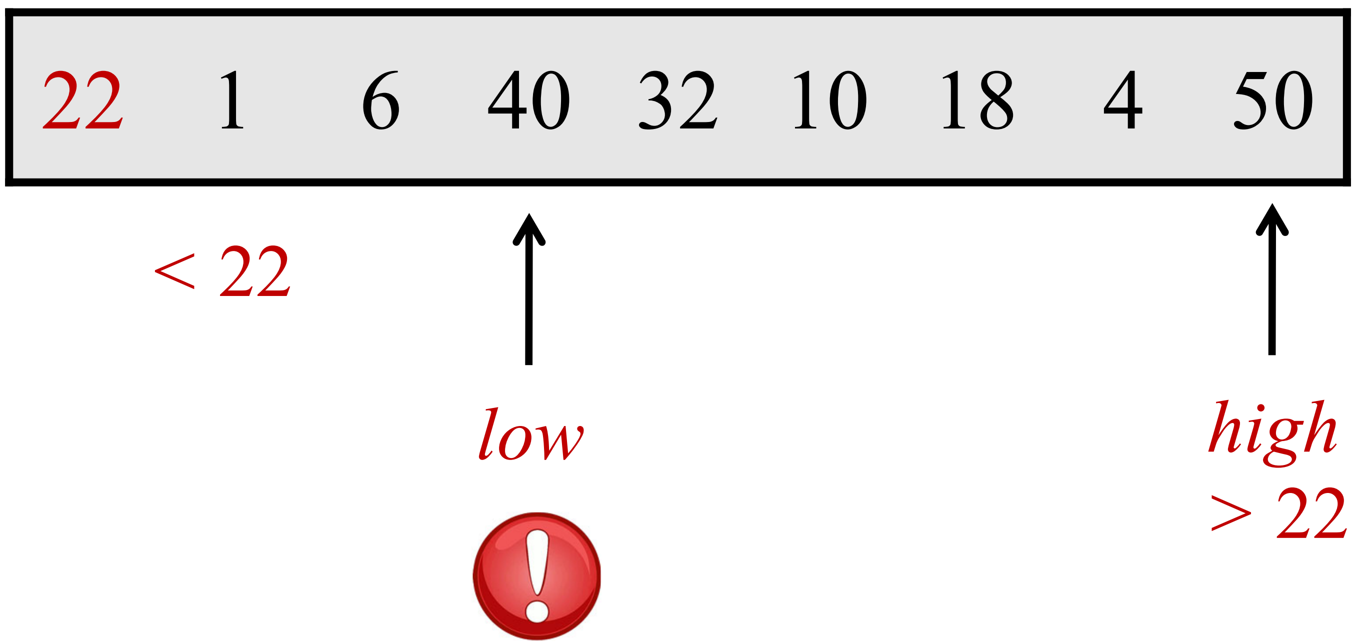
Partitioning an Array

Example: partition around 22



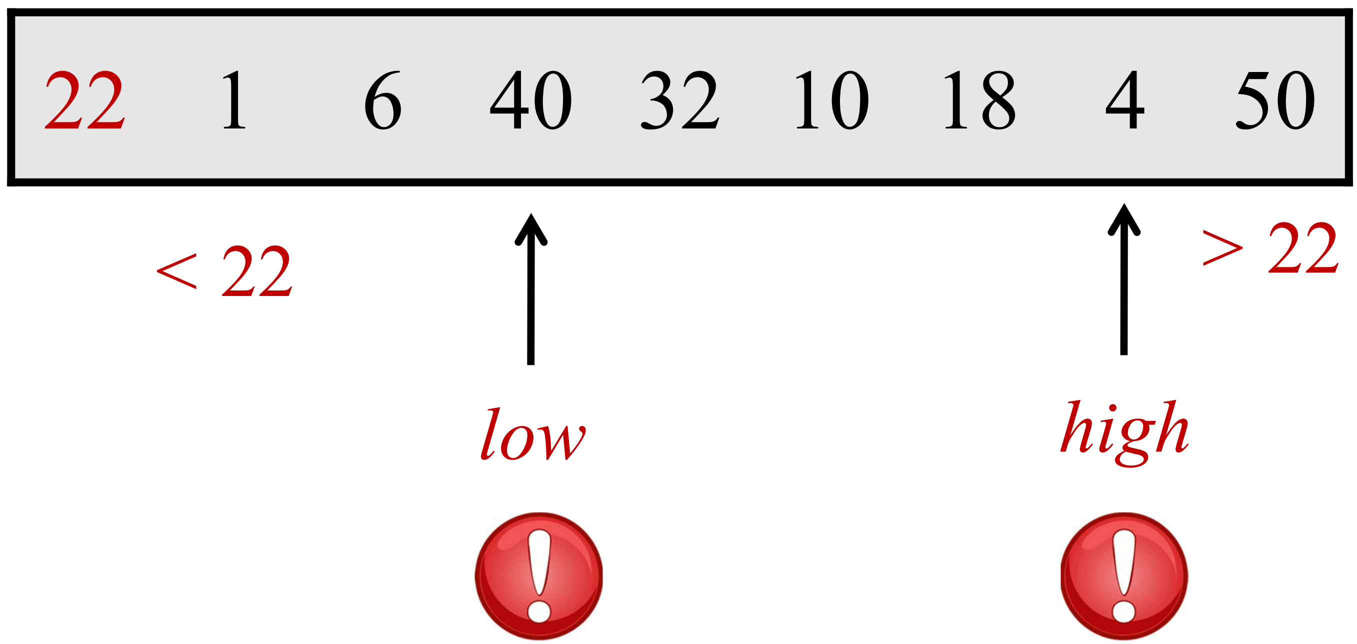
Partitioning an Array

Example: partition around 22



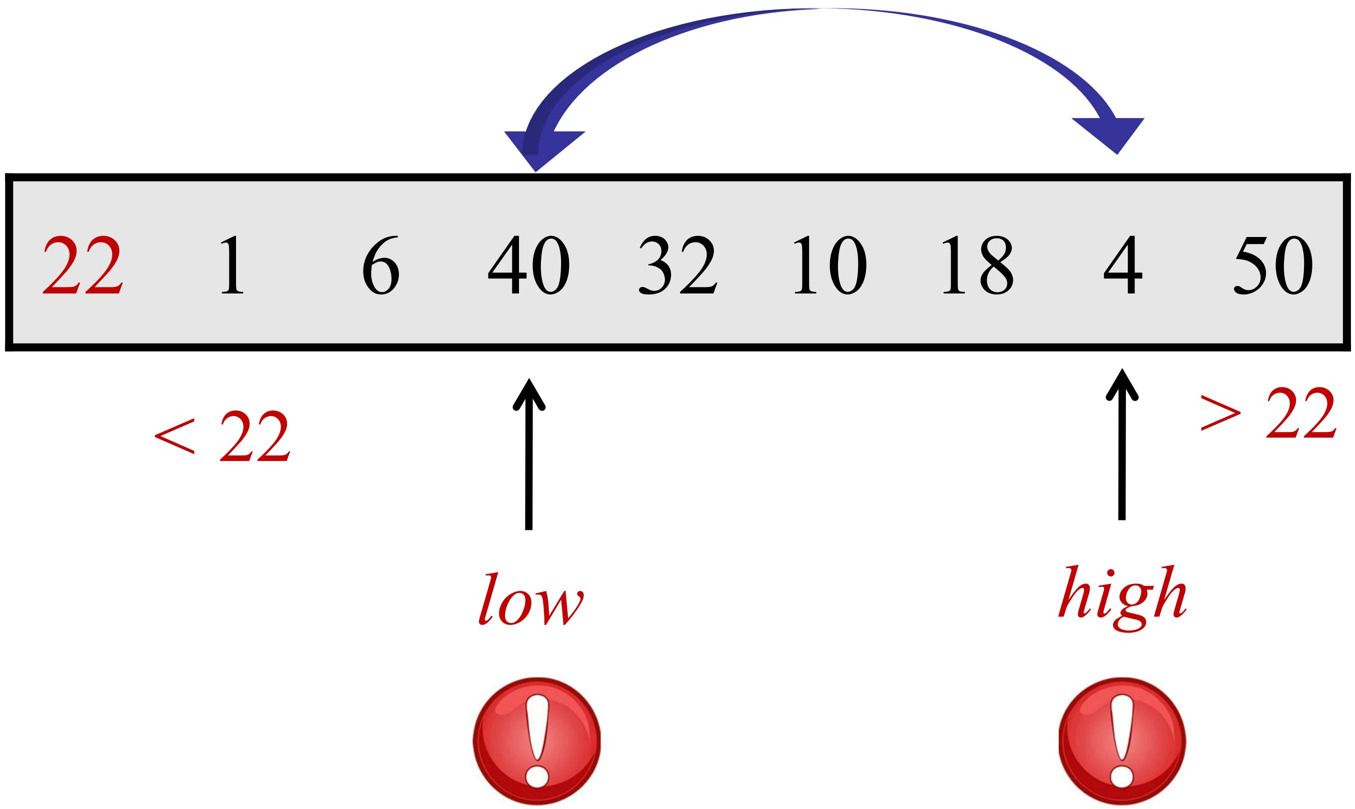
Partitioning an Array

Example: partition around 22



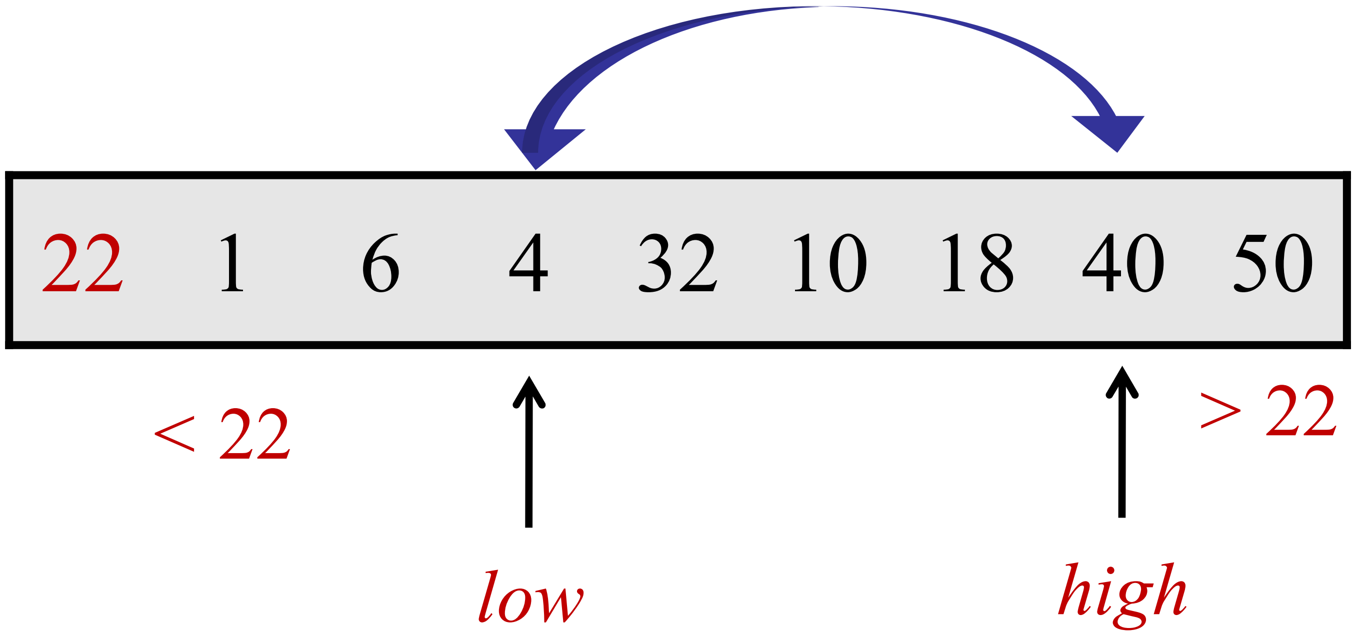
Partitioning an Array

Example: partition around 22



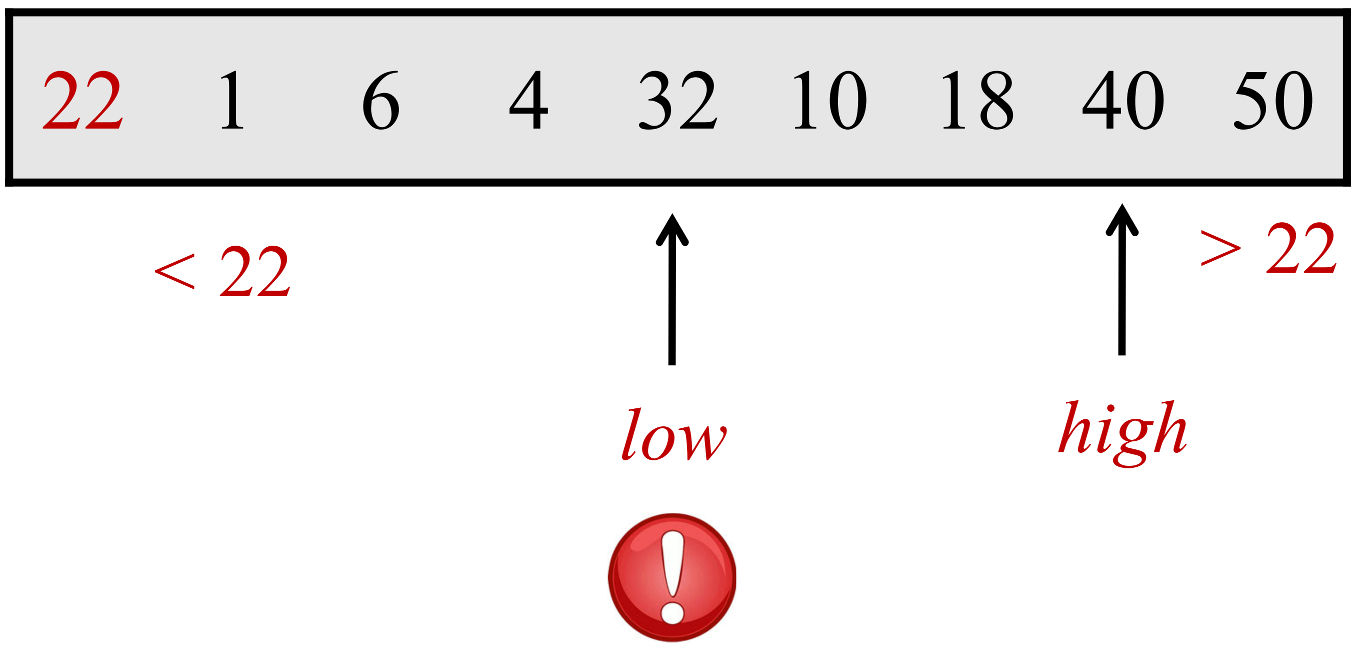
Partitioning an Array

Example: partition around 22



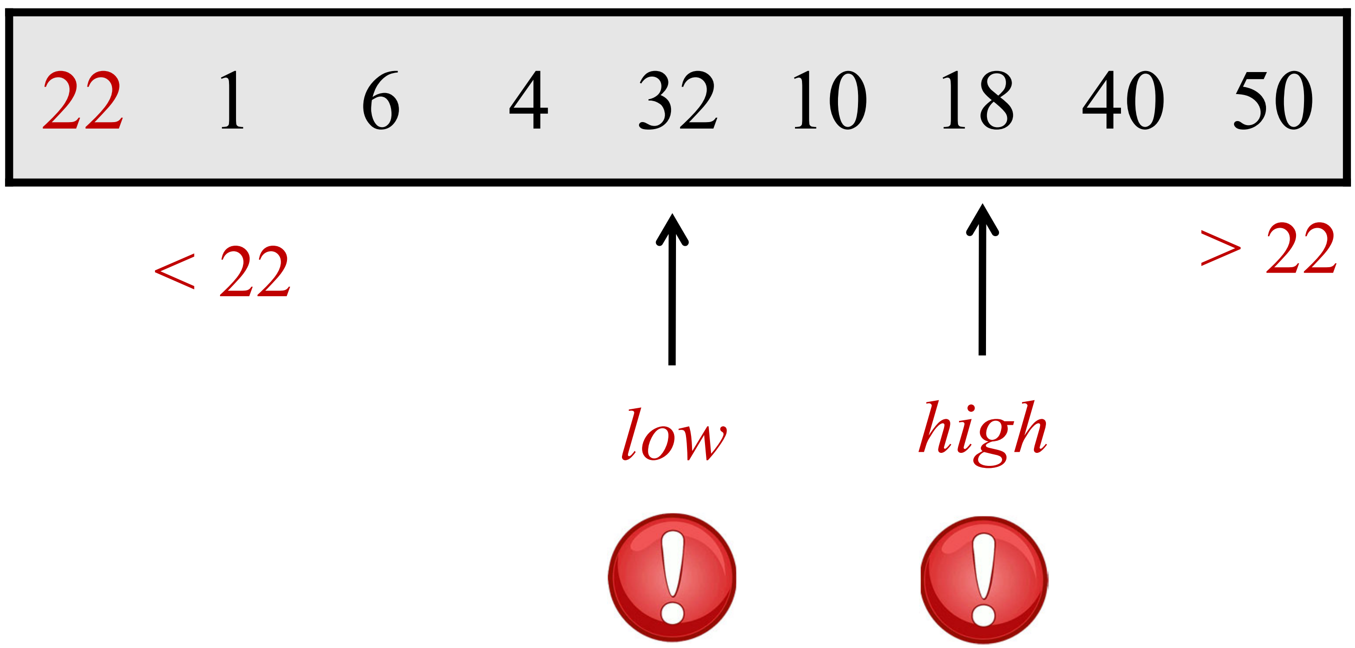
Partitioning an Array

Example: partition around 22



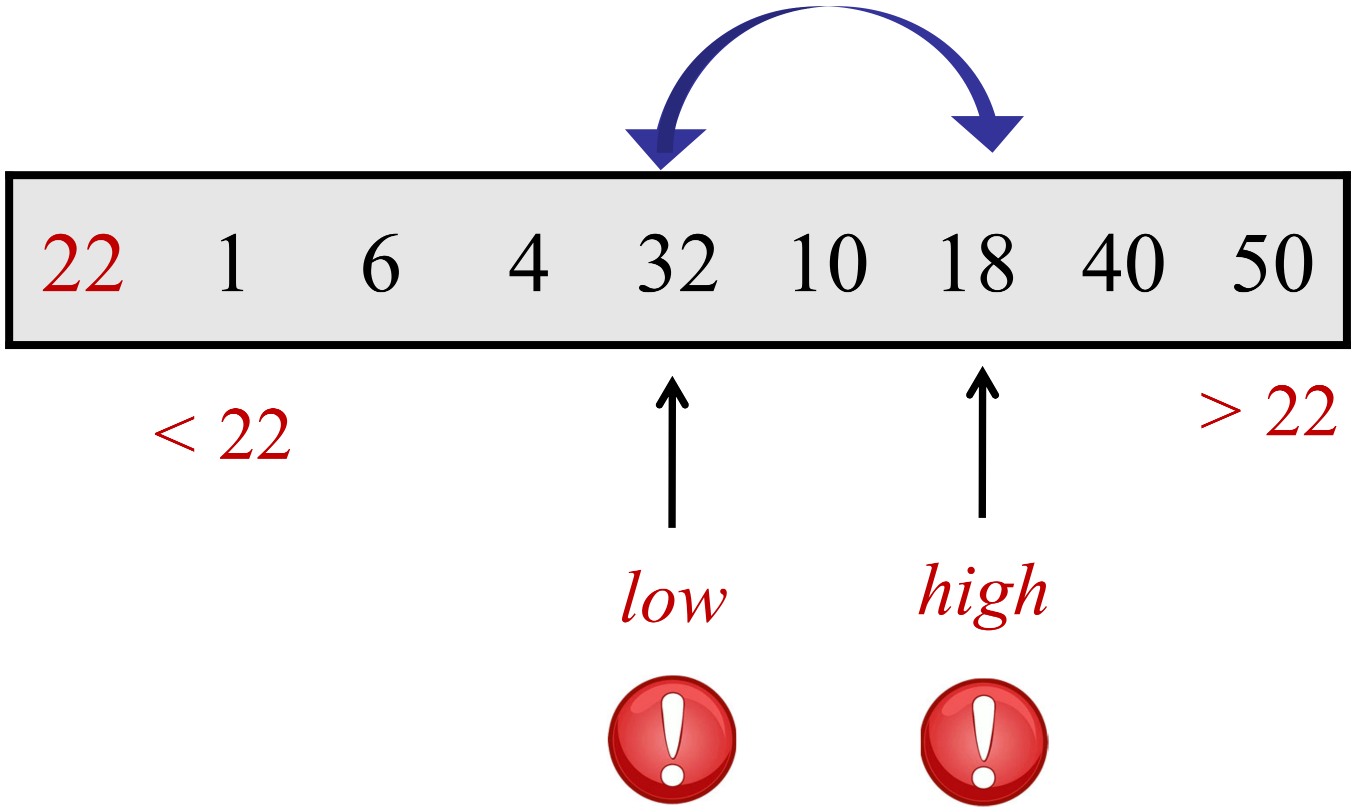
Partitioning an Array

Example: partition around 22



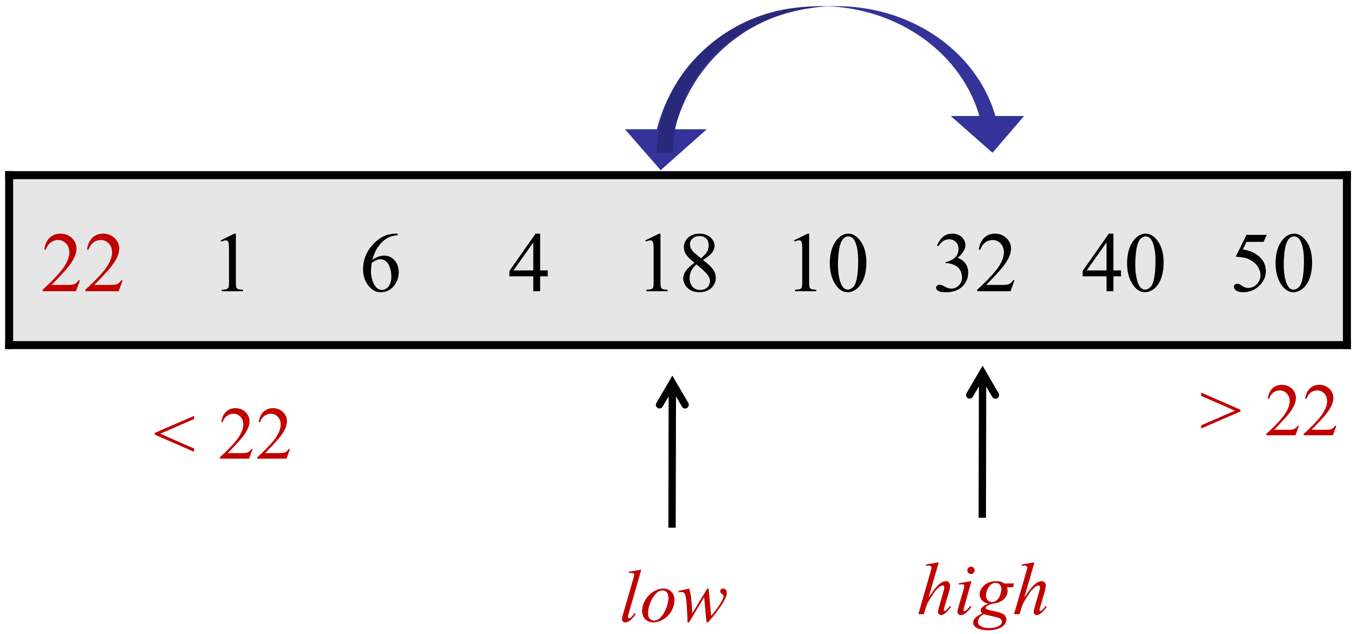
Partitioning an Array

Example: partition around 22



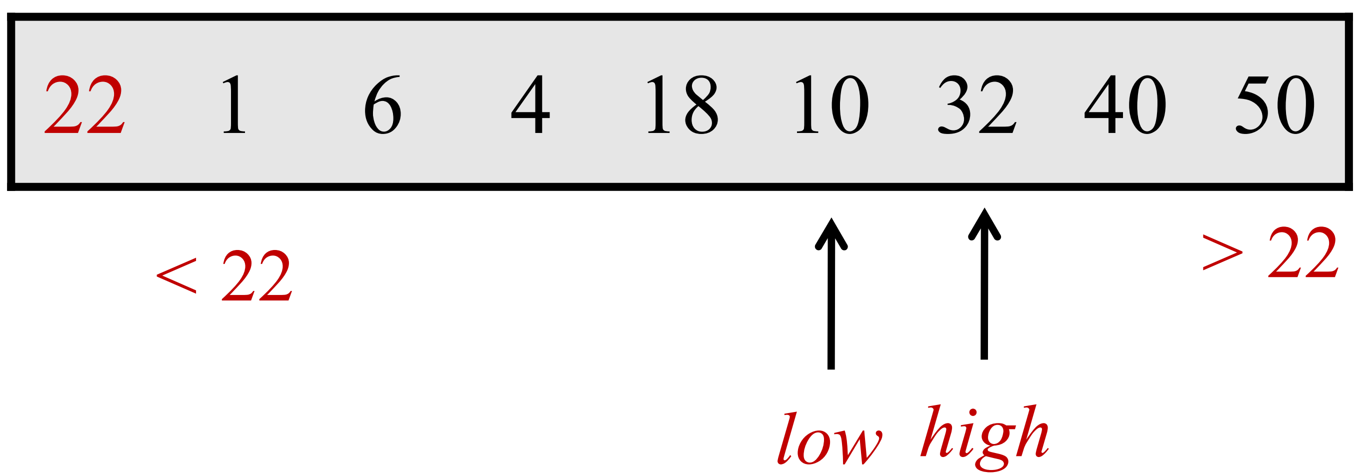
Partitioning an Array

Example: partition around 22



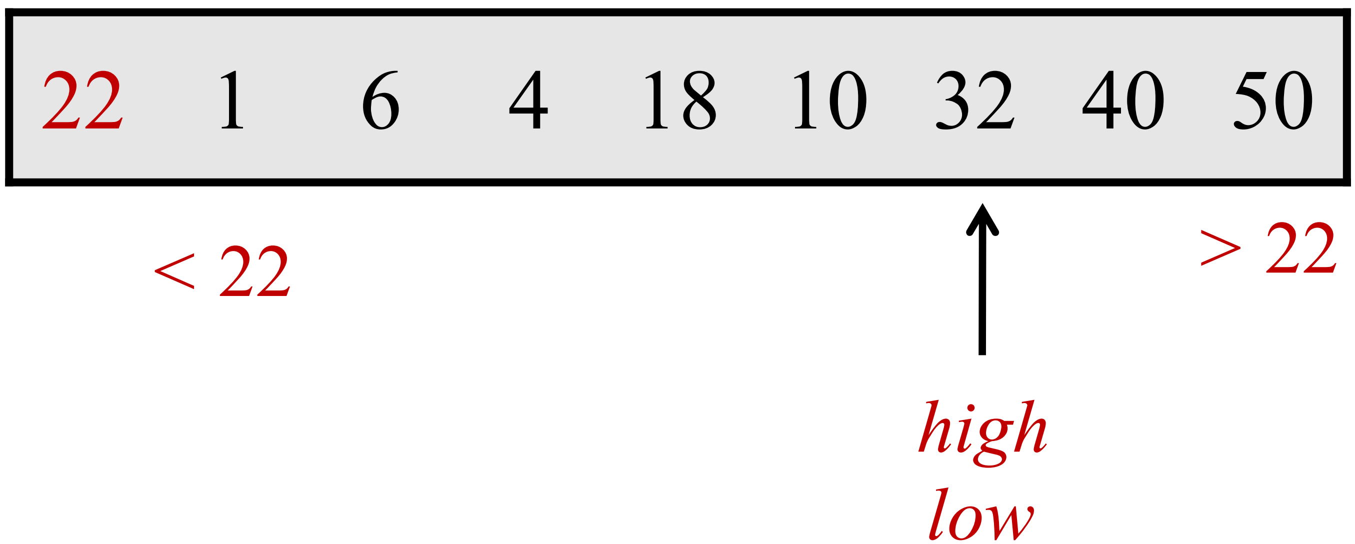
Partitioning an Array

Example: partition around 22



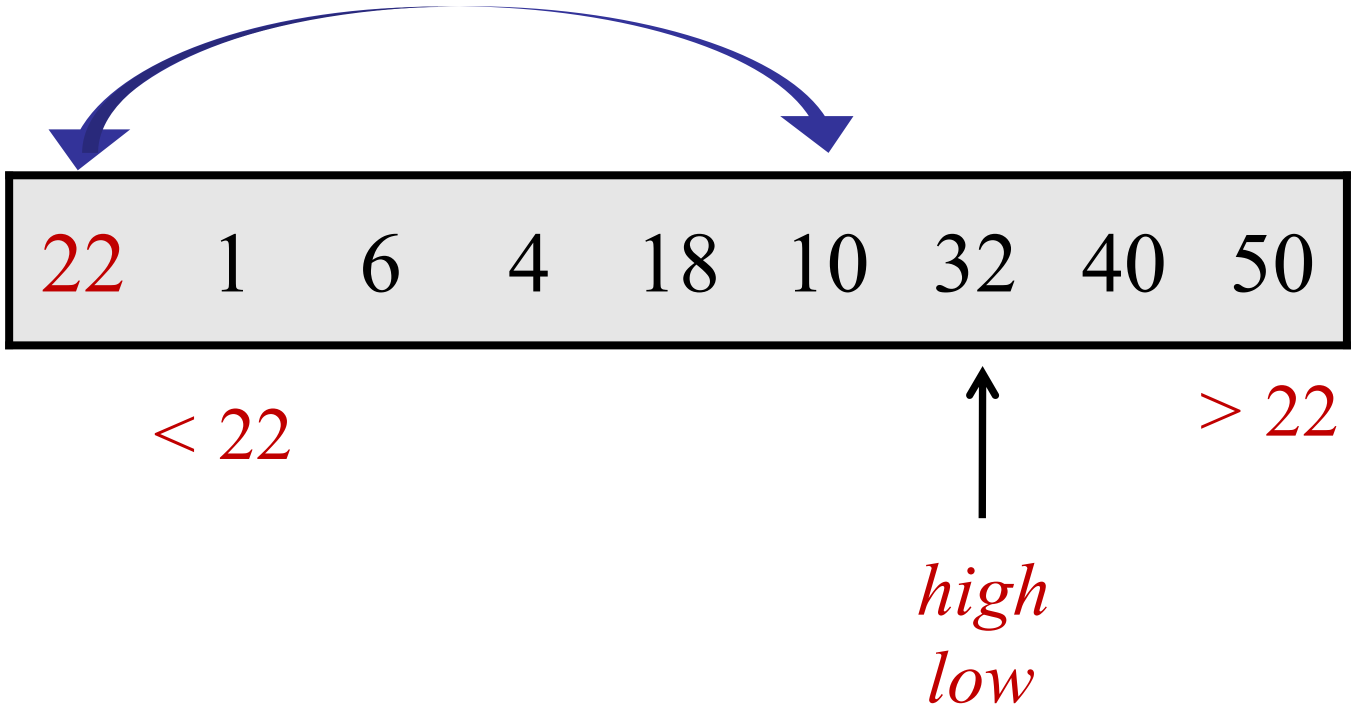
Partitioning an Array

Example: partition around 22



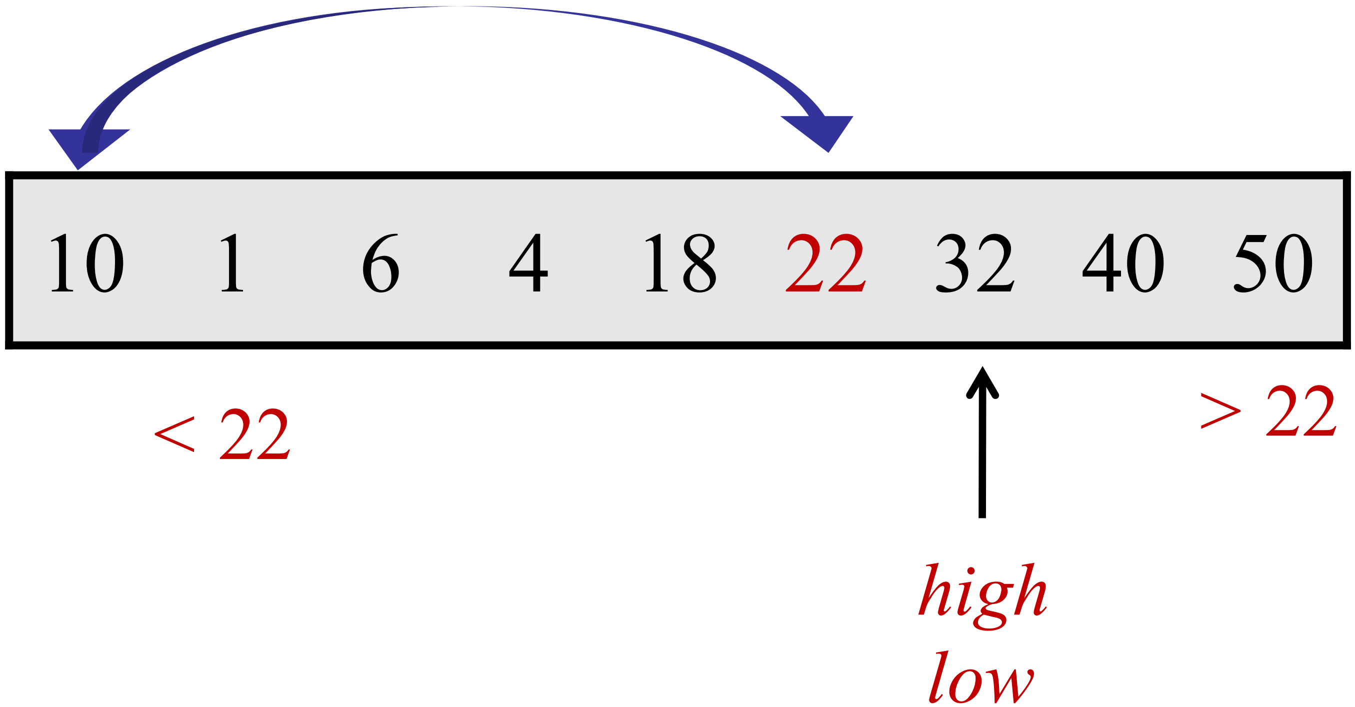
Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22



```
partition(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.

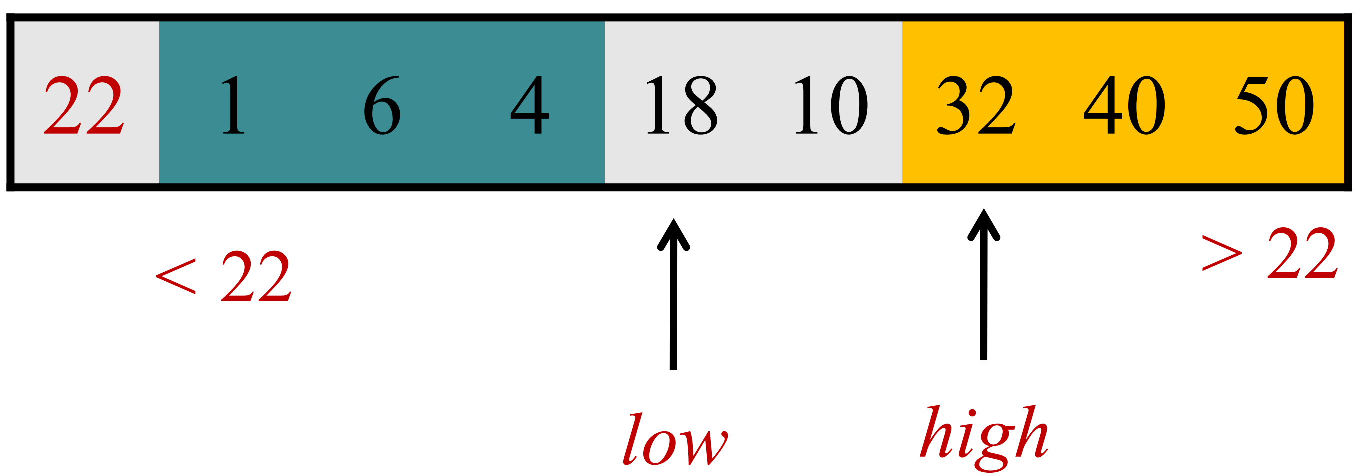
```
partition(A[1..n], n, pIndex)          // Assume no duplicates, n>1
    pivot = A[pIndex];                 // pIndex is the index of pivot
    swap(A[1], A[pIndex]);             // store pivot in A[1]
    low = 2;                           // start after pivot in A[1]
    high = n+1;            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

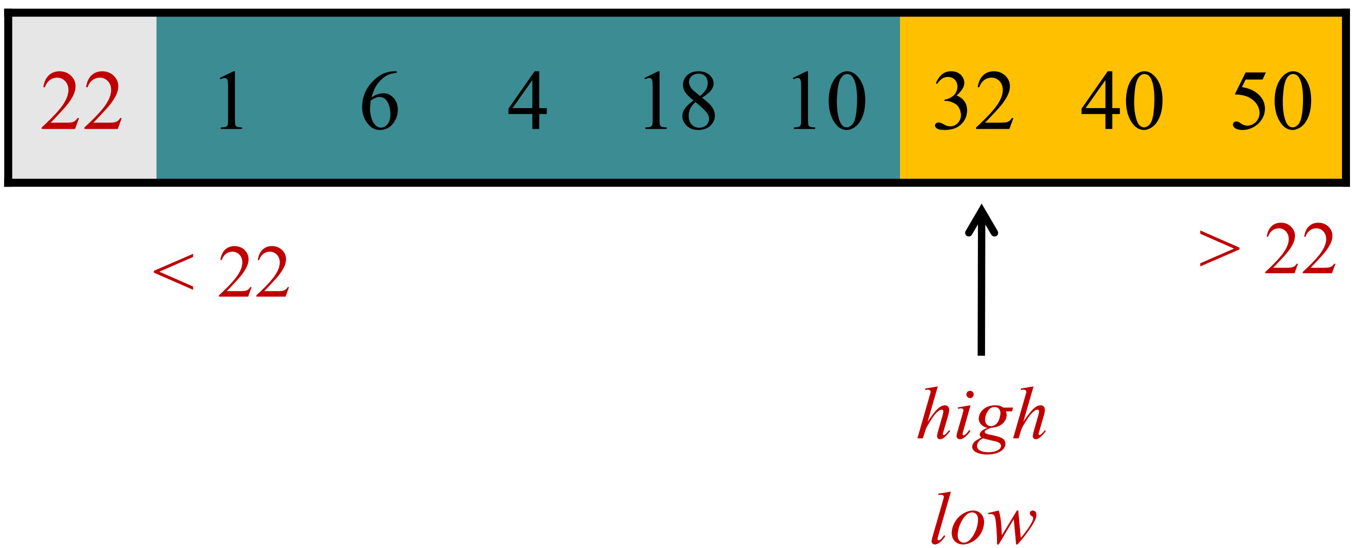


Partition

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

Today: Sorting, Part II

QuickSort

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