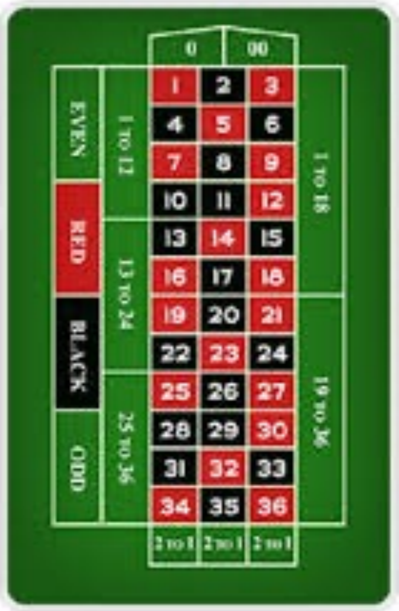
CS2040S
Data Structures and Algorithms

Welcome!

PotW: Gambling for Profit?

Alice

- Begins with $100
- Bets $1 each time.
- Each bet has a 51% chance of winning.
- On win: +1
On lose: -1



Bob

- Begins with $100
- Bets $1 each time.
- Each bet has a 49% chance of winning.
- On win: +1
On lose: -1

PotW: Gambling for Profit?

Alice

- Begins with $100
- Bets $1 each time.
- Each bet has a 51% chance of winning.
- On win: +1
  On lose: -1

Bob

- Begins with $100
- Bets $1 each time.
- Each bet has a 49% chance of winning.
- On win: +1
  On lose: -1

PotW: Gambling for Profit?

Alas, both Alice and Bob lose all their money (gambling at two different tables). Who is more likely to go bankrupt first (conditioned on both losing)?

Alice

- Begins with $100
- Bets $1 each time.
- Each bet has a 51% chance of winning.
- On win: +1
  On lose: -1

Bob

- Begins with $100
- Bets $1 each time.
- Each bet has a 49% chance of winning.
- On win: +1
  On lose: -1

PotW: Gambling for Profit?

Alas, both Alice and Bob lose all their money (gambling at two different tables). Who is more likely to go bankrupt first?

Alice

- Begins with $100
- Bets $1 each time.
- Each bet has a 51% chance of winning.
- On win: +1
  On lose: -1

Hints:

- Bayes Rule!
- Alice eventually goes bankrupt w.p. $(0.49/0.51)^{100}$.
- For every sequence where Alice loses, you can construct an inverted sequence where Bob loses.

Bob

- Begins with $100
- Bets $1 each time.
- Each bet has a 49% chance of winning.
- On win: +1
  On lose: -1

Sorting, Part I

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

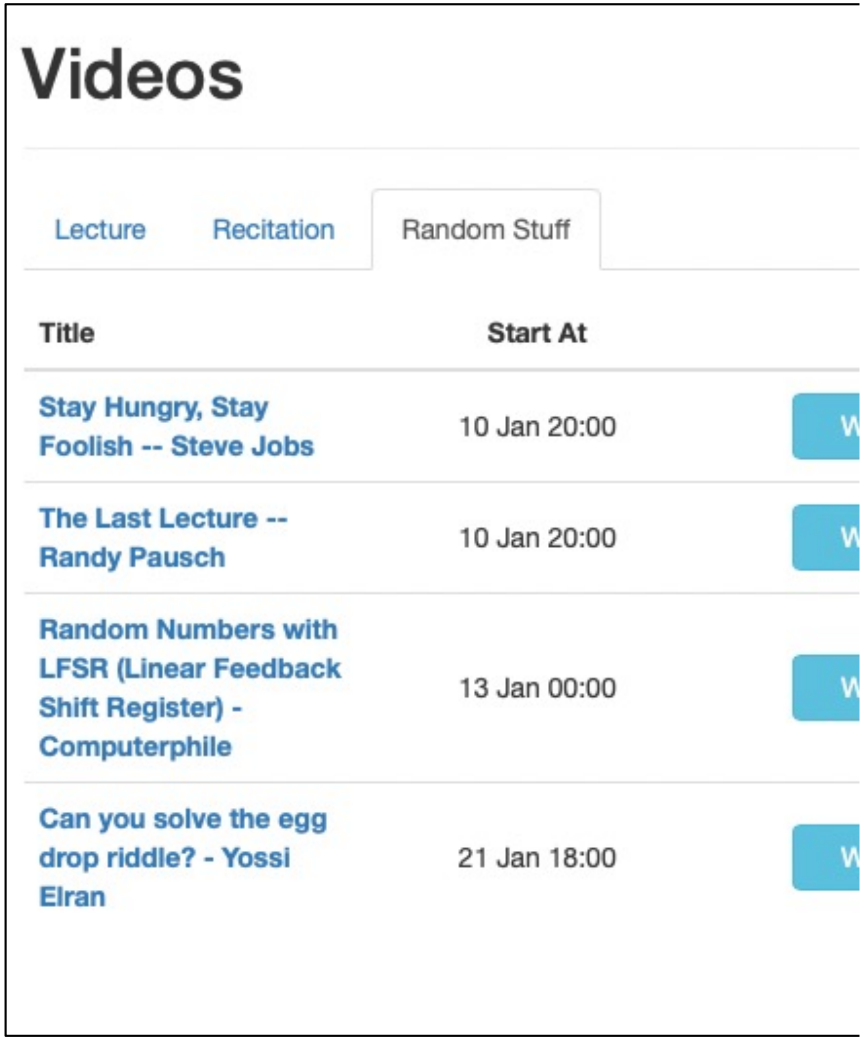
- Running time
- Space usage
- Stability

Admin

TGIF Video of the Weekend

- Random video posted each week
- Selected by the tutor team as something "fun"
- Sometimes related to class, sometimes a little bit different
- Not just another lecture…

(Nominate videos to your tutor!)



Videos

Title	Start At
Stay Hungry, Stay Foolish -- Steve Jobs	10 Jan 20:00
The Last Lecture -- Randy Pausch	10 Jan 20:00
Random Numbers with LFSR (Linear Feedback Shift Register) - Computerphile	13 Jan 00:00
Can you solve the egg drop riddle? - Yossi Elran	21 Jan 18:00

Puzzle: Slowest Sorting Algorithm

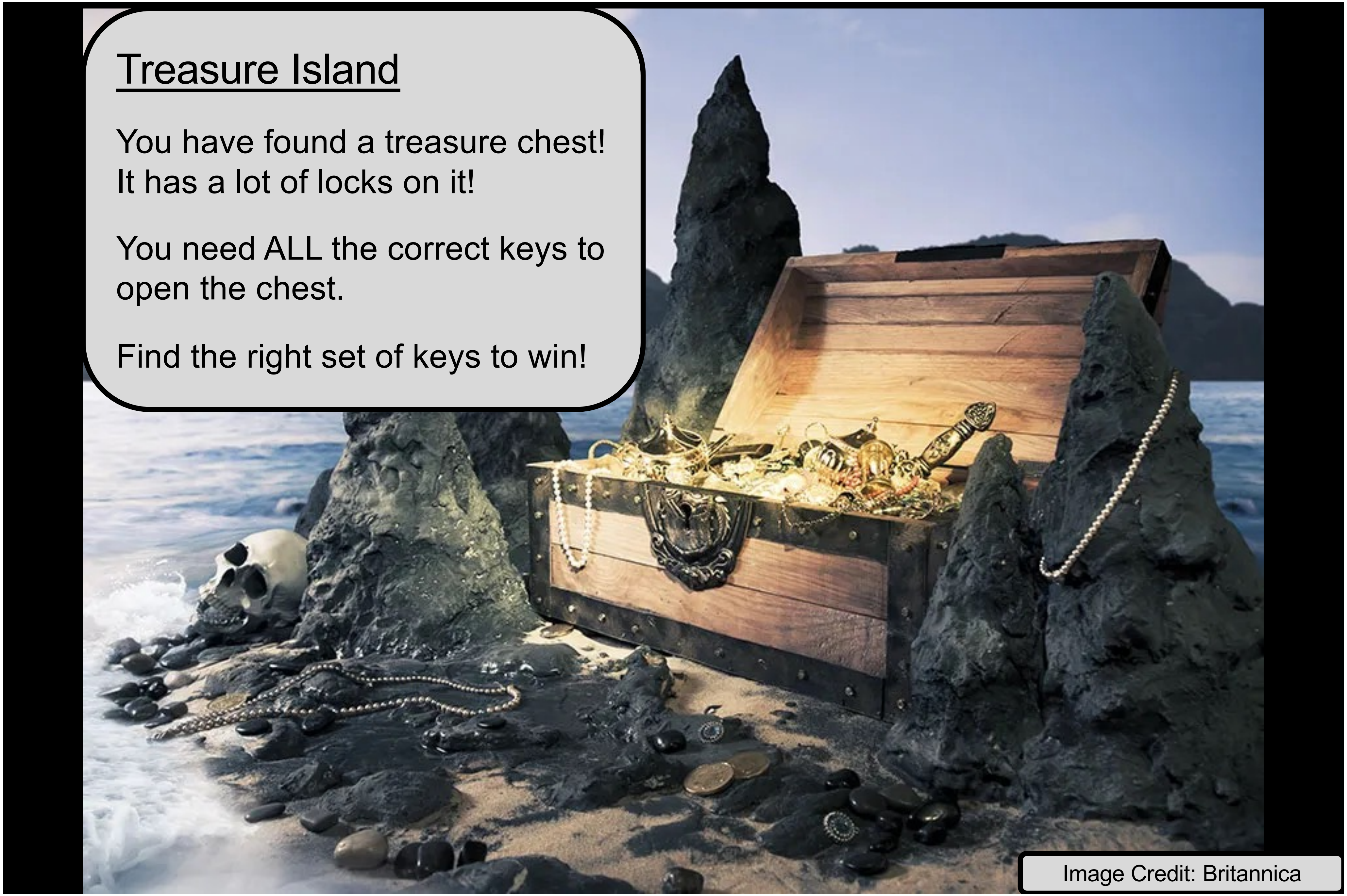
What is the *slowest* sorting algorithm you can think of?

Slower than BogoSort…

But must always sort correctly…

Hint: recursion can be a powerful source of slowness!

Contest: Deadline Feb. 3



Treasure Island

You have found a treasure chest! It has a lot of locks on it!

You need ALL the correct keys to open the chest.

Find the right set of keys to win!

Image Credit: Britannica

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

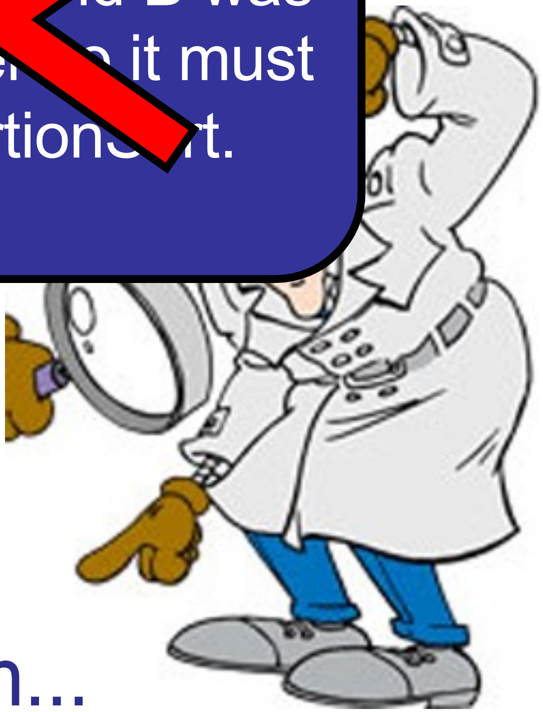
- Absolute speed is not a good reason…

Problem Set 3

Sorting Detective

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

~~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:

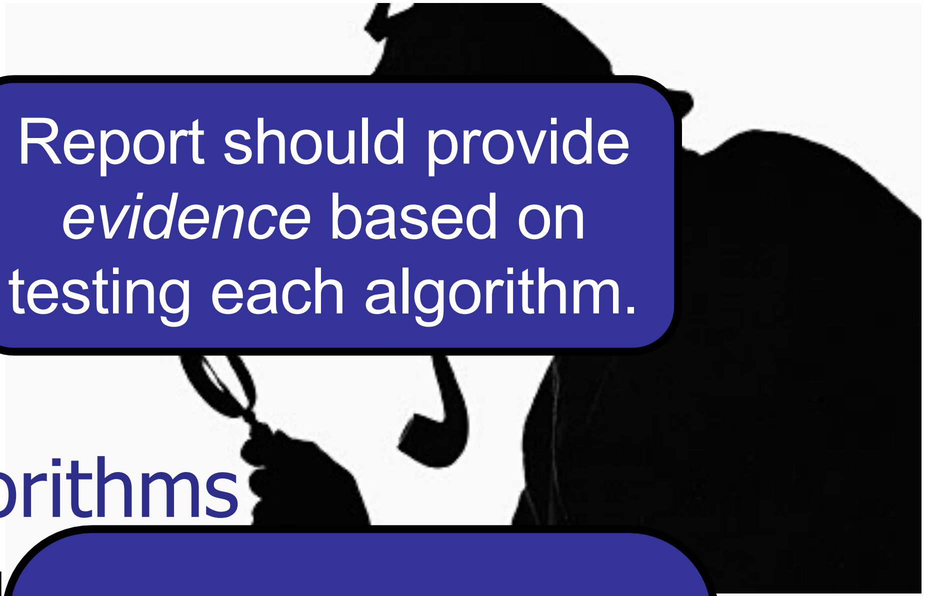
mance

ability

erformance on special inputs

- Absolute speed is not a good reason…

Problem Set 3



Report should provide *evidence* based on testing each algorithm.

Sorting Detective

- Six suspicious sorting algorithms
  - Investigate the mysterious sorting
  - 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…

I ran algorithm A on these sets of arrays and from the results, I discovered that….

Sorting

Problem definition:

Input: array A[1..n] of words / numbers

Output: array B[1..n] that is a permutation of A such that:

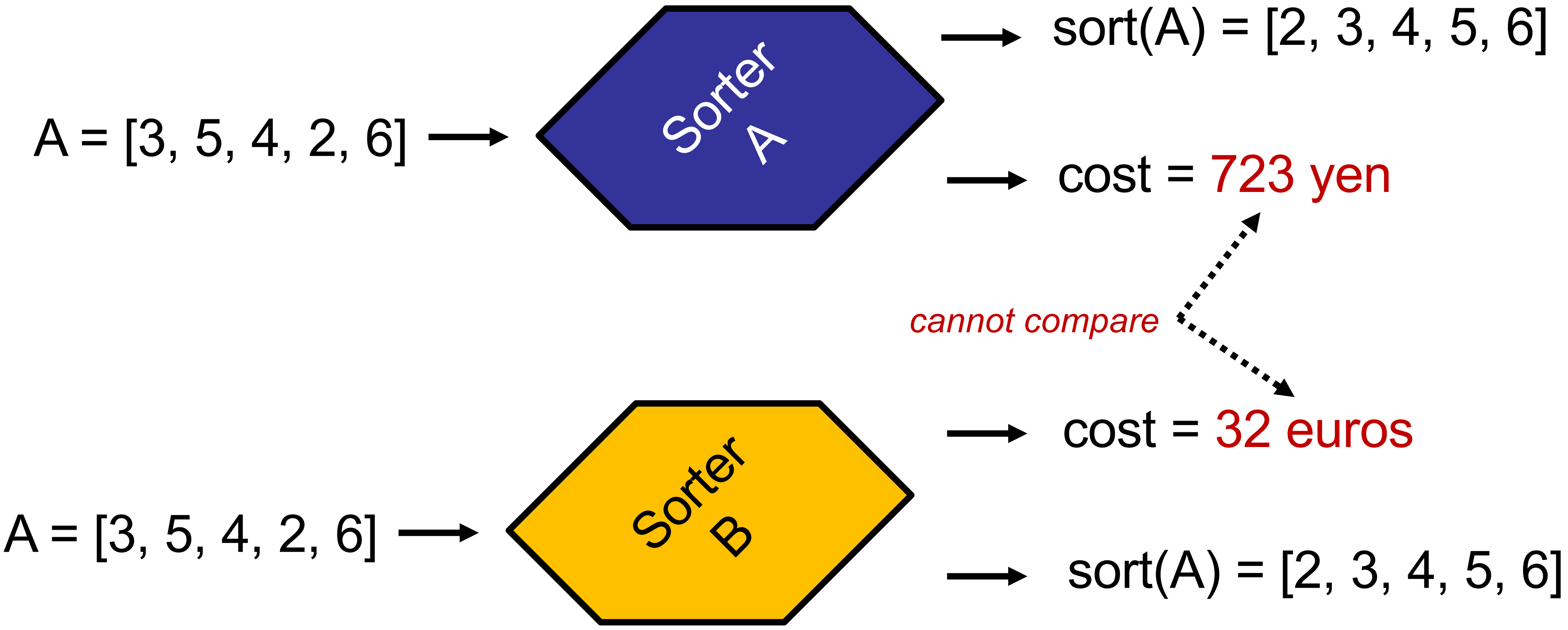
$B[1] \leq B[2] \leq \ldots \leq B[n]$

Example:

A = [9, 3, 6, 6, 6, 4] → [3, 4, 6, 6, 6, 9]

Problem Set 3

Sorting Detective



A = [3, 5, 4, 2, 6] → Sorter A → sort(A) = [2, 3, 4, 5, 6]

→ cost = 723 yen

cannot compare

A = [3, 5, 4, 2, 6] → Sorter B → cost = 32 euros

→ sort(A) = [2, 3, 4, 5, 6]

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

Warning: we cover QuickSort on Wednesday…

Sorting, Part I

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

Properties of Sorting Algorithms

Time complexity

Properties of Sorting Algorithms

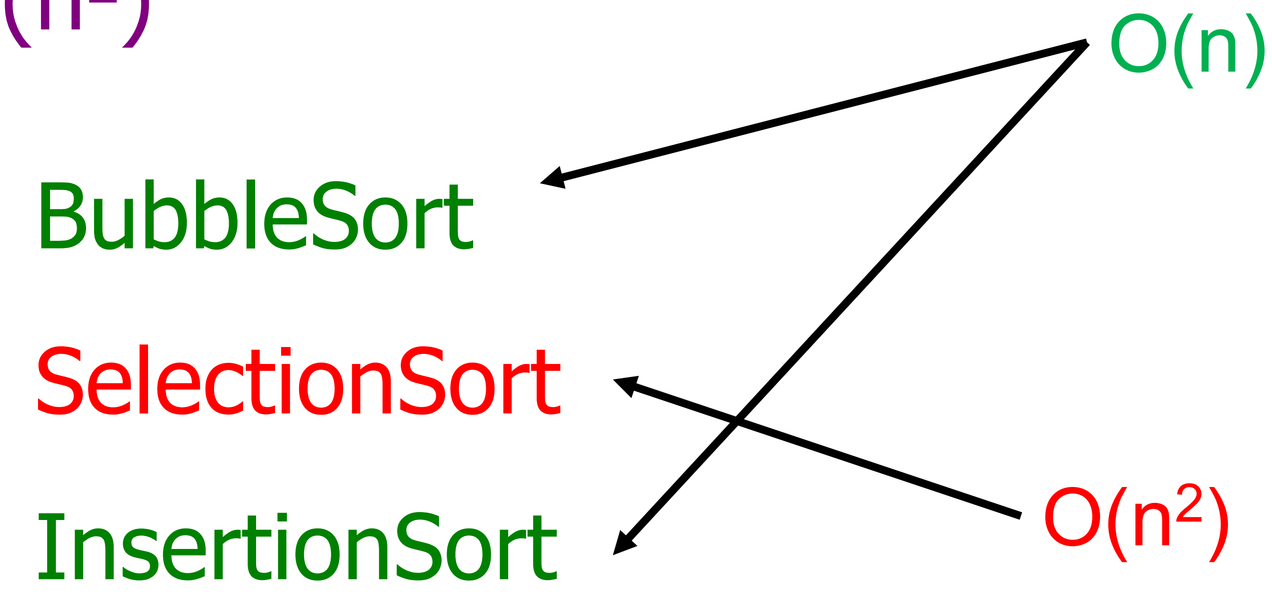
Time complexity

- Worst case: $O(n^2)$
- Sorted list:

Properties of Sorting Algorithms

Time complexity

- Worst case: $O(n^2)$
- Sorted list:



BubbleSort

SelectionSort

InsertionSort

$O(n)$

$O(n^2)$

How expensive is it to sort:

[1, 2, 3, 4, 5, **7**, **6**, 8, 9, 10]

Bubblesort?

SelectionSort?

InsertionSort?

How expensive is it to sort:

[1, 2, 3, 4, 5**, 7, 6**, 8, 9, 10]

BubbleSort and InsertionSort are fast.

SelectionSort is slow.

Challenge of the Day:

Find a permutation of [1..n] where:

- BubbleSort is slow.
- InsertionSort is fast.

Or explain why no such sequence exists.

Properties of Sorting Algorithms

Moral:

Different sorting algorithms have different inputs that they are good or bad on.

All $O(n^2)$ algorithms are not the same.

Properties of Sorting Algorithms

Space complexity

How much space does a sorting algorithm need?

- Worst case: O(n)

Properties of Sorting Algorithms

Space complexity

- Worst case: O(n)

- In-place sorting algorithm:
  - Only O(1) extra space needed.
  - All manipulation happens within the array.

So far:

All sorting algorithms we have seen are in-place.

Subtle issue:

How do you count space?

- Maximum space every allocated at one time?
- Total space ever allocated.

Properties of Sorting Algorithms

Stability

What happens with repeated elements?

Key	1	2	5	3	4	5	6	7	8	9
Value	a	b	C	g	h	D	j	k	l	m

Databases often contain (key, value) pairs.

The key is an index to help organize the data.

Properties of Sorting Algorithms

Stability

What happens with repeated elements?

Key	1	2	**5**	3	4	**5**	6	7	8	9
Value	a	b	**C**	g	h	**D**	j	k	l	m

Two values have the same key!

Properties of Sorting Algorithms

Stability

What happens with repeated elements?

Key	1	2	**5**	3	4	**5**	6	7	8	9
Value	a	b	**C**	g	h	**D**	j	k	l	m

⇩ UNSTABLE

Key	1	2	3	4	**5**	**5**	6	7	8	9
Value	a	b	g	h	**D**	**C**	j	k	l	m

Properties of Sorting Algorithms

Stability: preserves order of equal elements

What happens with repeated elements?

Key	1	2	**5**	3	4	**5**	6	7	8	9
Data	a	b	**C**	g	h	**D**	j	k	l	m

STABLE

Key	1	2	3	4	**5**	**5**	6	7	8	9
Data	a	b	g	h	**C**	**D**	j	k	l	m

Which are stable?

A. BogoSort
B. BubbleSort
C. SelectionSort
D. InsertionSort

ARCHIPELAGO

is open

Which are stable?

A. BogoSort

B. BubbleSort

C. SelectionSort

D. InsertionSort

Not stable:
Random permutation
may swap elements!

Which are stable?

A. BogoSort

B. BubbleSort

C. SelectionSort

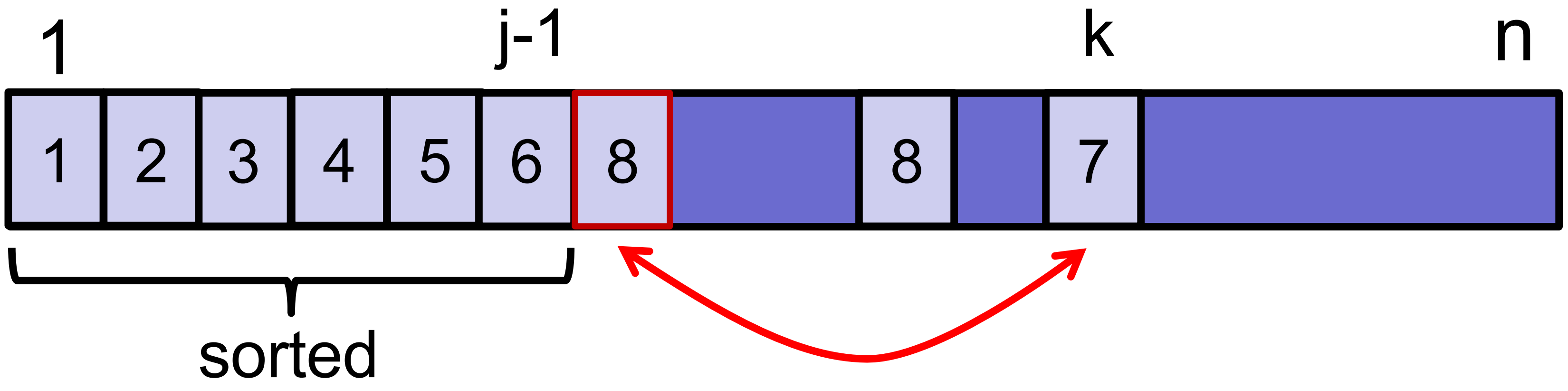
D. InsertionSort

Stable:
Only swap elements
that are different.

SelectionSort

```
SelectionSort(A, n)
  for j ← 1 to n-1:
    find minimum element A[j] in A[j..n]
    swap(A[j], A[k])
```

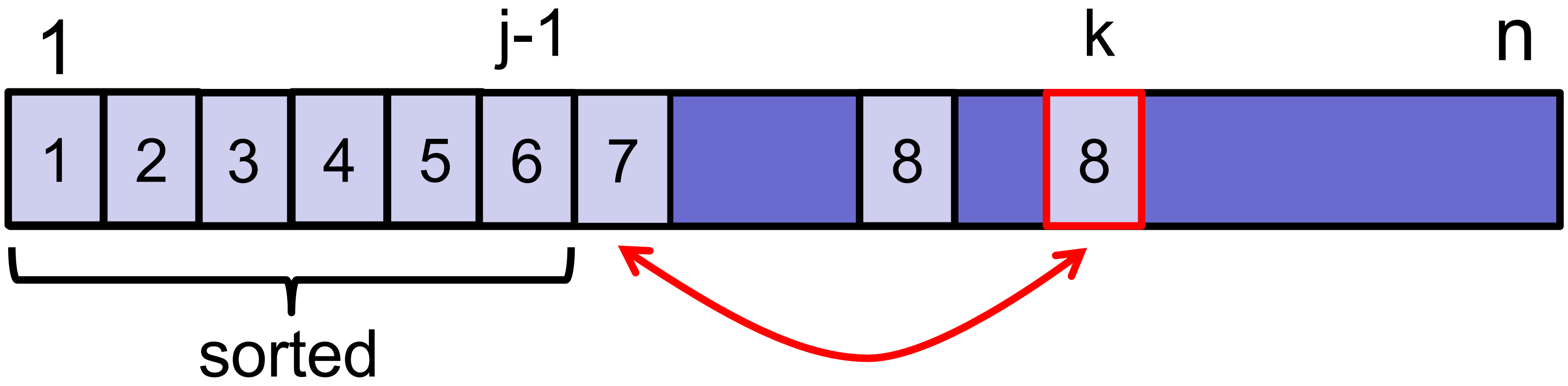
Not stable: swap changes order



SelectionSort

```
SelectionSort(A, n)
    for j ← 1 to n-1:
        find minimum element A[j] in A[j..n]
        swap(A[j], A[k])
```

Not stable: swap changes order



InsertionSort

```
Insertion-Sort(A, n)
    for $j \leftarrow 2$ to $n$
        $key \leftarrow A[j]$
        $i \leftarrow j-1$
        while($i > 0$) and($A[i] > key$)
            $A[i+1] \leftarrow A[i]$
            $i \leftarrow i-1$
            $A[i+1] \leftarrow key$
```

Stable as long as we are careful to implement it properly!

Sorting Analysis

Summary:

BubbleSort: $O(n^2)$

SelectionSort: $O(n^2)$

InsertionSort: $O(n^2)$

Properties: time, space, stability

MergeSort

Divide-and-Conquer

1. Divide problem into smaller sub-problems.
2. Recursively solve sub-problems.
3. Combine solutions.

MergeSort

Divide-and-Conquer Sorting

1. Divide: split array into two halves.
2. Recurse: sort the two halves.
3. Combine: merge the two sorted halves.

MergeSort

Divide-and-Conquer Sorting

1. Divide: split array into two halves.
2. Recurse: sort the two halves.
3. Combine: merge the two sorted halves.

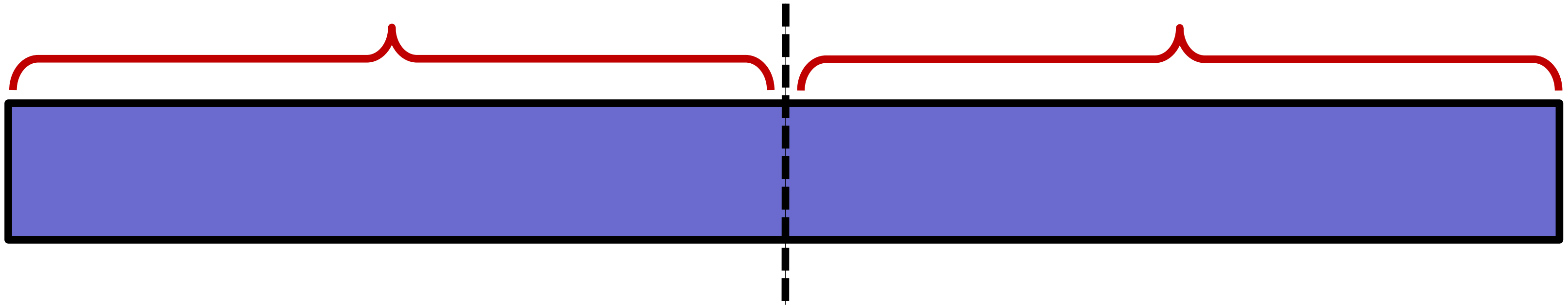
Advice:

When thinking about recursion, do not "unroll" the recursion.
Treat the recursive call as a magic black box.
(But don't forget the base case.)

MergeSort

Step 1:
Divide array into two pieces.

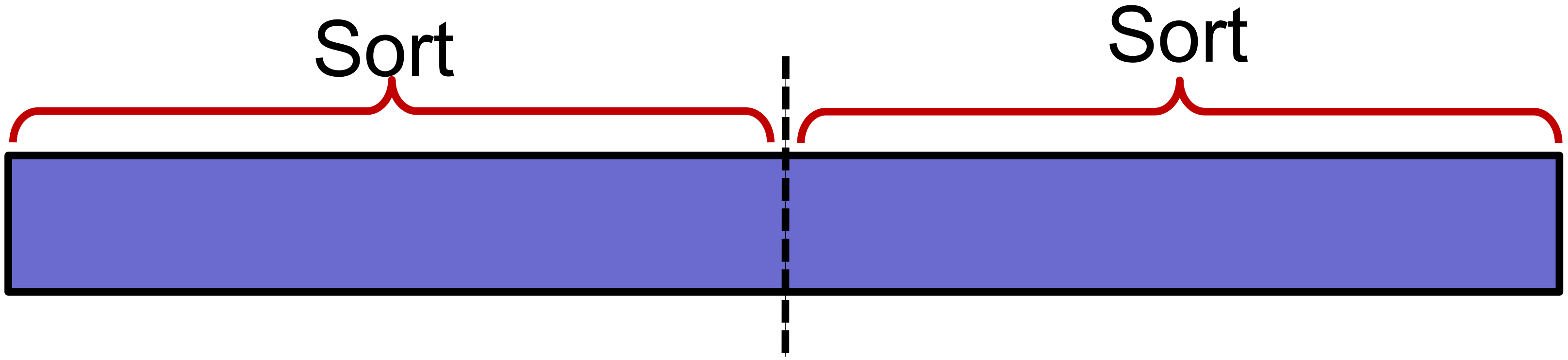
```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ←MergeSort(A[1..n/2],  n/2);
        Y ←MergeSort(A[n/2+1, n],  n/2);
    return Merge (X,Y, n/2);
```



MergeSort

Step 2:
Recursively sort the two halves.

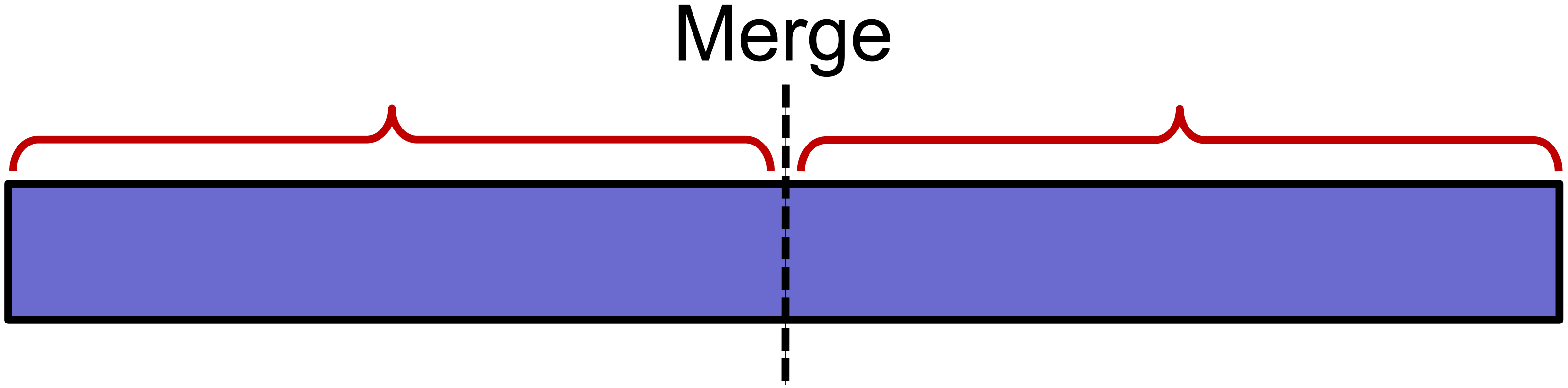
```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ← MergeSort(A[1..n/2],  n/2);
        Y ← MergeSort(A[n/2+1, n],  n/2);
    return Merge (X,Y, n/2);
```



MergeSort

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

```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ←MergeSort(A[1..n/2],  n/2);
        Y ←MergeSort(A[n/2+1, n],  n/2);
    return Merge (X,Y, n/2);
```



MergeSort

```
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);
```

Base case

Recursive “conquer” step

Combine solutions

The only “interesting” part is merging!

MergeSort

Divide-and-Conquer Sorting

1. Divide: split array into two halves.

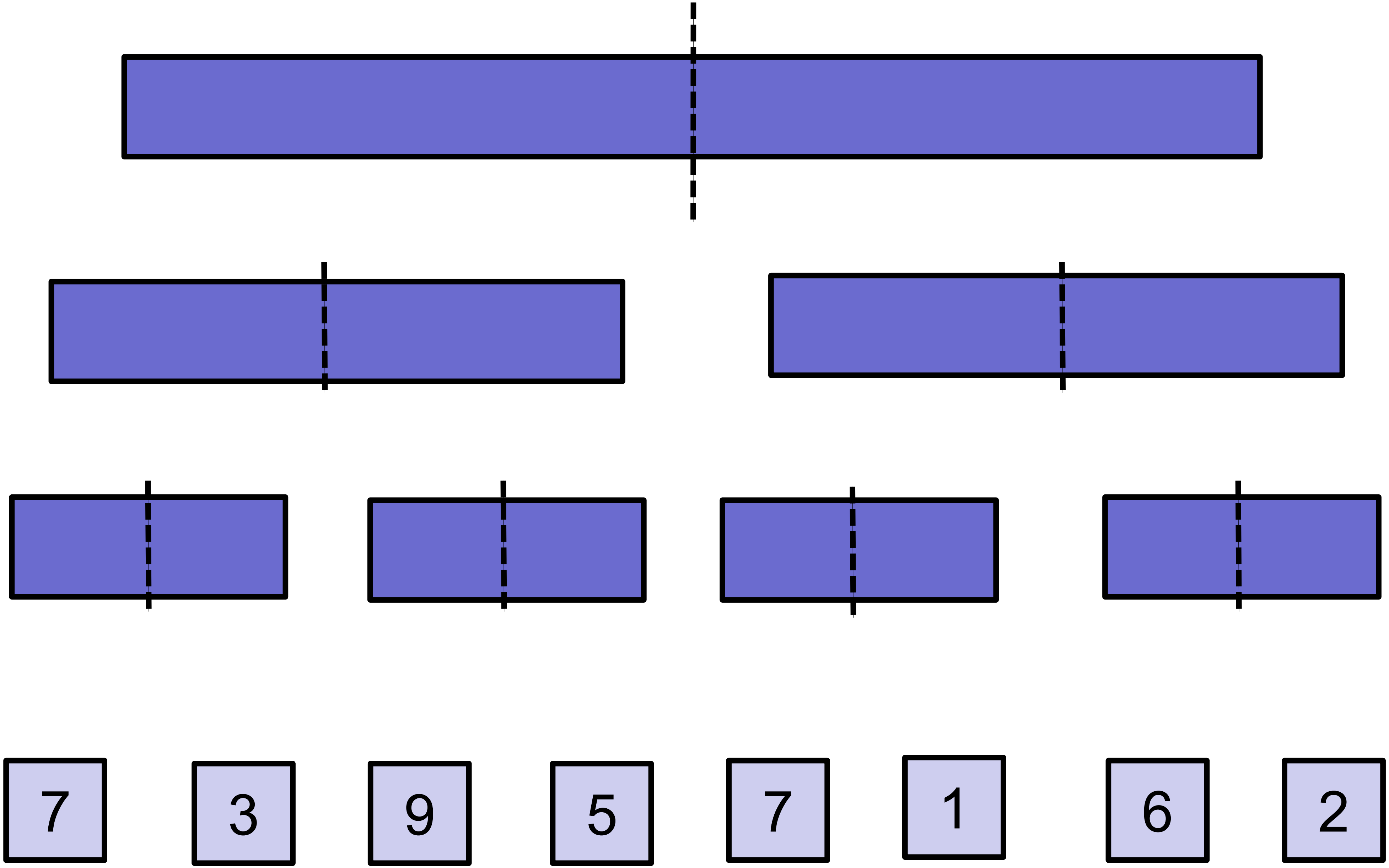
2. Recurse: sort the two halves.

3. Combine: merge the two sorted halves.

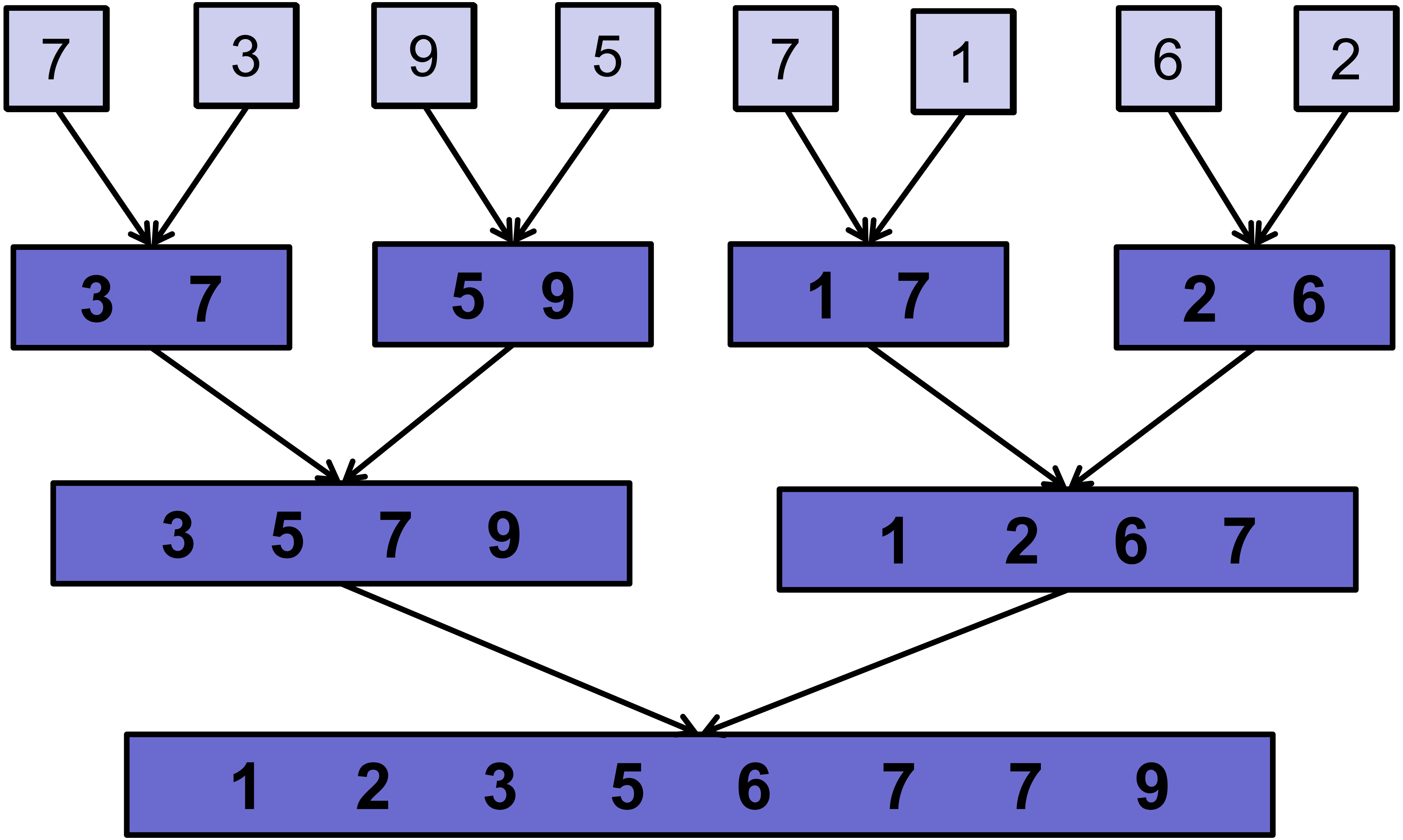
Advice:

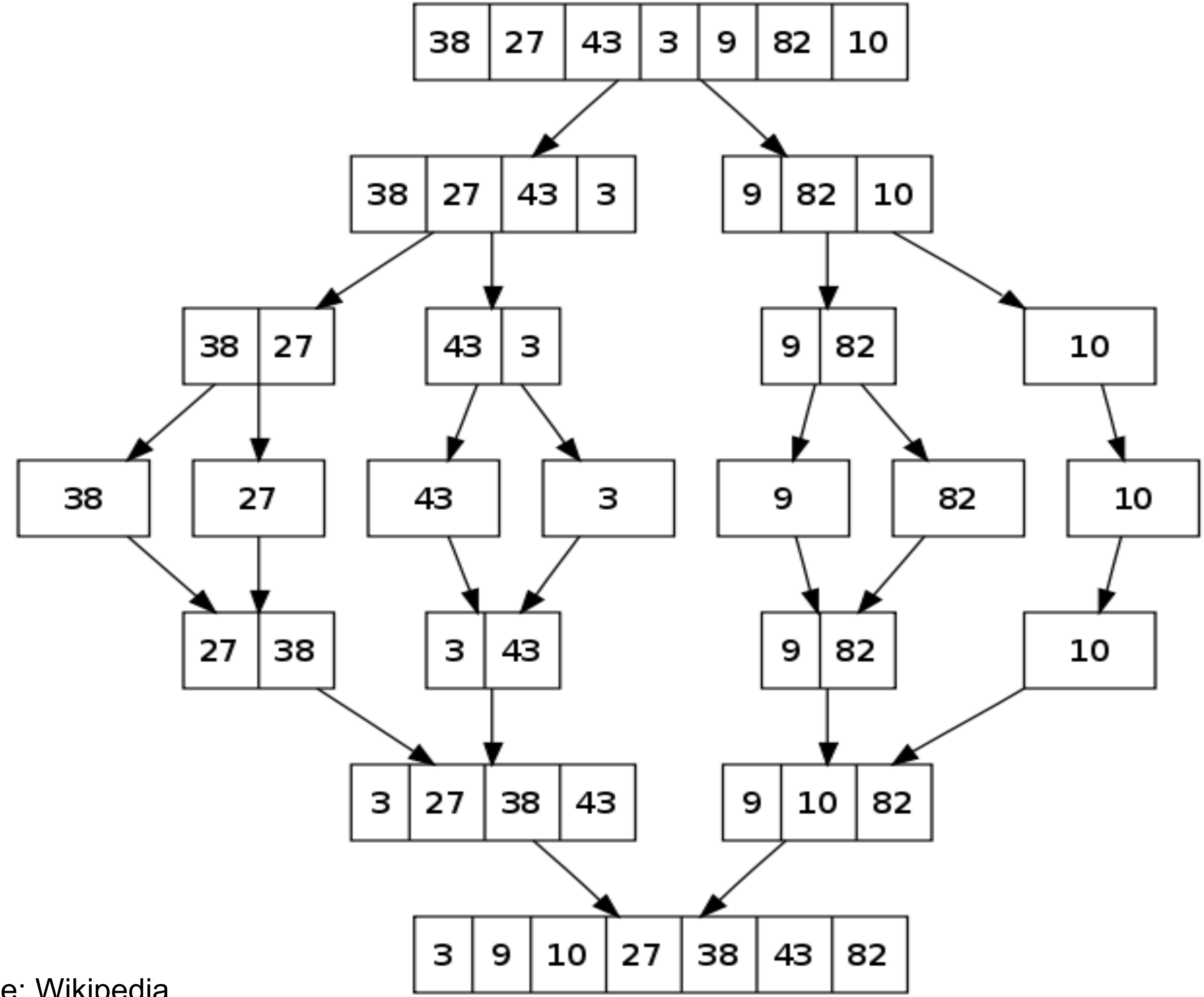
When thinking about recursion, do not "unroll" the recursion.
Treat the recursive call as a magic black box.
(But don't forget the base case.)

Divide-and-Conquer



Merging





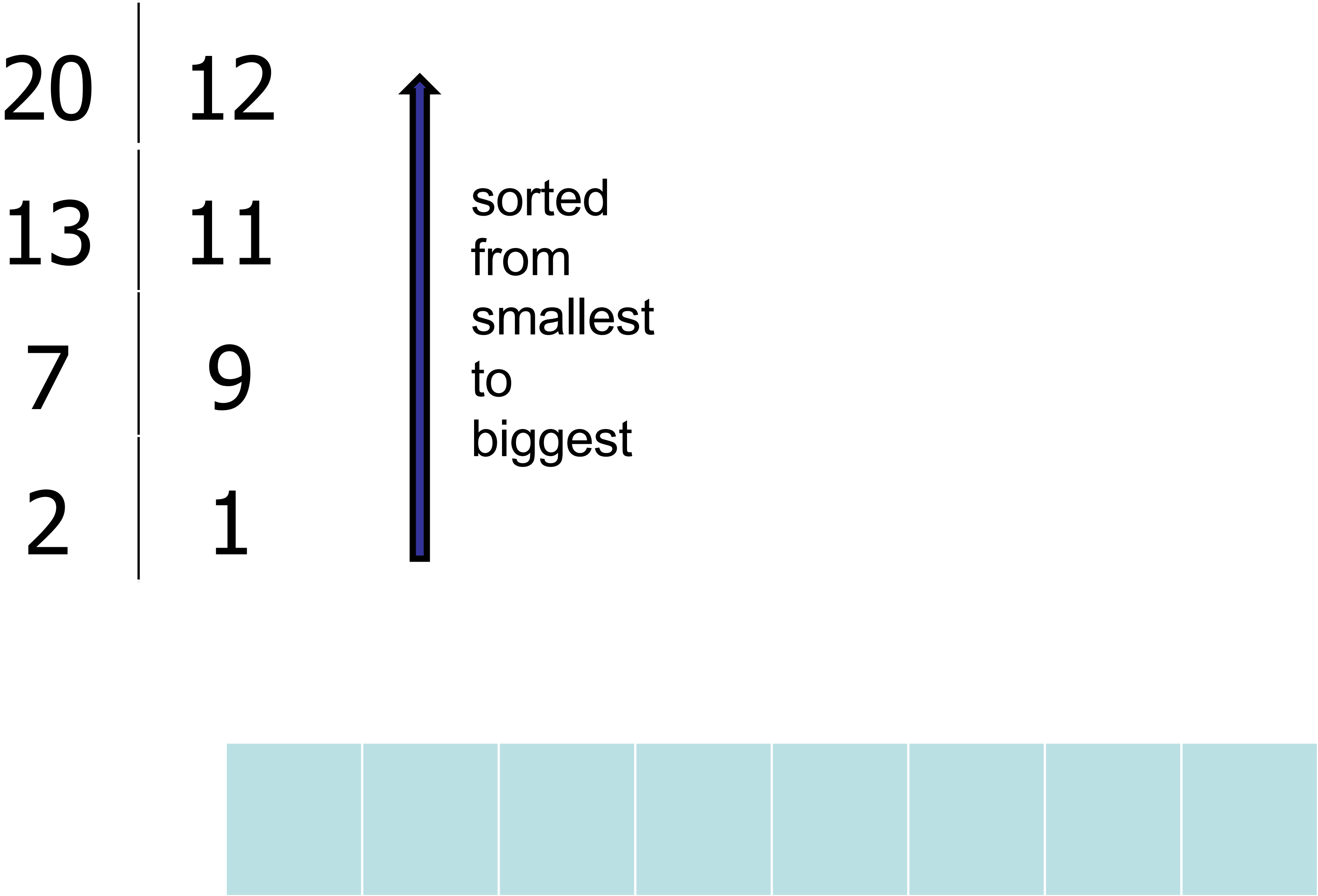
Source: Wikipedia

Merging Two Sorted Lists

Key subroutine: Merge

- How to merge?
- How fast can we merge?

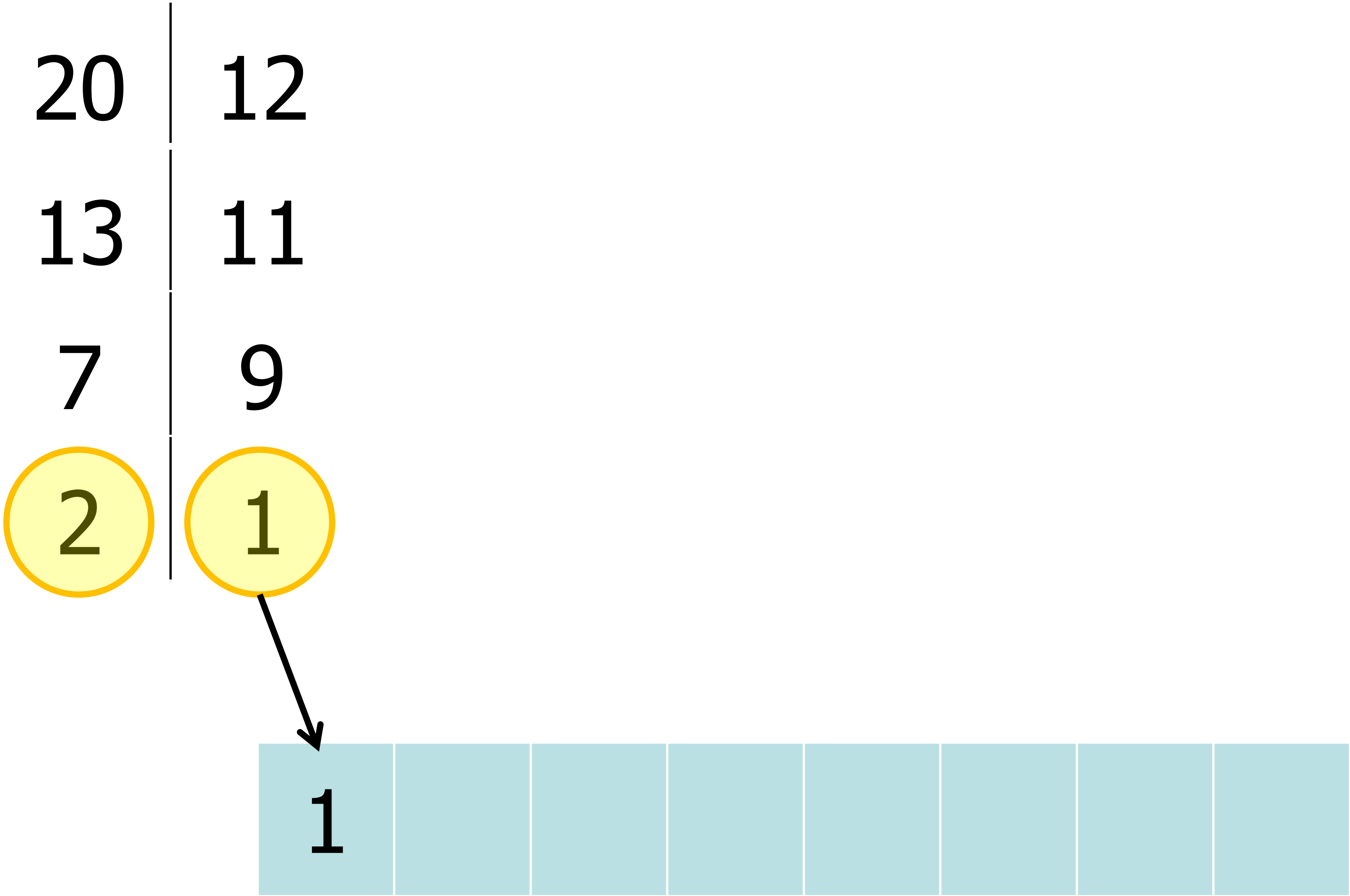
Merging Two Sorted Lists



20	12
13	11
7	9
2	1

sorted
from
smallest
to
biggest

Merging Two Sorted Lists



20 12

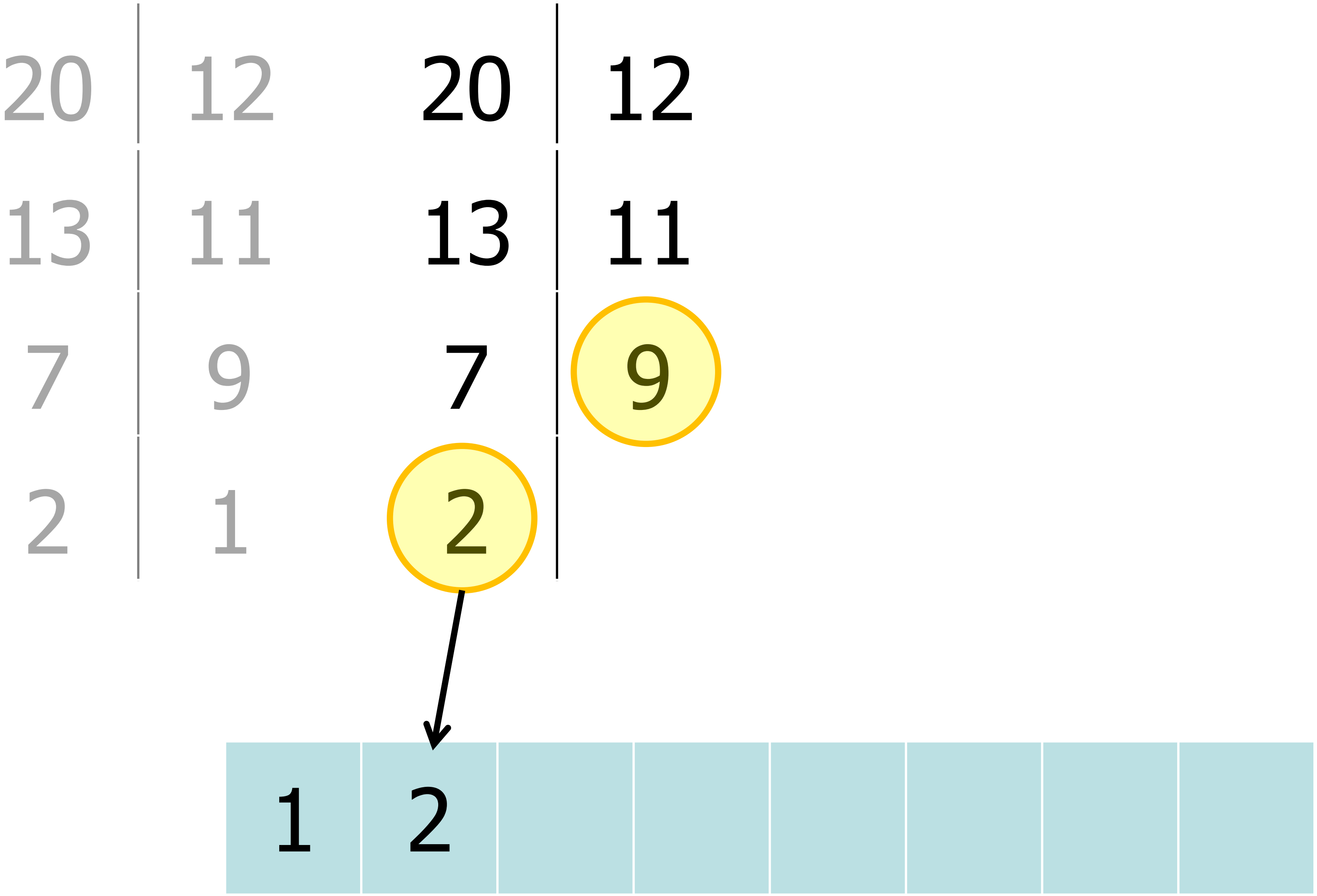
13 11

7 9

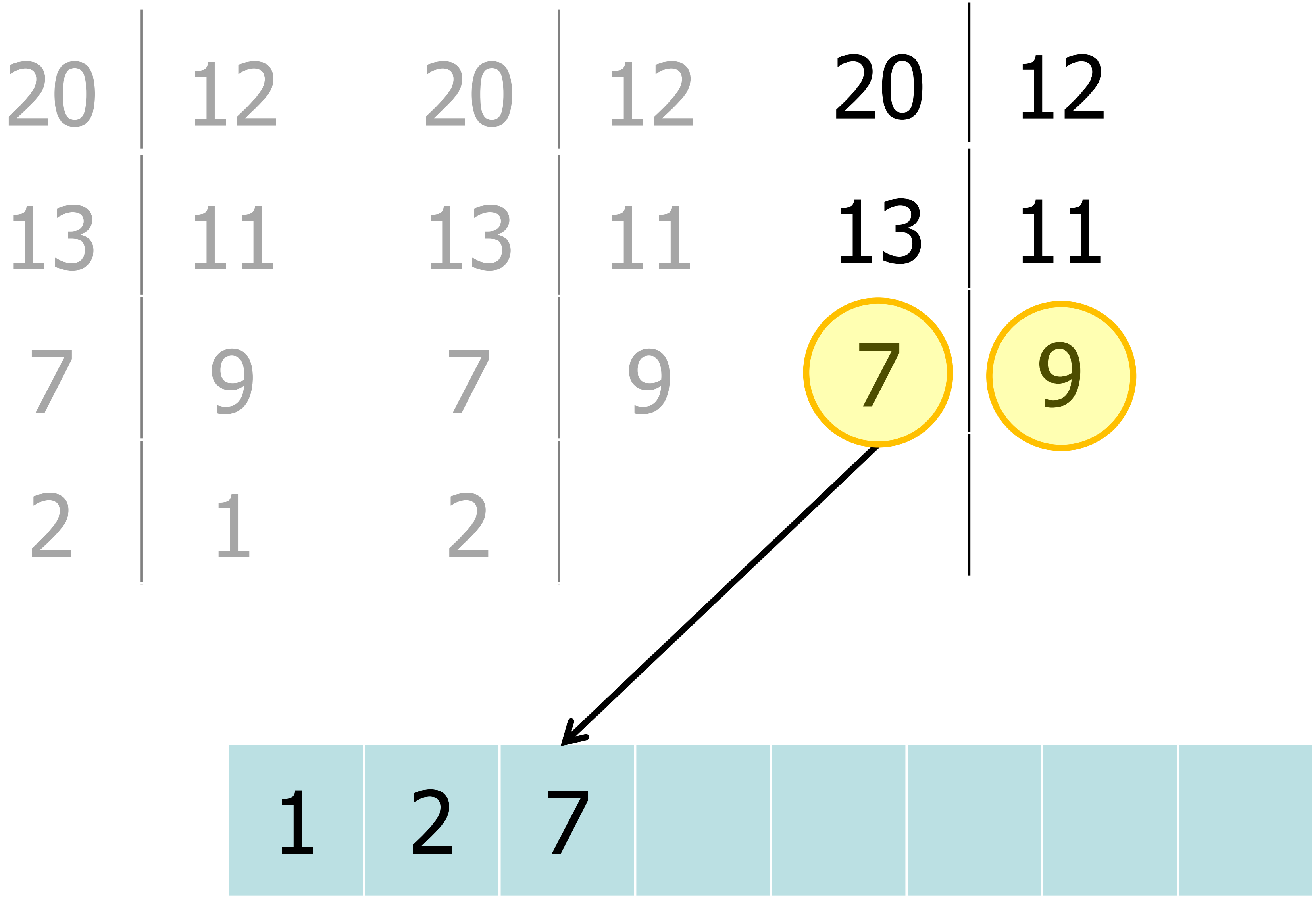
2 1

1

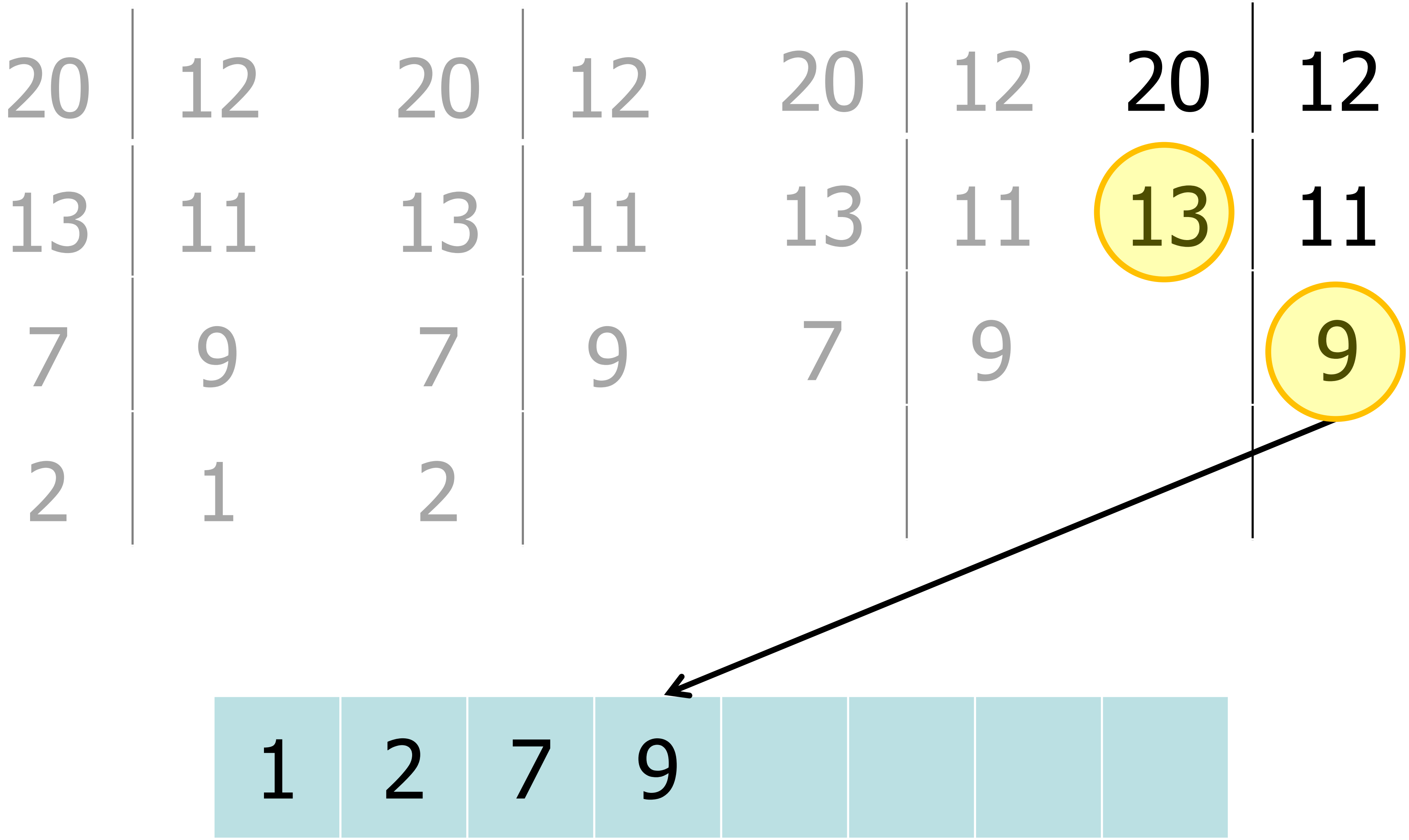
Merging Two Sorted Lists



Merging Two Sorted Lists



Merging Two Sorted Lists



1	2	7	9				

Merging Two Sorted Lists

20	12
13	11
7	9
2	1

20	12
13	11
7	9
2	

20	12
13	11
7	9

20	12
13	11

1	2	7	9	11	12	13	20

Merge: Running Time

Given two lists:

- A of size n/2
- B of size n/2

Total running time: ??

ARCHIPELAGO

is open

Merge: Running Time

Given two lists:

- A of size n/2
- B of size n/2

Total running time: O(n) = cn

- In each iteration, move *one* element to final list.
- After n iterations, all the items are in the final list.
- Each iteration takes O(1) time to compare two elements and copy one.

Merge-Sort Analysis

Let $T(n)$ be the worst-case running time for an array of n elements.

```
MergeSort(A, n)
    if $(n=1)$ then return;  <--------------------- $\theta(1)$
    else:
        X ← Merge-Sort(…);    <------------- $T(n/2)$
        Y ← Merge-Sort(…);    <--------------- $T(n/2)$
    return Merge (X,Y, n/2);  <-------------- $\theta(n)$
```

MergeSort Analysis

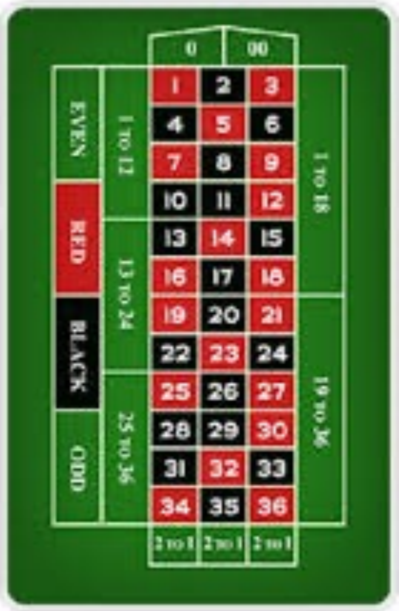
Let $T(n)$ be the worst-case running time for an array of n elements.

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

PotW: Gambling for Profit?

Alice

- Begins with $100
- Bets $1 each time.
- Each bet has a 51% chance of winning.
- On win: +1
  On lose: -1



Bob

- Begins with $100
- Bets $1 each time.
- Each bet has a 49% chance of winning.
- On win: +1
  On lose: -1

PotW: Gambling for Profit?

Alice

- Begins with $100
- Bets $1 each time.
- Each bet has a 51% chance of winning.
- On win: +1
  On lose: -1

Bob

- Begins with $100
- Bets $1 each time.
- Each bet has a 49% chance of winning.
- On win: +1
  On lose: -1

PotW: Gambling for Profit?

Alas, both Alice and Bob lose all their money (gambling at two different tables). Who is more likely to go bankrupt first (conditioned on both losing)?

Alice

- Begins with $100
- Bets $1 each time.
- Each bet has a 51% chance of winning.
- On win: +1
  On lose: -1

Bob

- Begins with $100
- Bets $1 each time.
- Each bet has a 49% chance of winning.
- On win: +1
  On lose: -1

PotW: Gambling for Profit?

Alas, both Alice and Bob lose all their money (gambling at two different tables). Who is more likely to go bankrupt first?

Alice

- Begins with \$100
- Bets \$1 each time.
- Each bet has a 51% chance of winning.
- On win: +1
  On lose: -1

Hints:

- Bayes Rule!
- Alice eventually goes bankrupt w.p. $(0.49/0.51)^{100}$.
- For every sequence where Alice loses, you can construct an inverted sequence where Bob loses.

Bob

- Begins with \$100
- Bets \$1 each time.
- Each bet has a 49% chance of winning.
- On win: +1
  On lose: -1

MergeSort Analysis

Let $T(n)$ be the worst-case running time for an array of n elements.

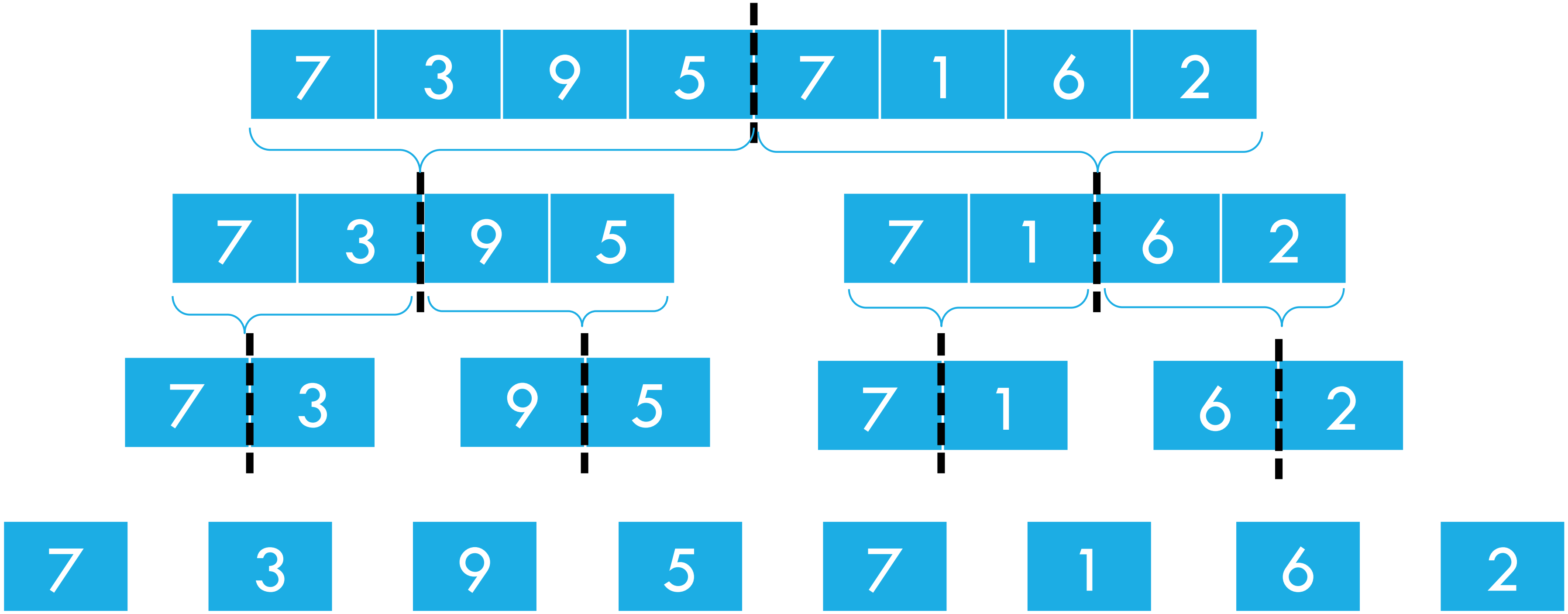
$$T(n) = \theta(1) \quad \textbf{if } (n=1)$$

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

Techniques for Solving Recurrences

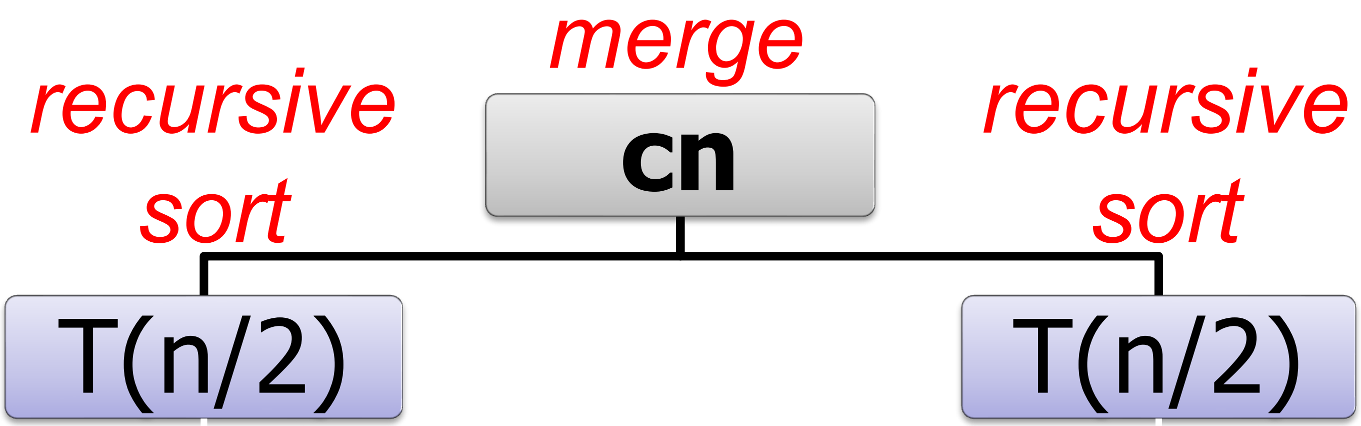
1. Guess and verify (via induction).
2. Draw the recursion tree.
3. Use the Master Theorem (see CS3230) or the Akra–Bazzi Method, or other advanced techniques.

MergeSort: Recurse “downwards”



MergeSortAnalysis

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



merge

cn

recursive sort

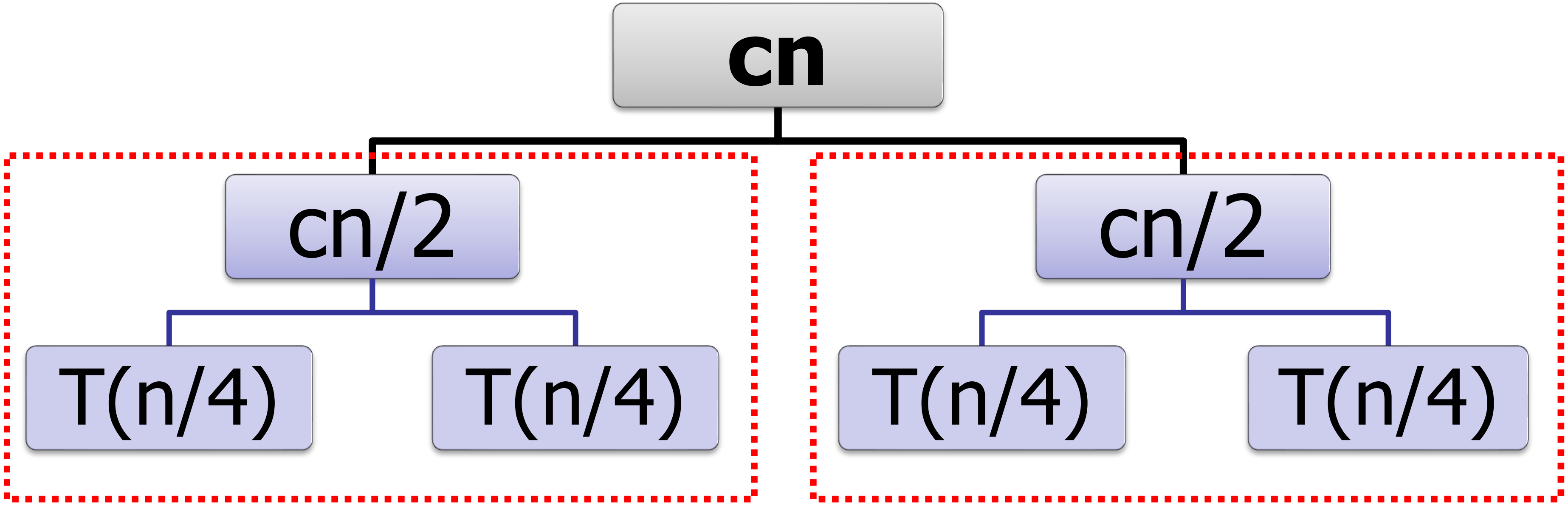
recursive sort

$T(n/2)$

$T(n/2)$

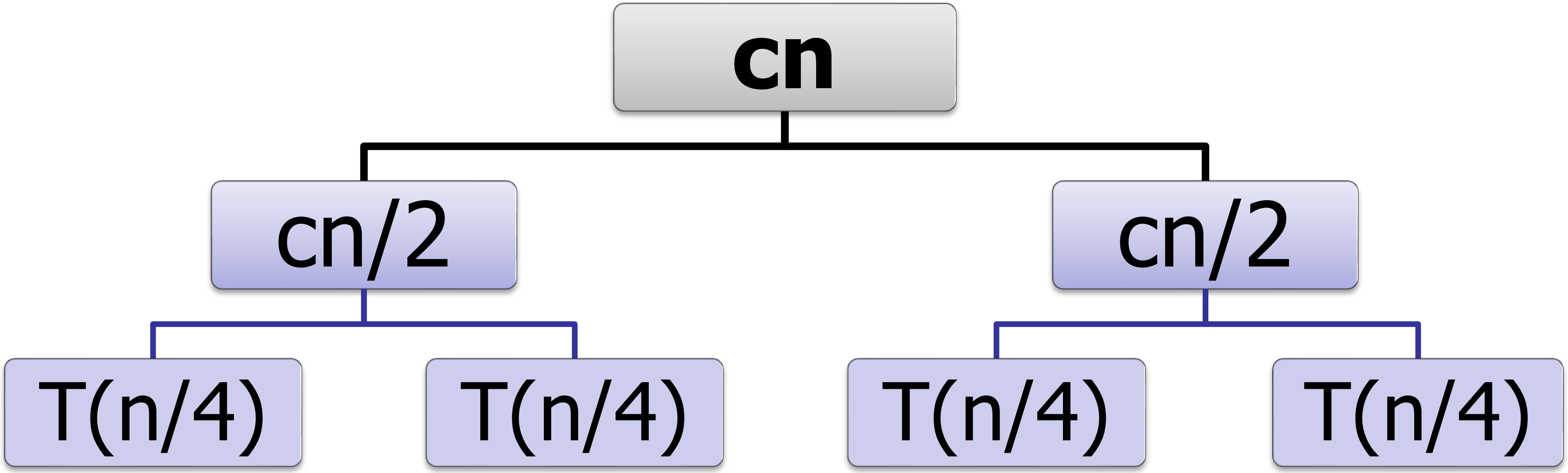
MergeSortAnalysis

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



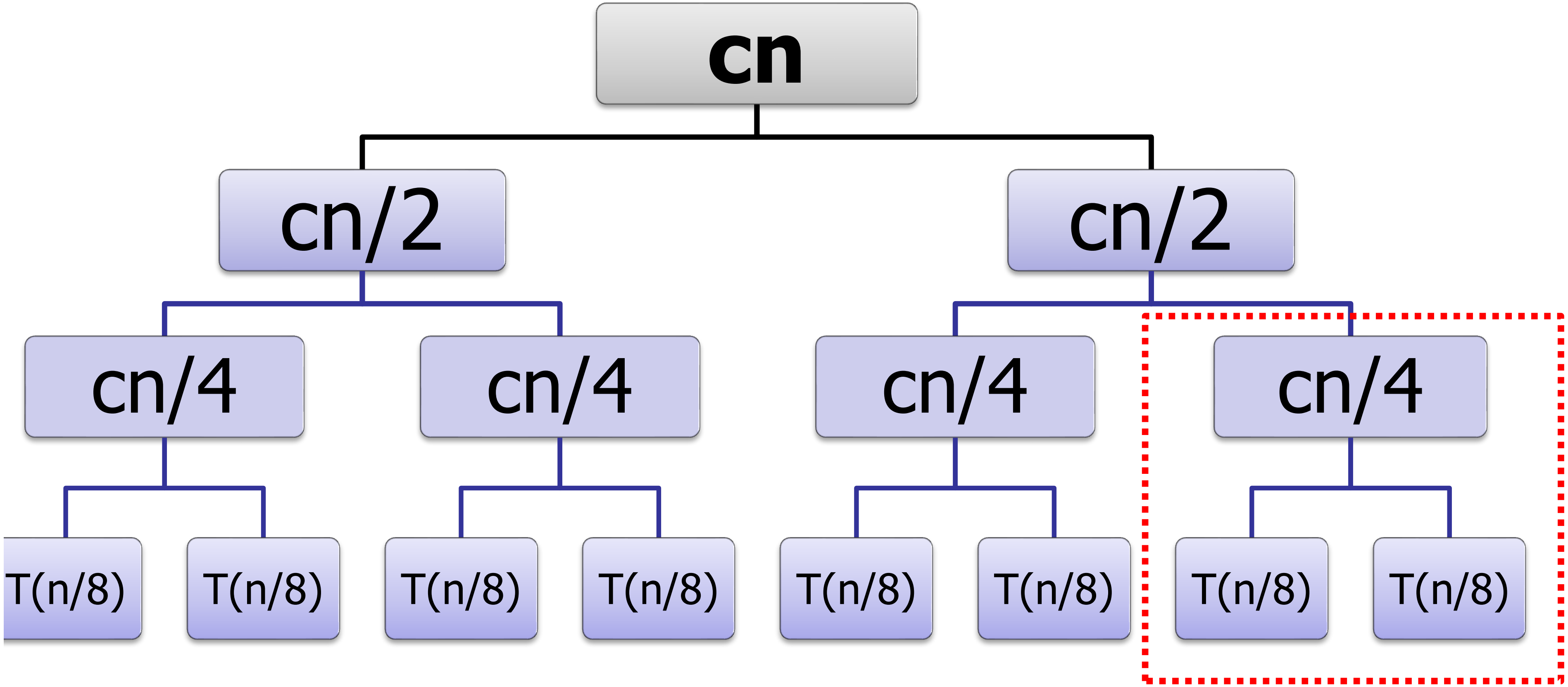
MergeSortAnalysis

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



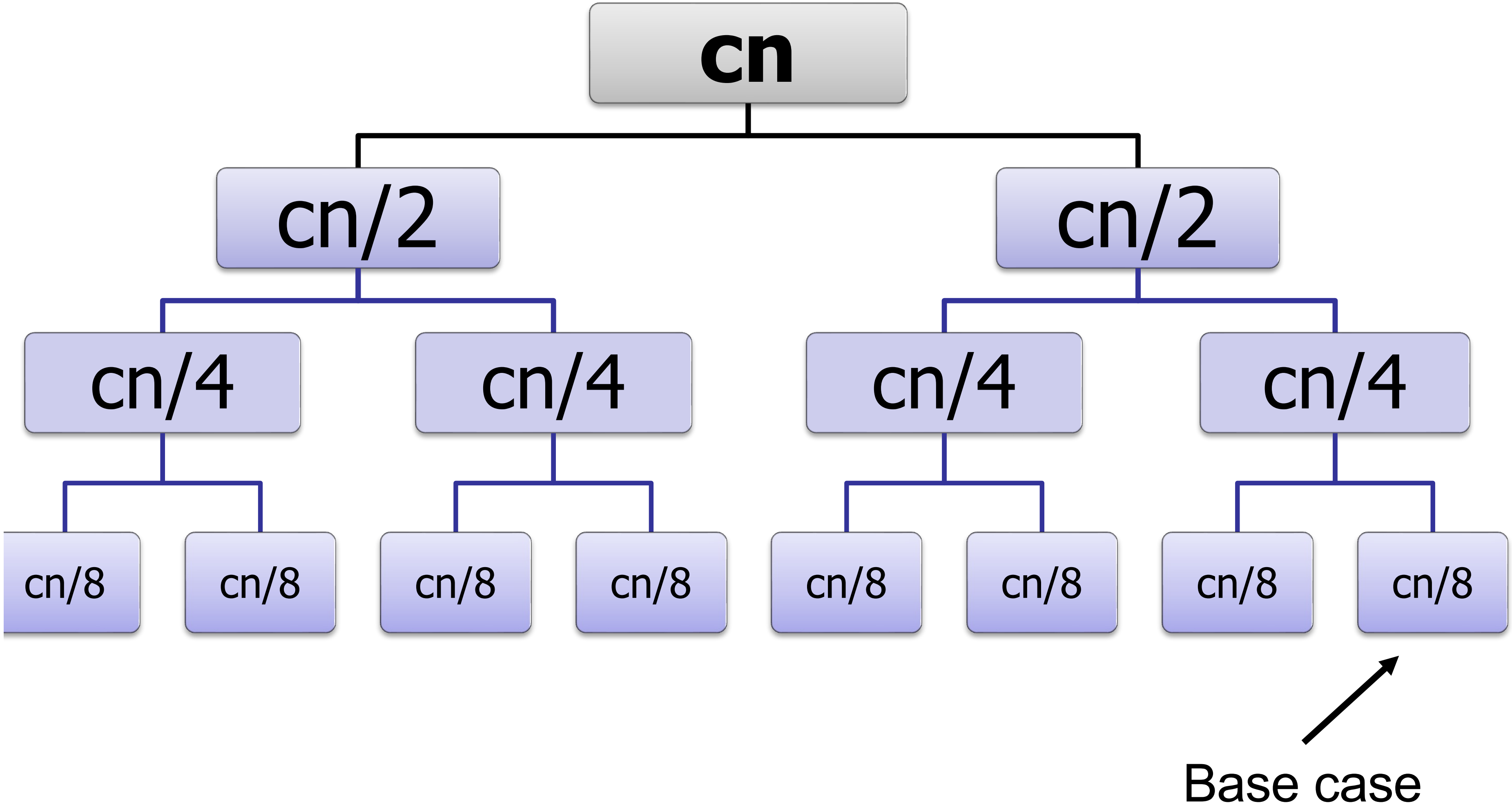
MergeSort Analysis

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



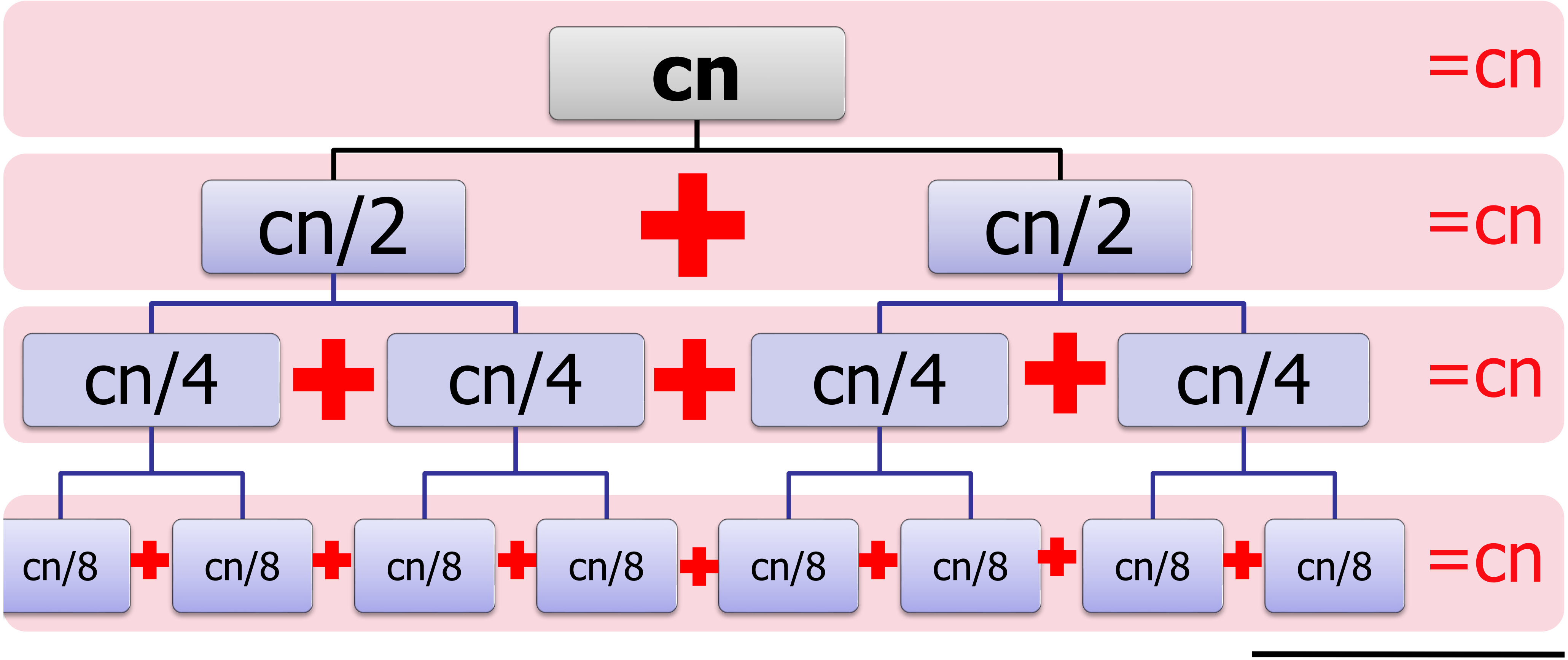
MergeSort Analysis

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



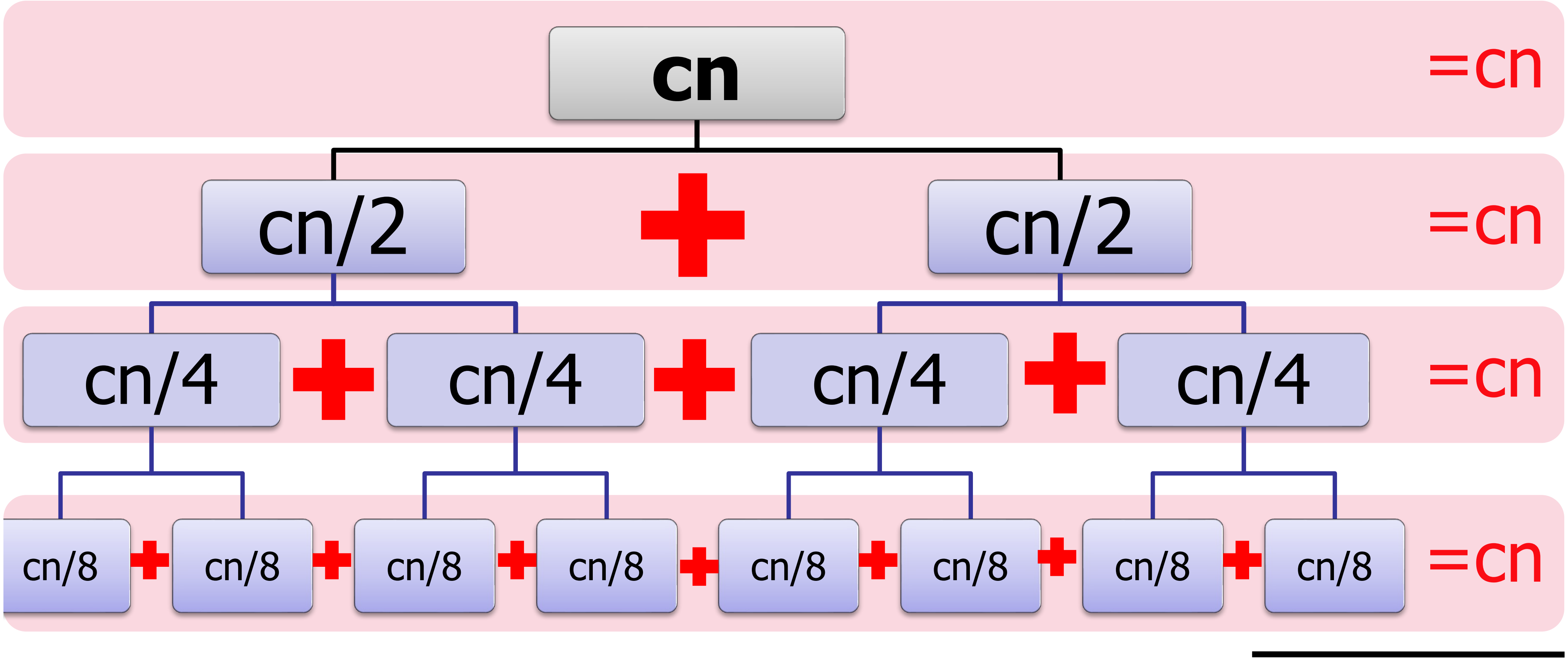
MergeSort Analysis

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



MergeSort Analysis

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



Key question: how many levels?

MergeSort Analysis

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

level	number
0	1
1	2
2	4
3	8
4	16
…	…
h	??

$$\text{number} = 2^{\text{level}}$$

MergeSort Analysis

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

level	number
0	1
1	2
2	4
3	8
4	16
…	…
h	*n*

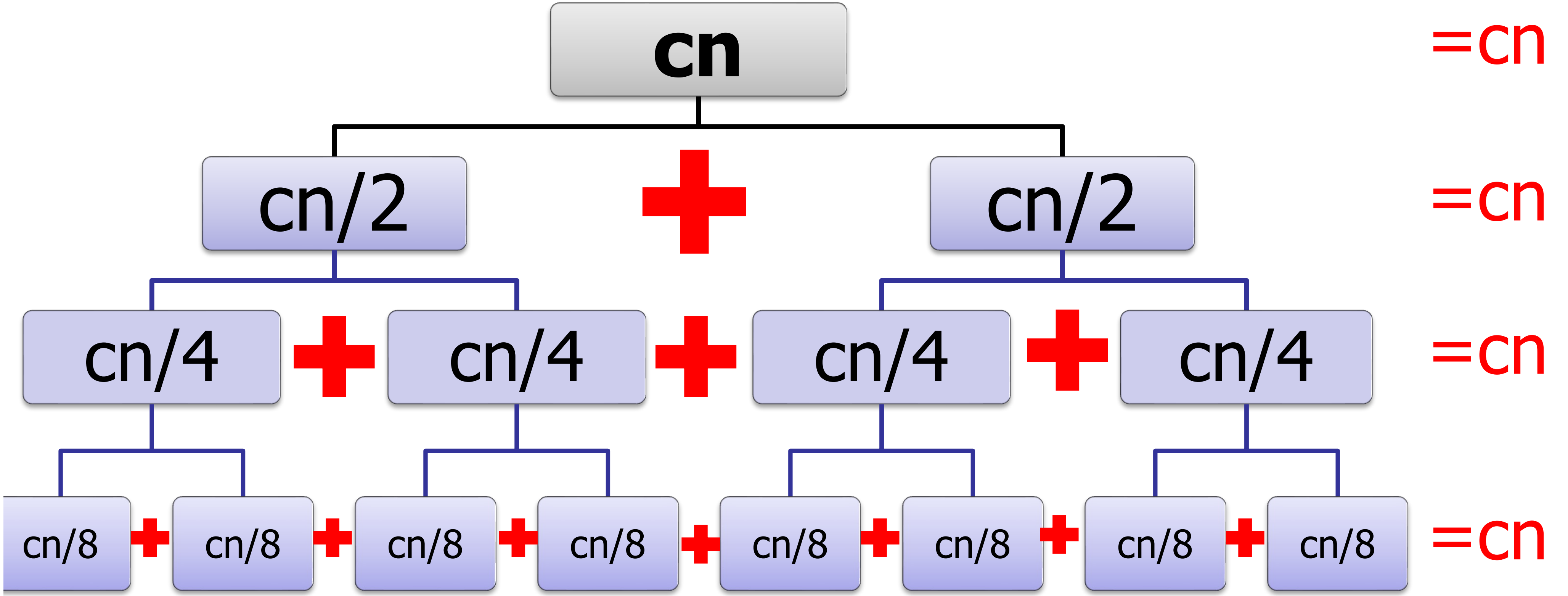
$$\text{number} = 2^{\text{level}}$$

$$n = 2^h$$

$$\log n = h$$

MergeSort Analysis

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



cn log n

MergeSortAnalysis

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

```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ← MergeSort(…);
        Y ← MergeSort(…);
    return Merge (X,Y, n/2);
```

Techniques for Solving Recurrences

1. Guess and verify (via induction).

2. Draw the recursion tree.

3. Use the Master Theorem (see CS3230) or the Akra–Bazzi Method, or other advanced techniques.

Guess: $T(n) = O(n \log n)$

Recurrence being analyzed:

$T(n) = 2T(n/2) + c \cdot n$

$T(1) = c$

Guess: $T(n) = c \cdot n \log n$

More precise guess:
Fix constant c.

Recurrence being analyzed:

$T(n) = 2T(n/2) + c \cdot n$

$T(1) = c$

Guess: $T(n) = c \cdot n \log n$

Induction:
Base case

$T(1) = c$

Recurrence being analyzed:

$T(n) = 2T(n/2) + c \cdot n$

$T(1) = c$

Guess: $T(n) = c \cdot n \log n$

Induction:
Assume true for all smaller values.

$T(1) = c$

$T(x) = c \cdot x \log x$ for all $x < n$.

Recurrence being analyzed:

$T(n) = 2T(n/2) + c \cdot n$

$T(1) = c$

Guess: T(n) = c·n log n

Induction:
Prove for n.

T(1) = c

T(x) = c·x log x for all x < n.

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

Recurrence being analyzed:

T(n) = 2T(n/2) + c·n

T(1) = c

Guess: T(n) = c·n log n

T(1) = c

T(x) = c·x log x for all x < n.

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

Induction:
It works!

Recurrence being analyzed:

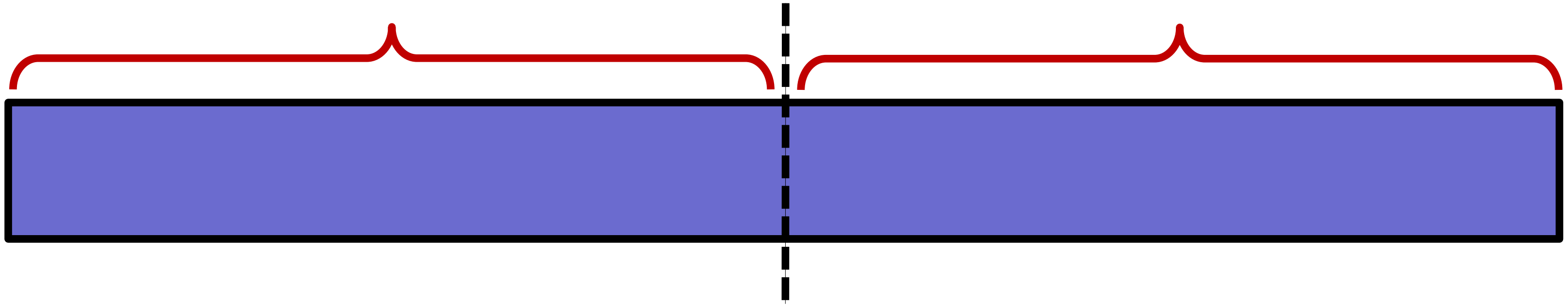
T(n) = 2T(n/2) + c·n

T(1) = c

Top-Down vs. ...

Step 1:
Divide array into two pieces.

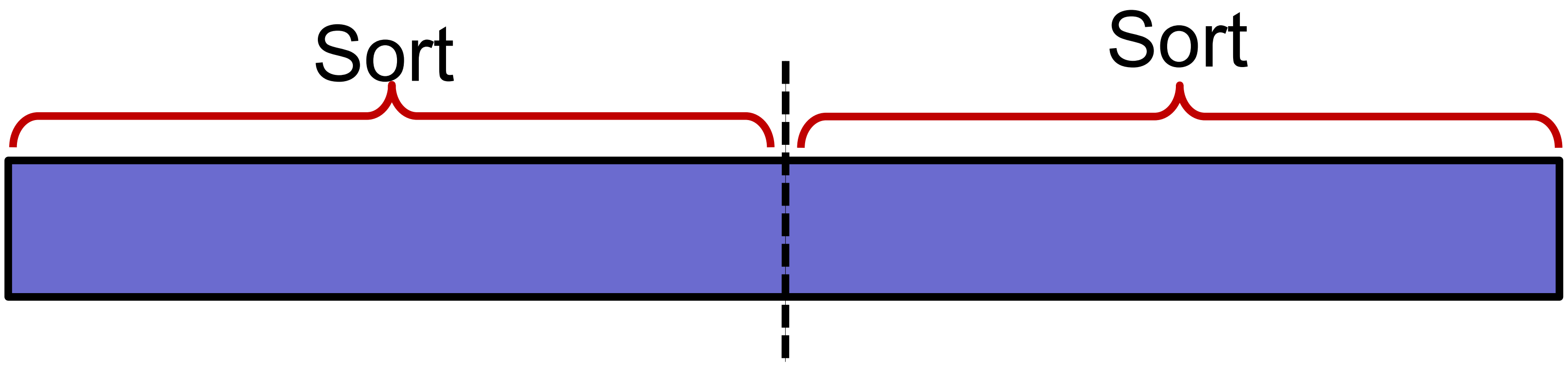
```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ←MergeSort(A[1..n/2], n/2);
        Y ←MergeSort(A[n/2+1, n], n/2);
    return Merge (X,Y, n/2);
```



Top-Down vs. …

Step 2:
Recursively sort the two halves.

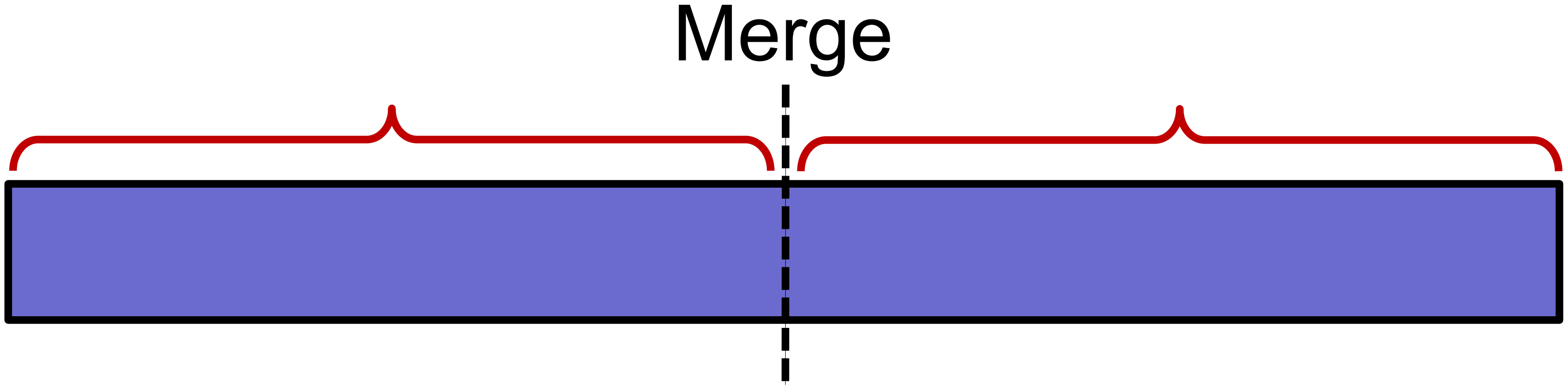
```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ←MergeSort(A[1..n/2],  n/2);
        Y ←MergeSort(A[n/2+1, n],  n/2);
    return Merge (X,Y, n/2);
```

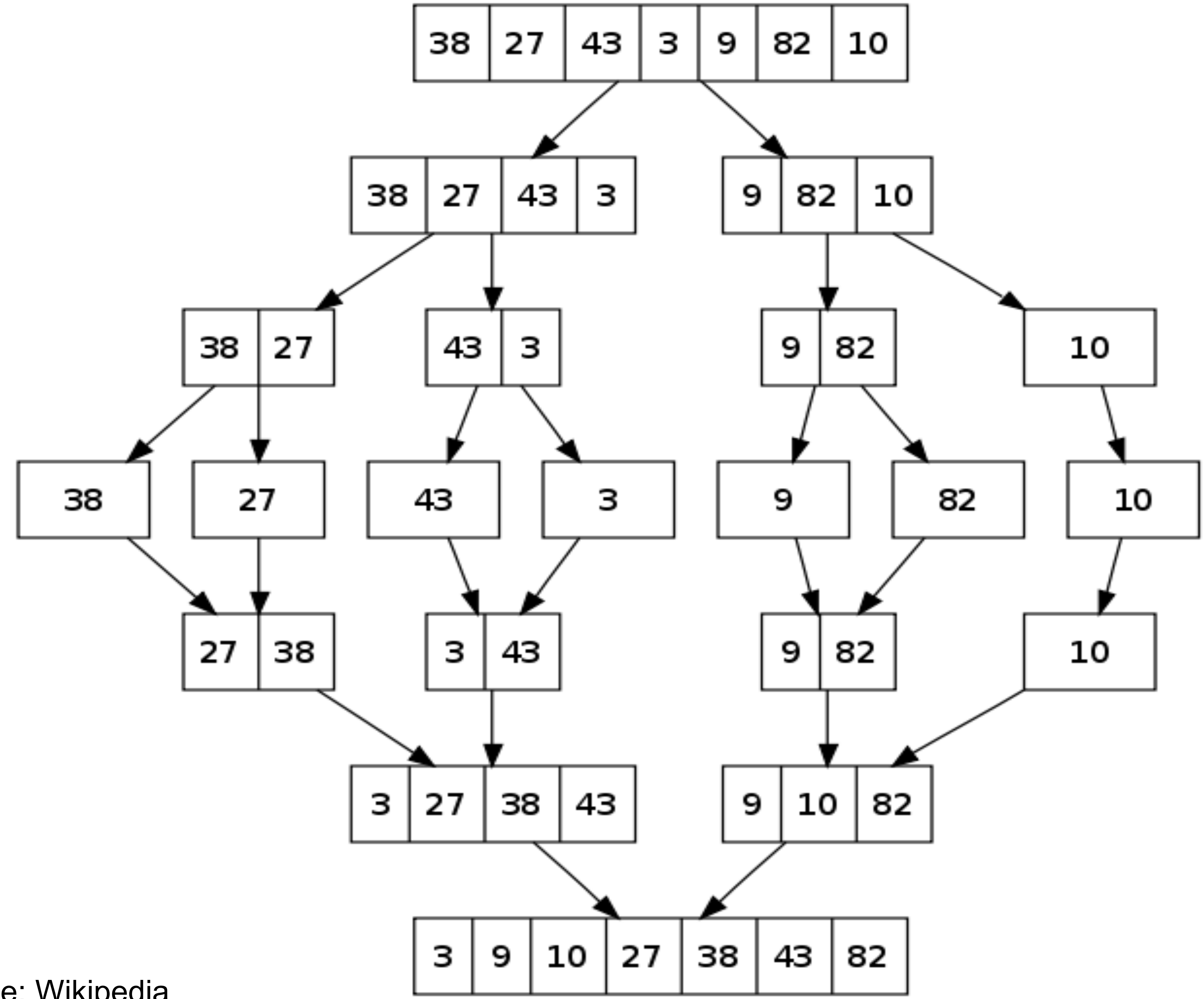


Top-Down vs. ...

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

MergeSort, Bottom Up

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

How much does it matter?

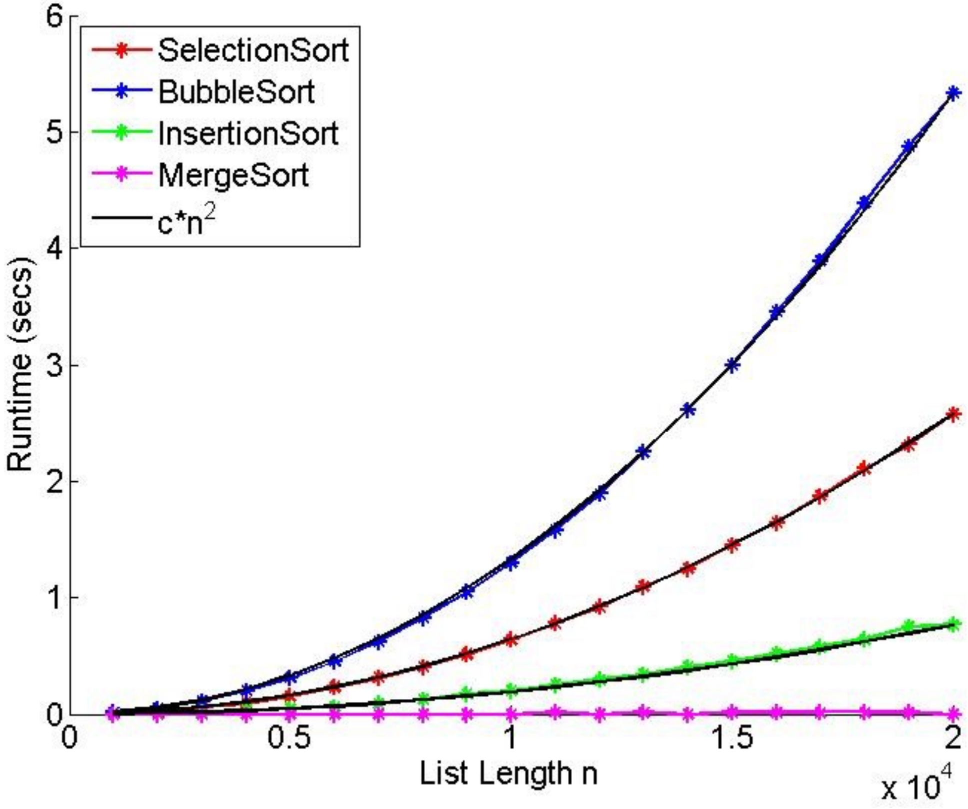
Comparing words in two files:

Version	Change	Running Time
Version 1		4,311.00s
Version 2	Better file handling	676.50s
Version 3	Mergesort replaces SelectionSort	6.59s
Version 4	Hashing replaces sorting	2.35s

Algorithm:

1. Read all text in both files.
2. Sort words.
3. Count how many times each word appears in each file.

real world performance



http://www.cs.toronto.edu/~jepson/csc148/2007F/notes/sorting.html]

When is it better to use InsertionSort instead of MergeSort?

A. When there is limited space?

B. When there are a lot of items to sort?

C. When there is a large memory cache?

D. When there are a small number of items?

E. When the list is mostly sorted?

F. Always

G. Never

MergeSort

When the list is mostly sorted:

- InsertionSort is fast!
- MergeSort is O(n log n)

How “close to sorted” should a list be for InsertionSort to be faster?

How would you check?

MergeSort

Small number of items to sort:

- MergeSort is slow!
- Caching performance, branch prediction, etc.
- User InsertionSort for n < 1024, say.

Base case of recursion:

- Use slower sort.

Run an experiment
and post on the forum
what the best switch-over
point is for your machine.

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.

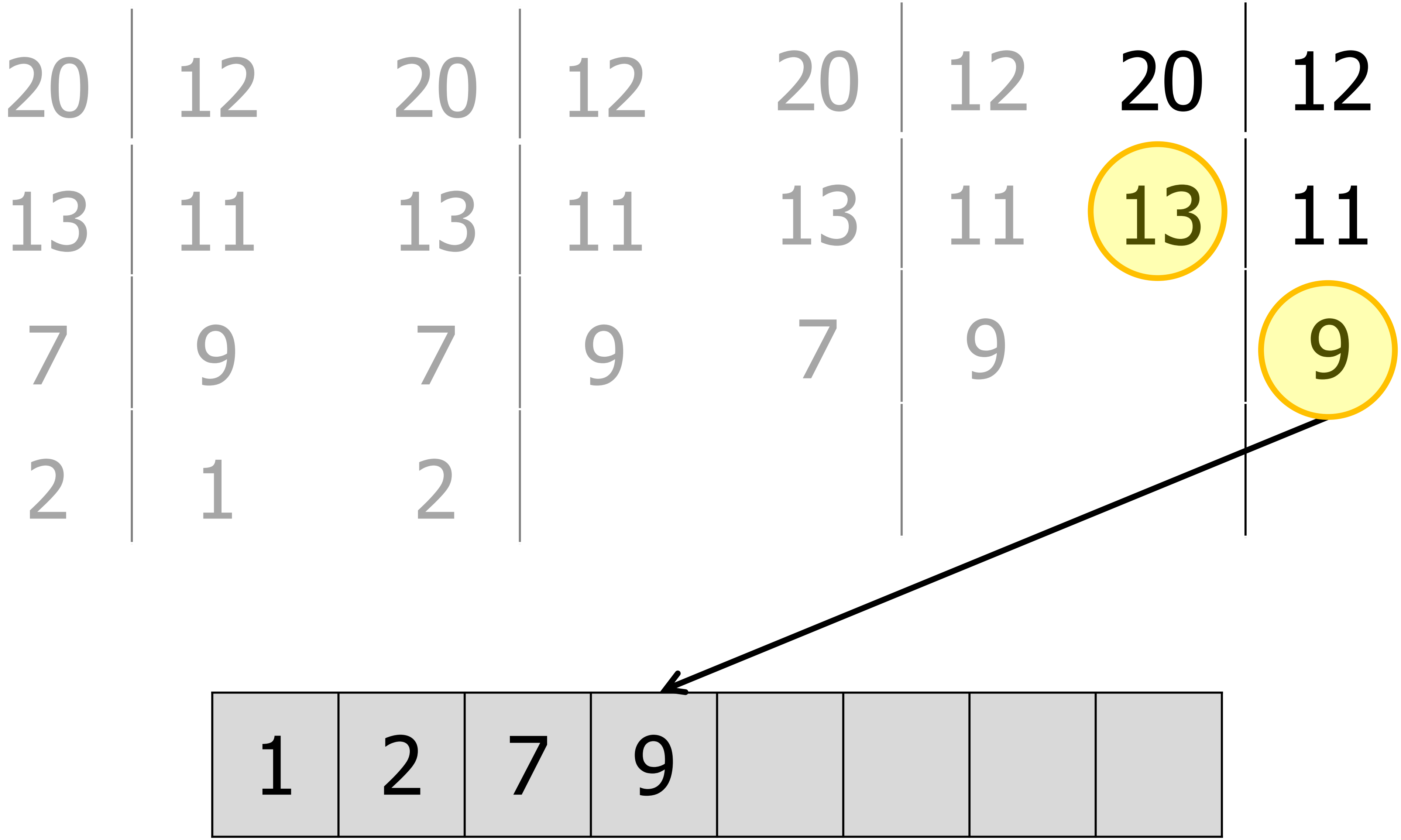
Space Complexity

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;          ←-------------------- θ(1)
    else:
        X ← Merge-Sort(…);         ←------------ S(n/2)
        Y ← 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

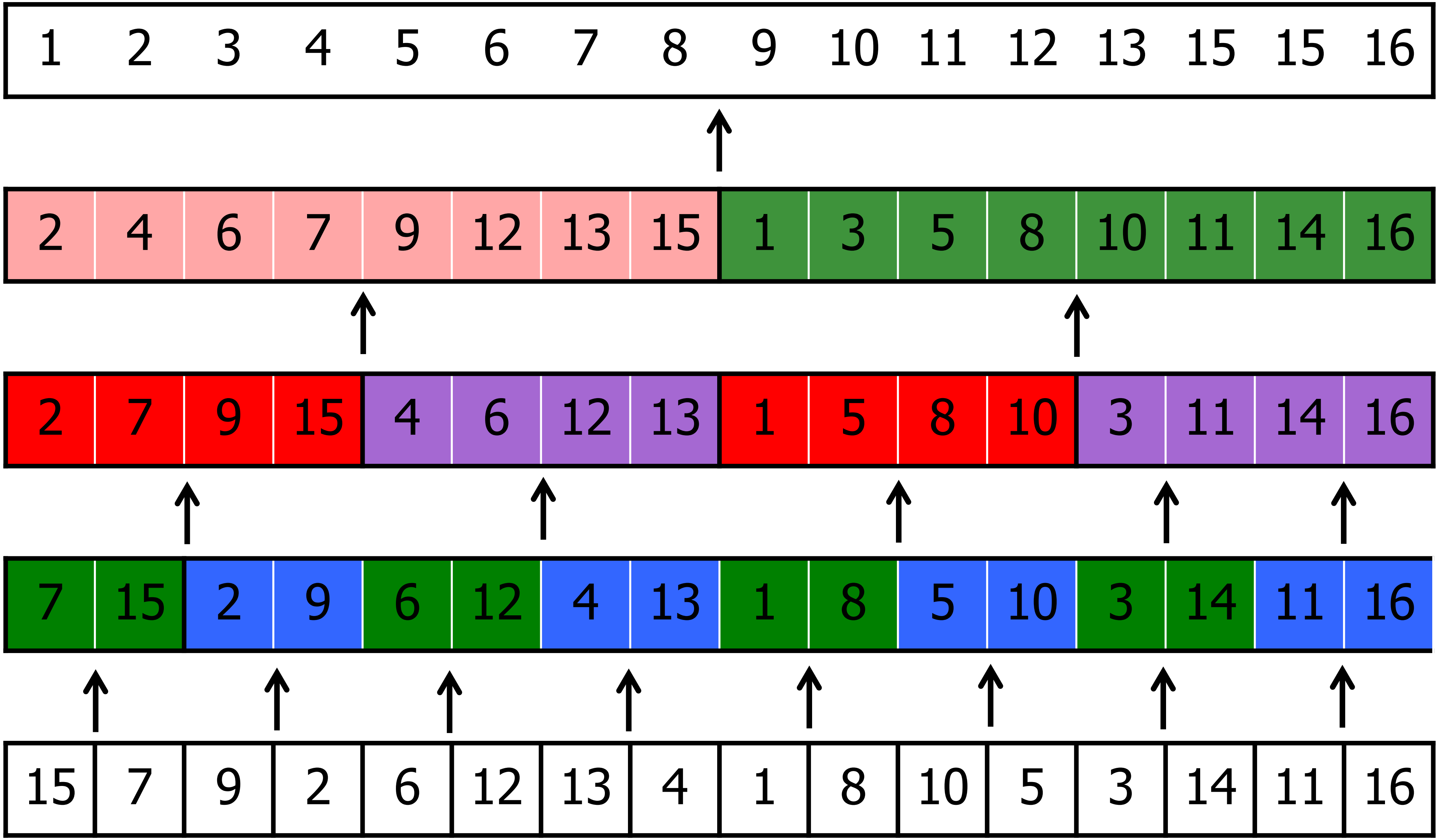
is open

Space Analysis

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

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

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

Challenge of the Day:

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

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

Stability

Is MergeSort stable?

ARCHIPELAGO

is open

MergeSort

Stability:

- MergeSort is stable if “merge” is stable.
- Merge is stable if properly implemented.

Sorting Analysis

Summary:

BubbleSort: $O(n^2)$

SelectionSort: $O(n^2)$

InsertionSort: $O(n^2)$

MergeSort: $O(n \log n)$

Also:

The power of divide-and-conquer!

How to solve recurrences…

Properties: time, space, stability

Sorting, continued

QuickSort

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

(Warning: PS3 opens today and depends on QuickSort, but you can get started without that.)

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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes

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)

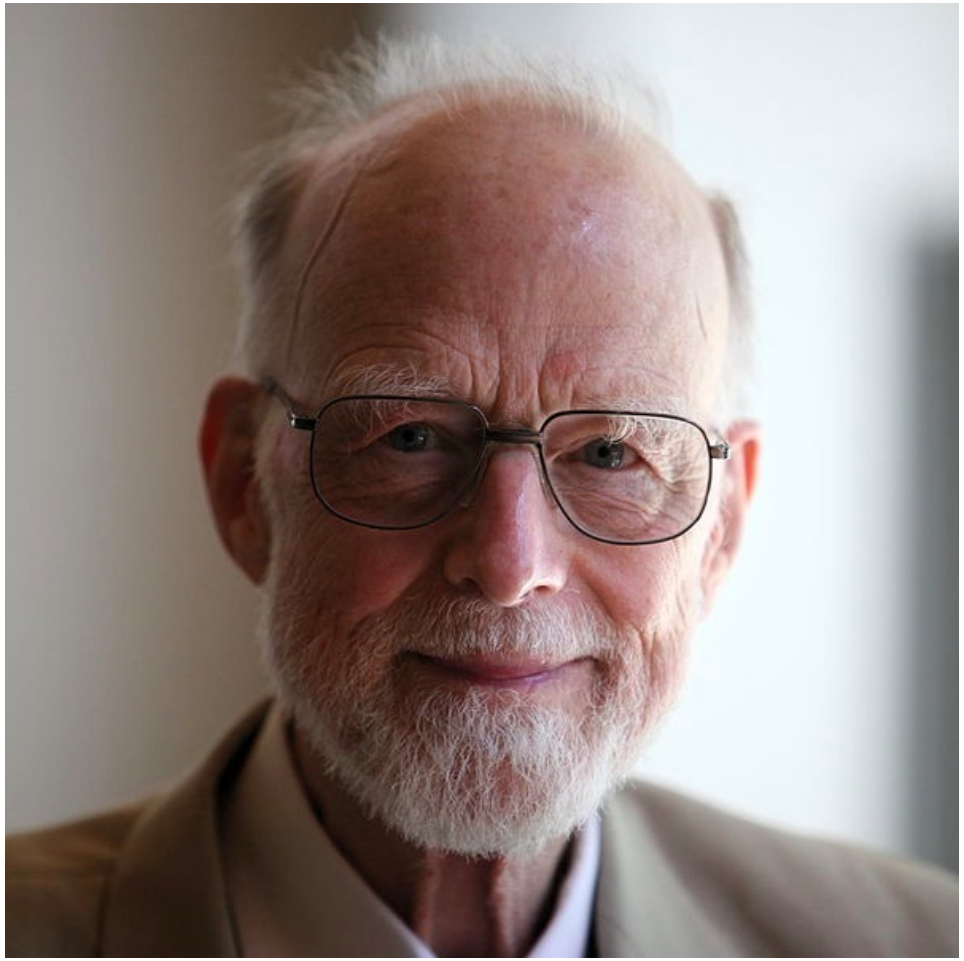


Photo: Wikimedia Commons (Rama)

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!

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

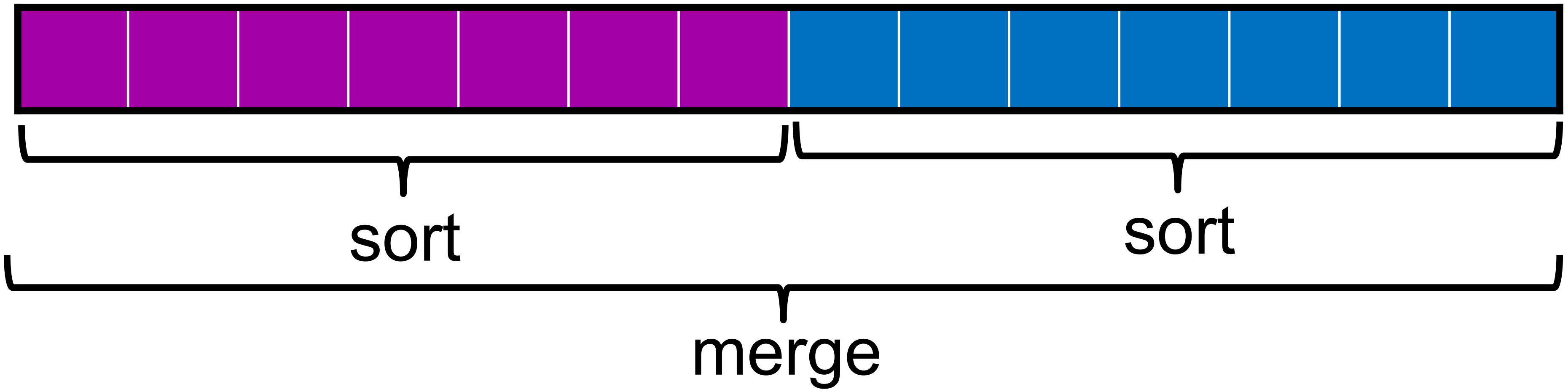
QuickSort

In class:

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

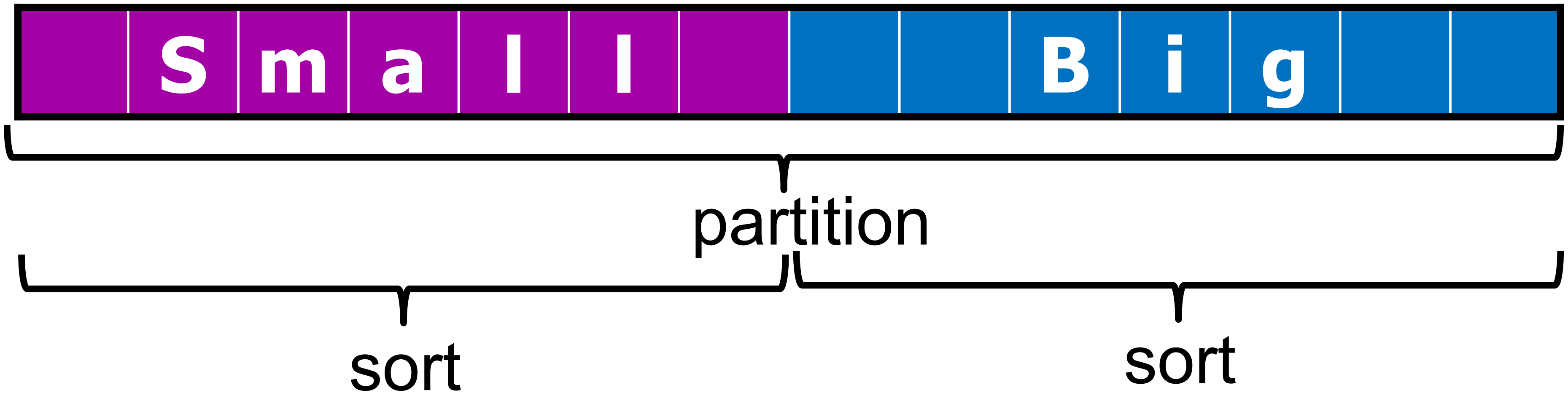
Recall: MergeSort

```
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)
```



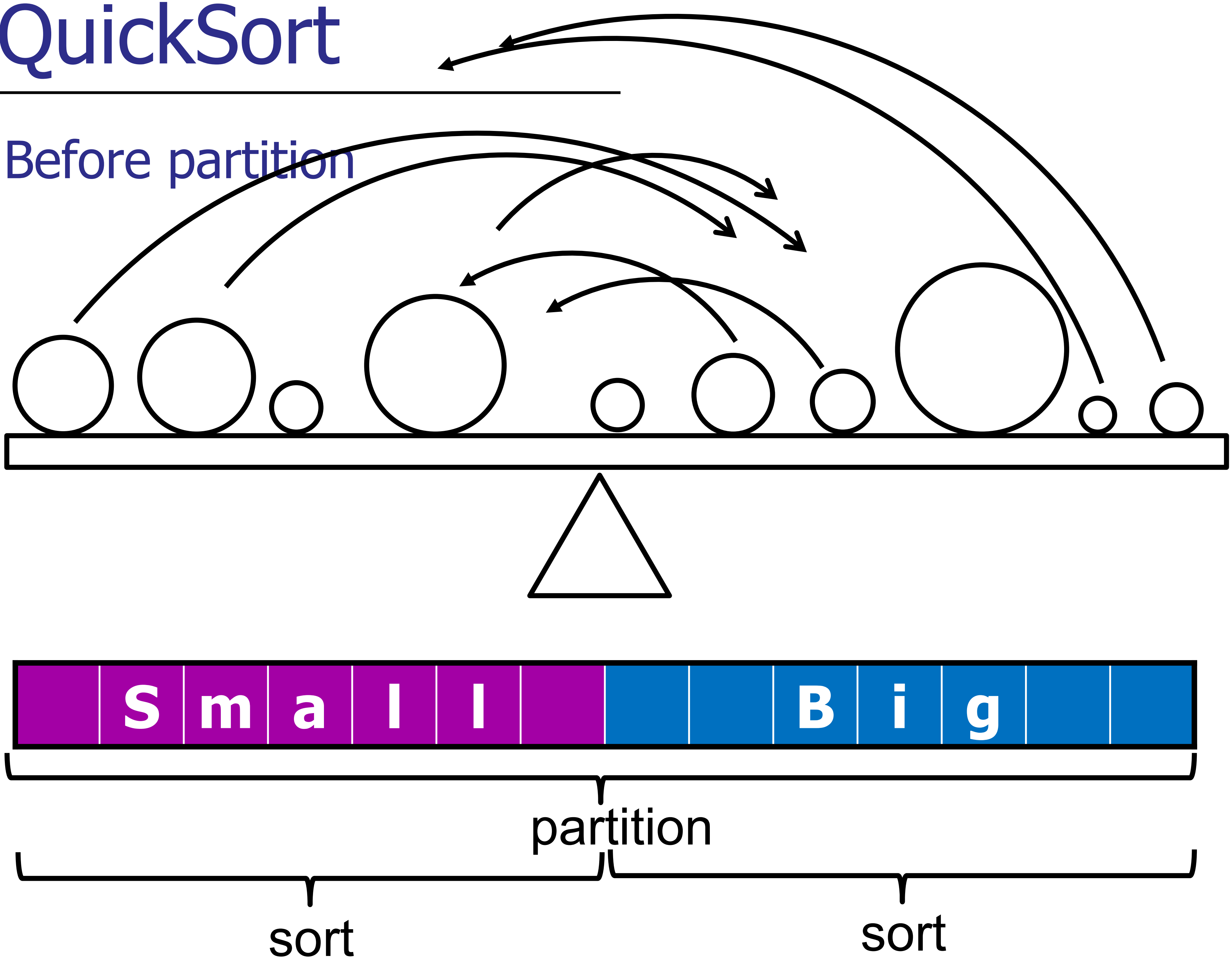
QuickSort

```
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)
```



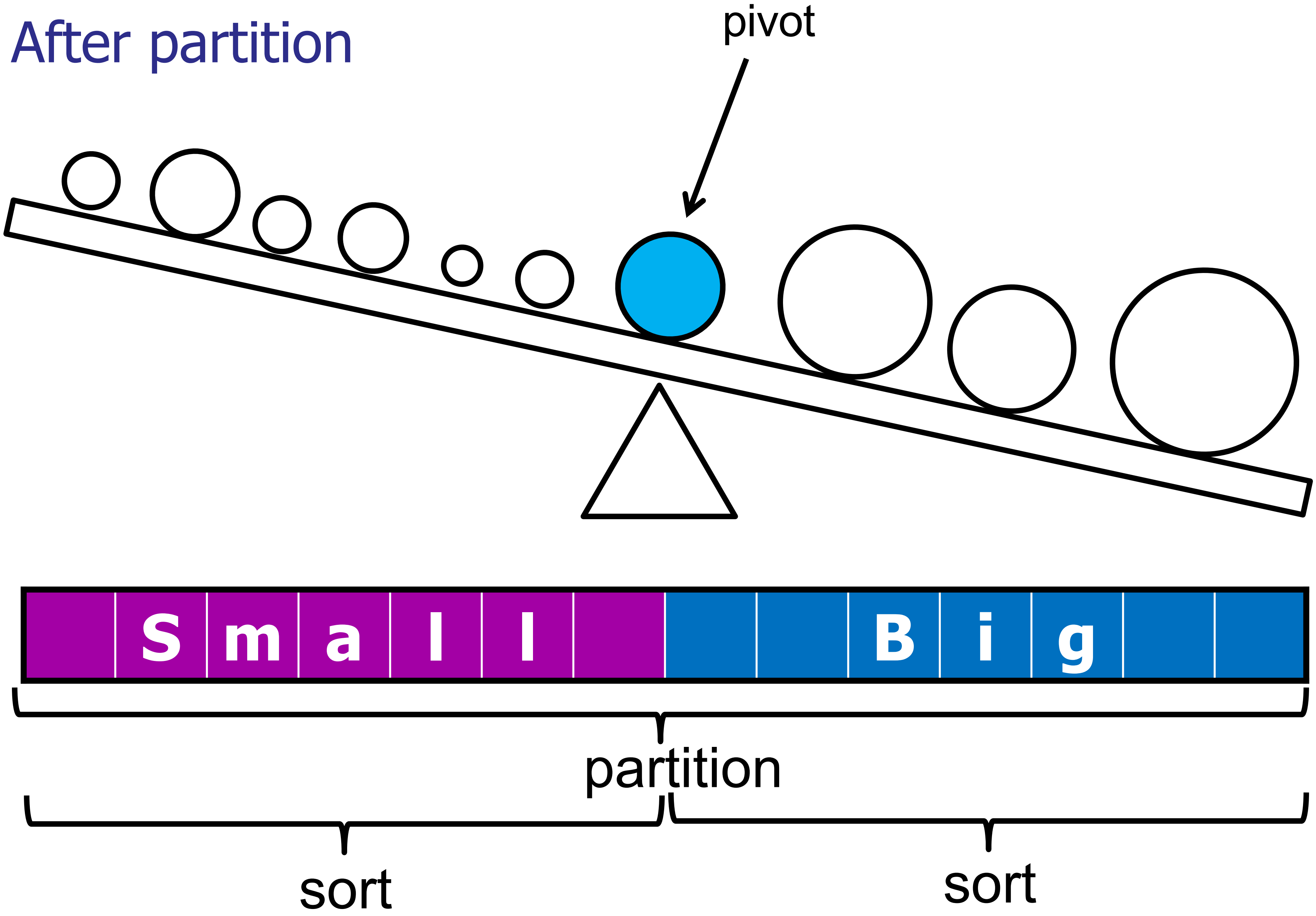
QuickSort

Before partition



QuickSort

After partition


pivot

S m a l l

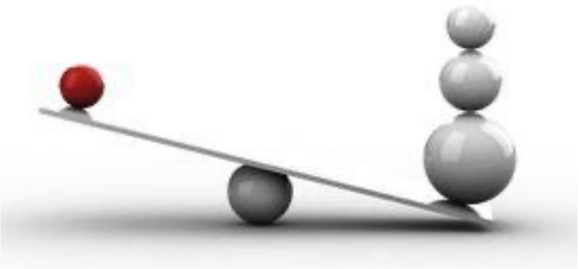
B i g

partition

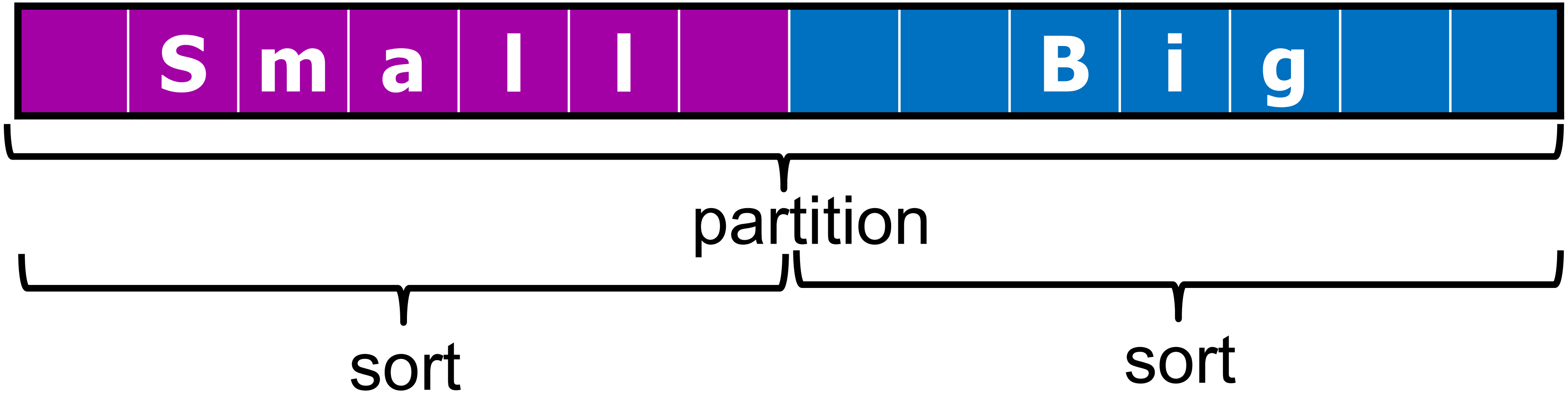
sort

sort

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)
```



S m a l l B i g

partition

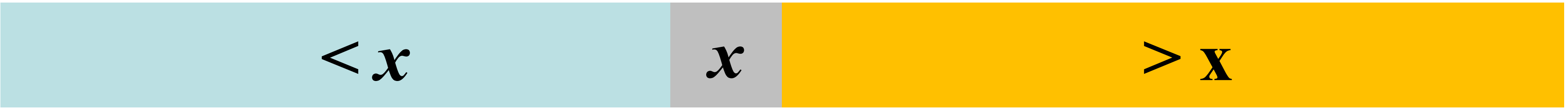
sort

sort

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.



$< \boldsymbol{x}$

$\boldsymbol{x}$

$> \mathbf{x}$

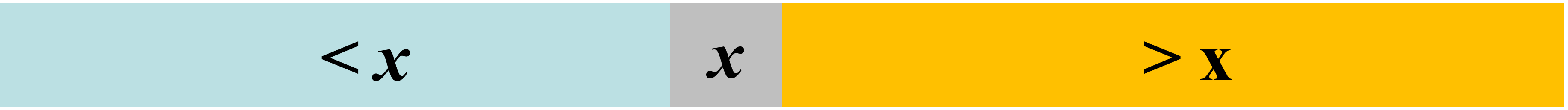
2. Conquer: Recursively sort the two sub-arrays.
3. Combine: Trivial, do nothing.

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 $> x$

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
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	50

a. 6

b. 10

✓ c. 22

d. 40

e. 32

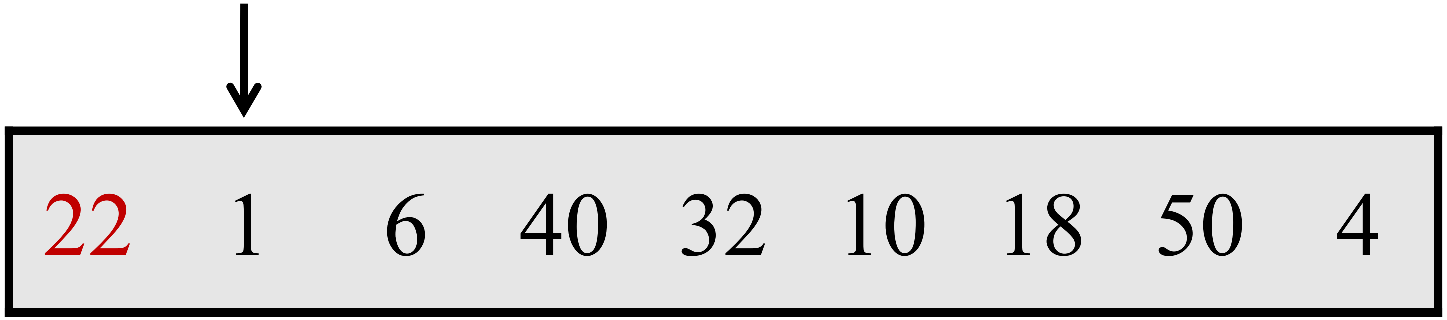
f. I don’t know.

ARCHIPELAGO

is open

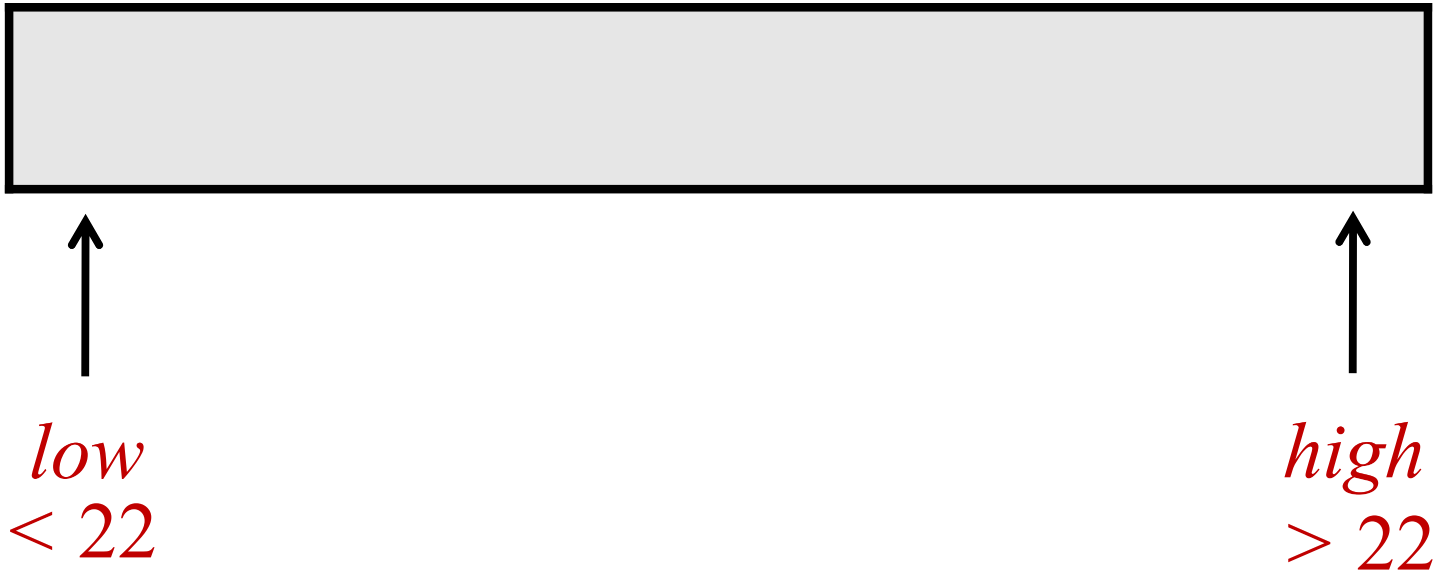
Partitioning an Array

Example: partition around 22



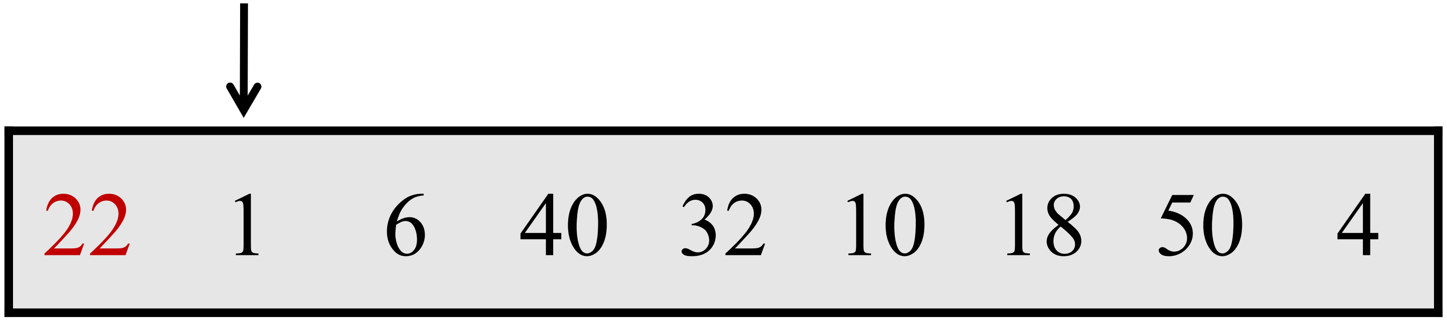
22 1 6 40 32 10 18 50 4

Output array:



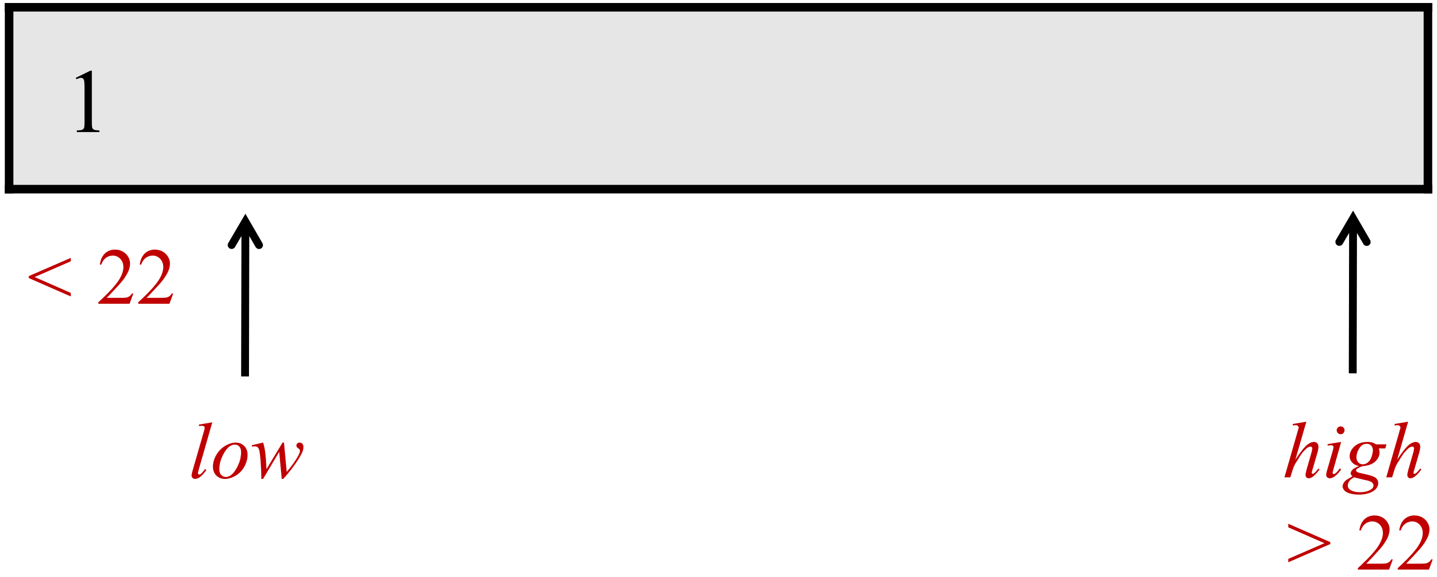
Partitioning an Array

Example: partition around 22



22 1 6 40 32 10 18 50 4

Output array:



1

< 22

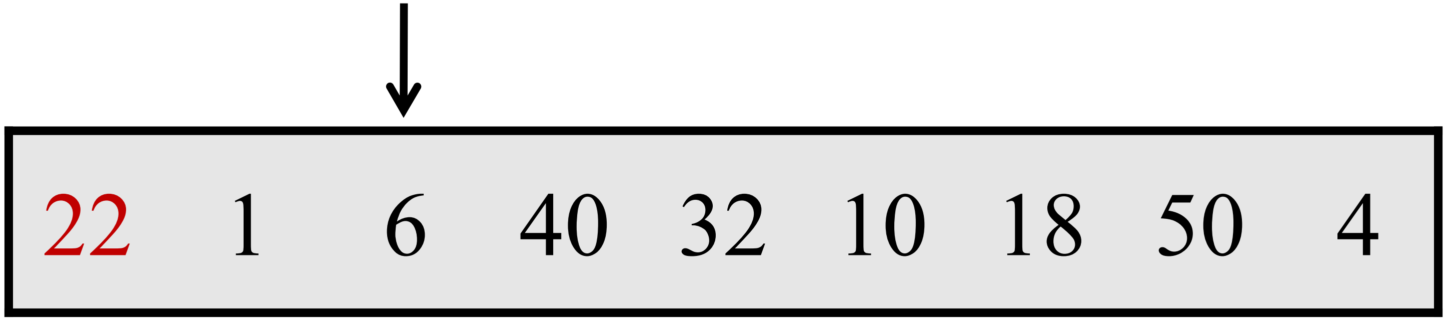
low

high

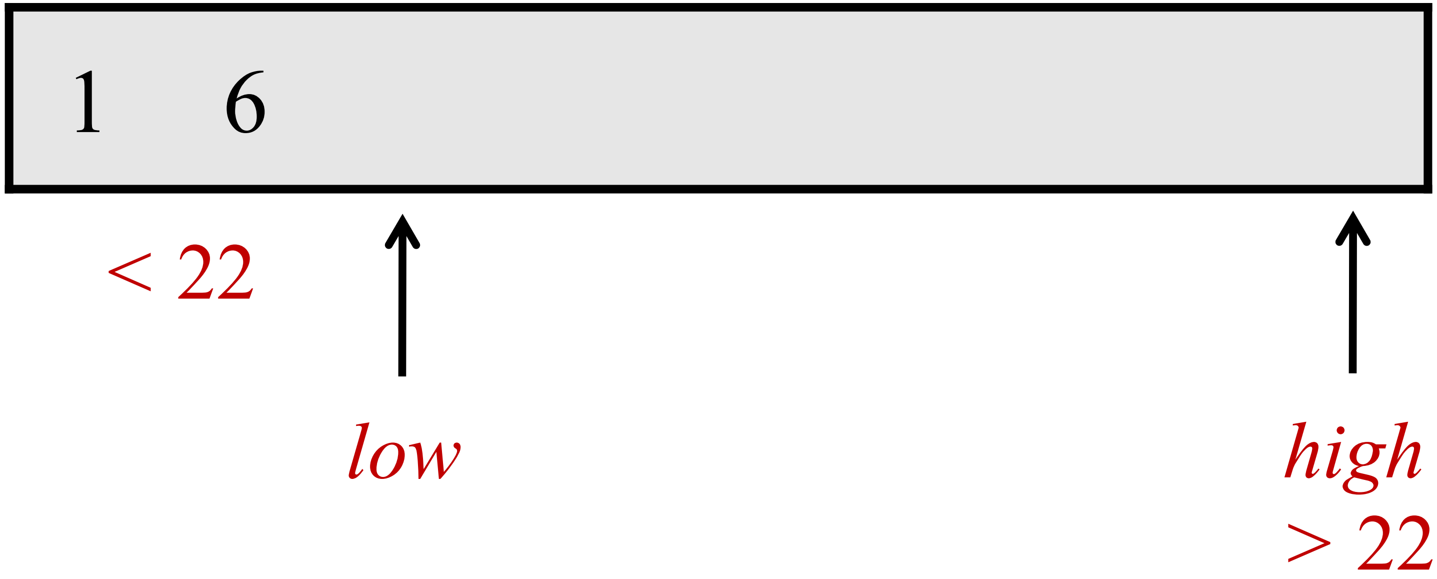
> 22

Partitioning an Array

Example: partition around 22

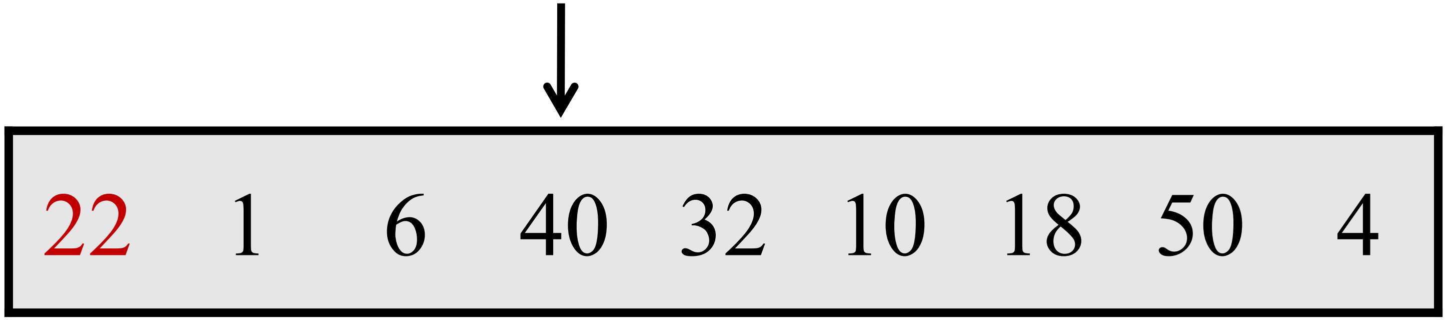


Output array:



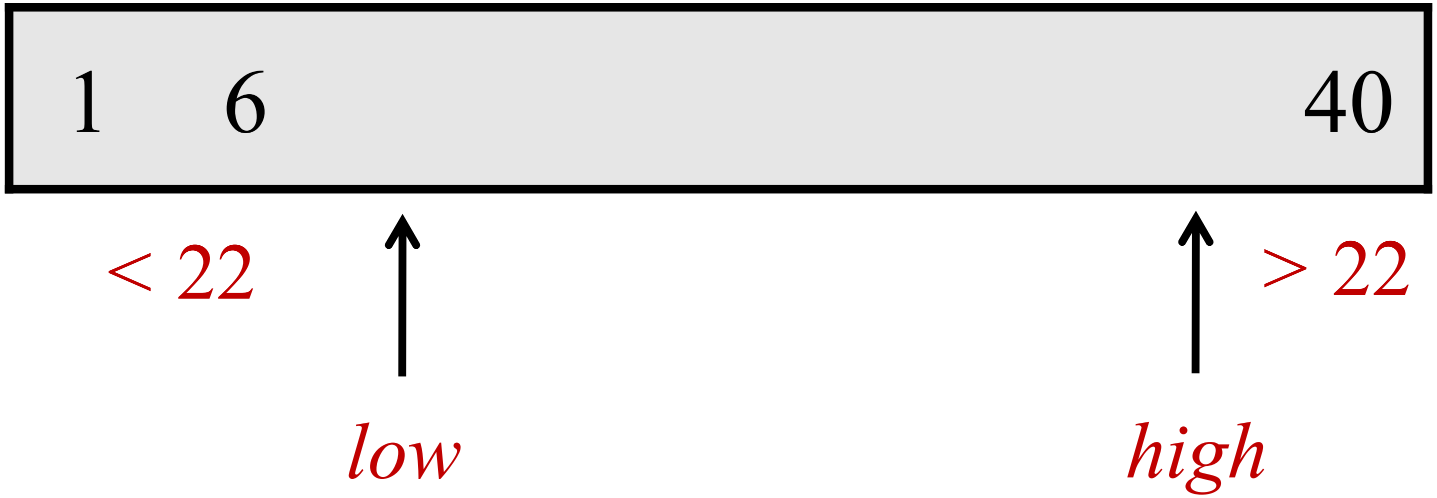
Partitioning an Array

Example: partition around 22



22	1	6	40	32	10	18	50	4

Output array:



1	6							40

< 22

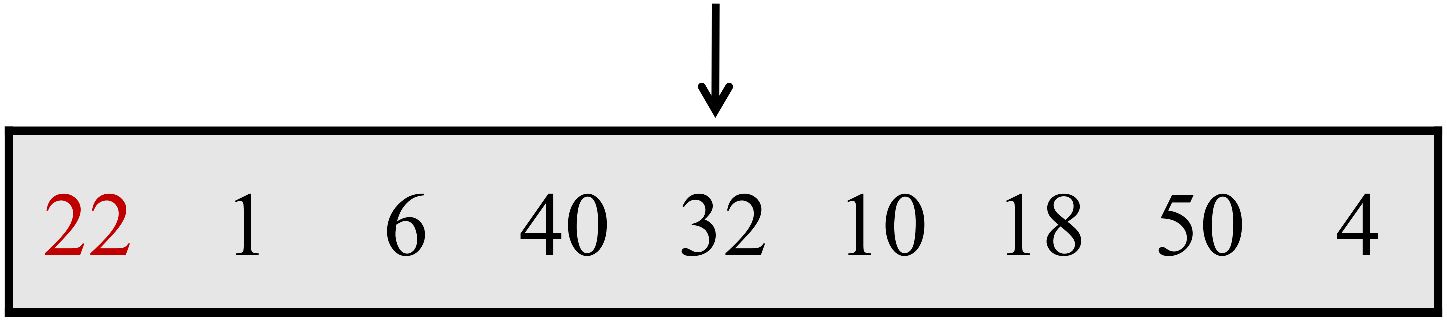
low

high

> 22

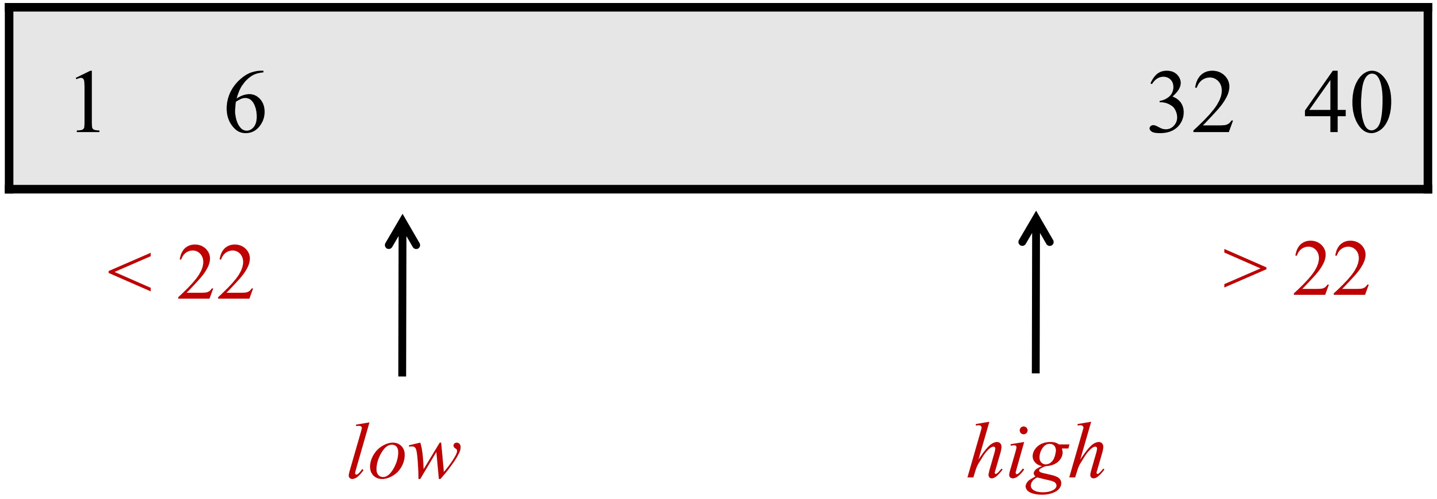
Partitioning an Array

Example: partition around 22



22 1 6 40 32 10 18 50 4

Output array:



1 6 32 40

< 22

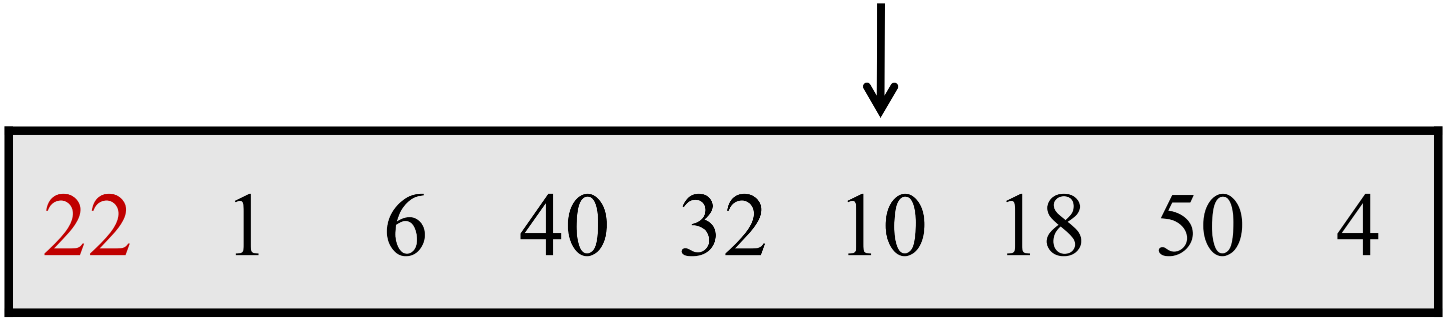
> 22

low

high

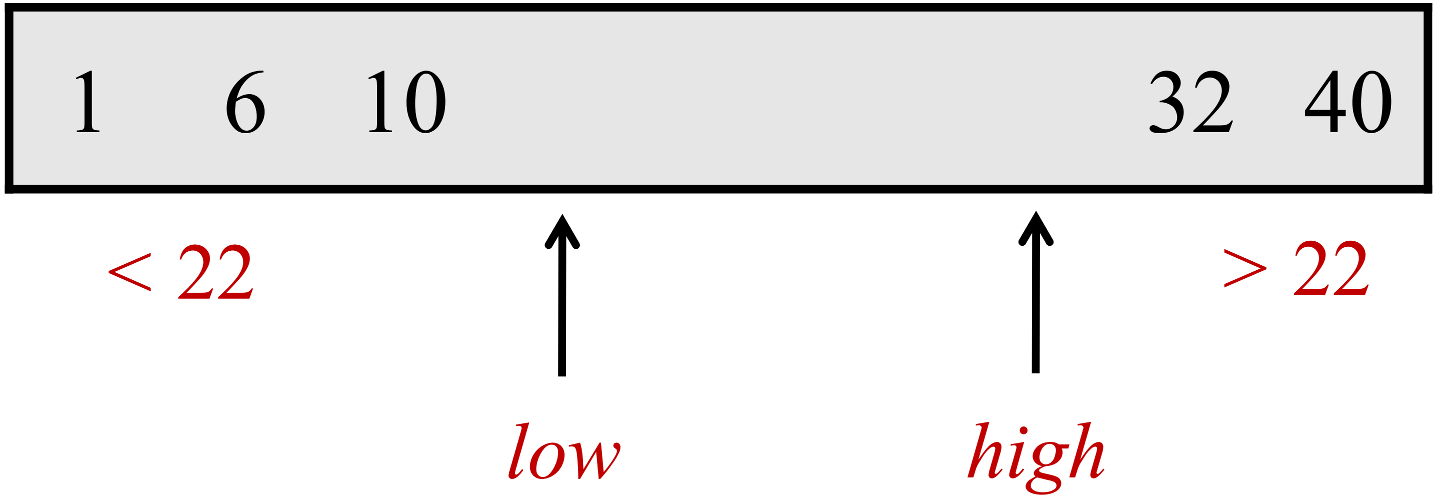
Partitioning an Array

Example: partition around 22



22 1 6 40 32 10 18 50 4

Output array:



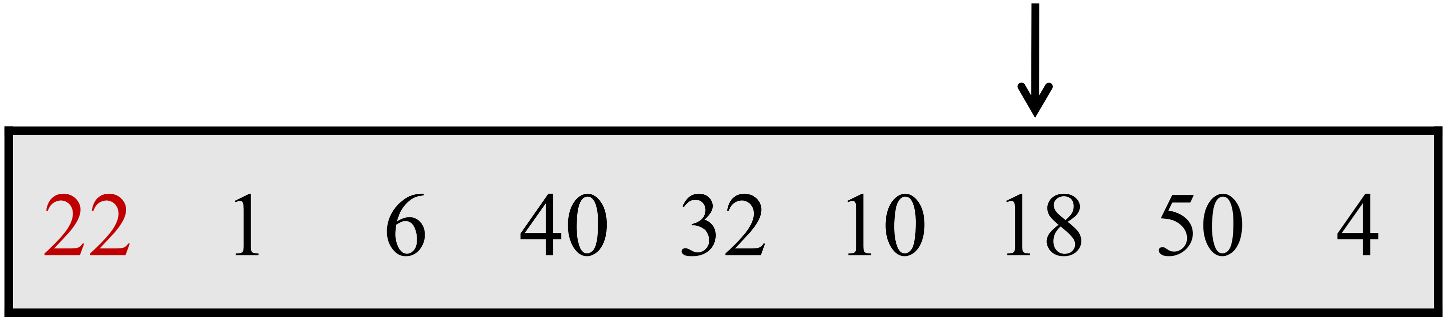
1 6 10 32 40

< 22

> 22

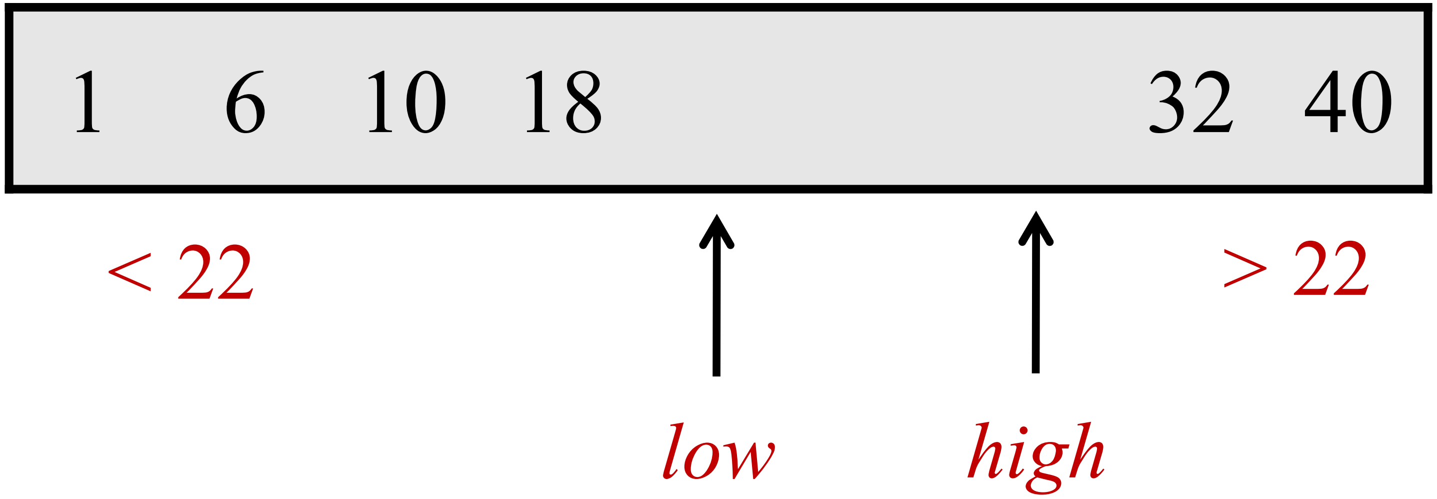
Partitioning an Array

Example: partition around 22



22 1 6 40 32 10 18 50 4

Output array:



1 6 10 18 32 40

< 22

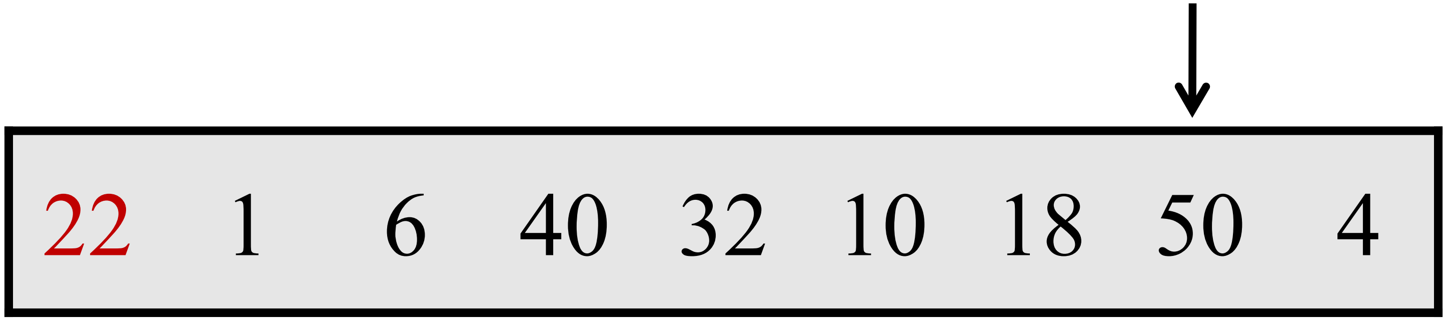
low

high

> 22

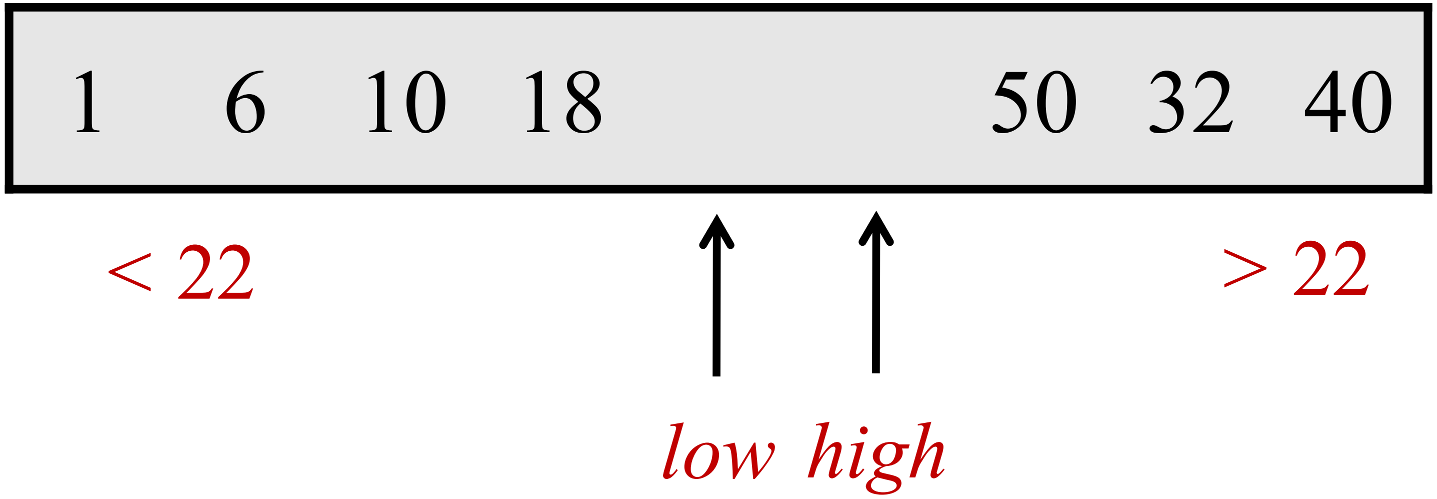
Partitioning an Array

Example: partition around 22



22 1 6 40 32 10 18 50 4

Output array:



1 6 10 18 50 32 40

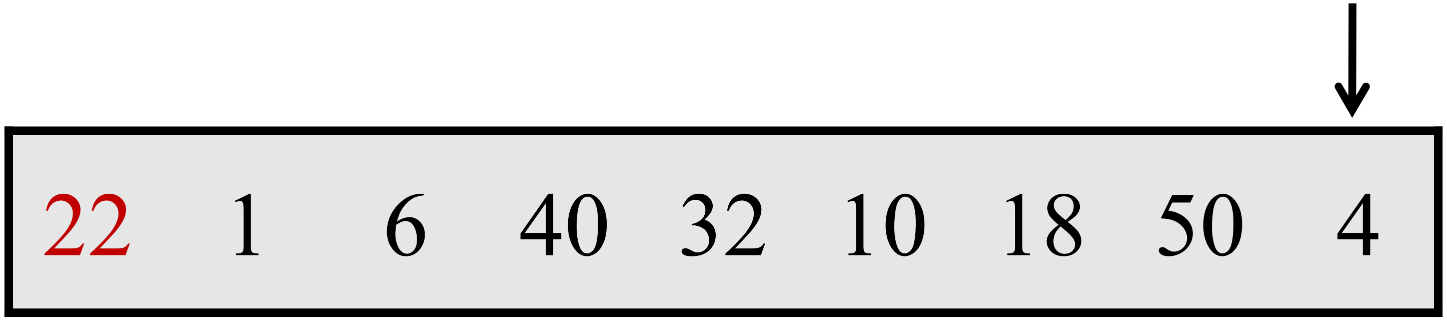
< 22

> 22

low *high*

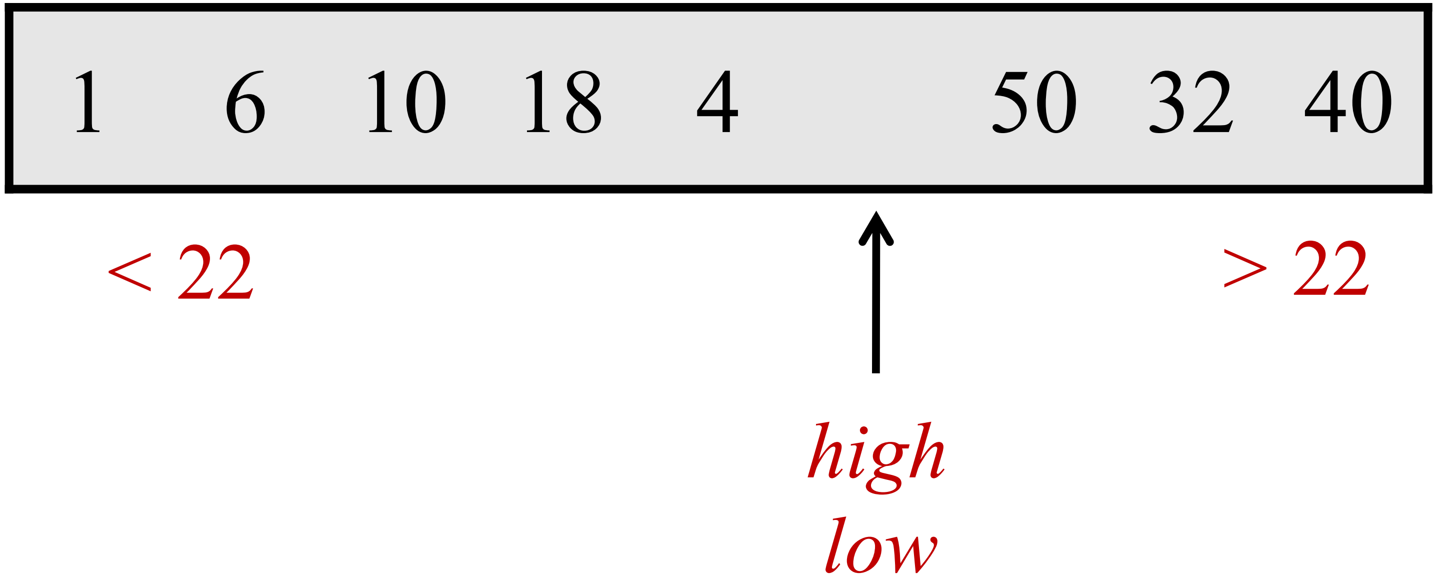
Partitioning an Array

Example: partition around 22



22 1 6 40 32 10 18 50 4

Output array:



1 6 10 18 4 50 32 40

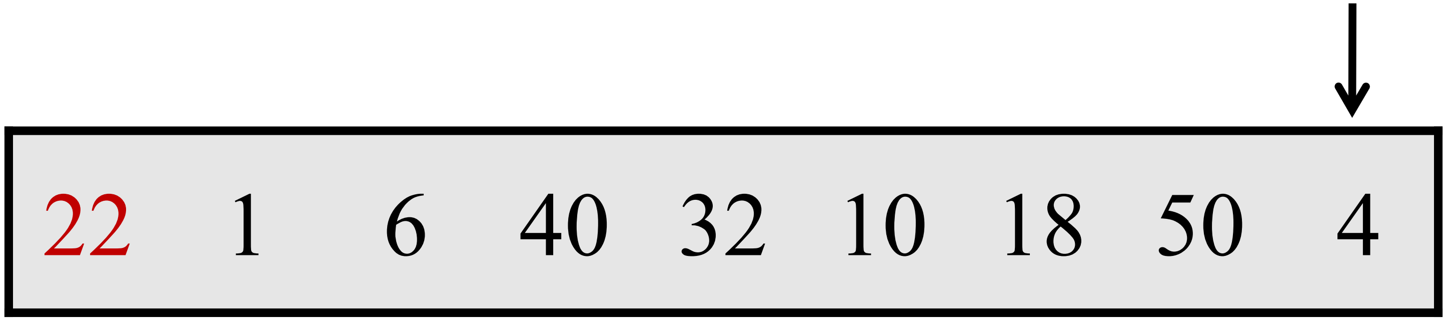
< 22

> 22

high
low

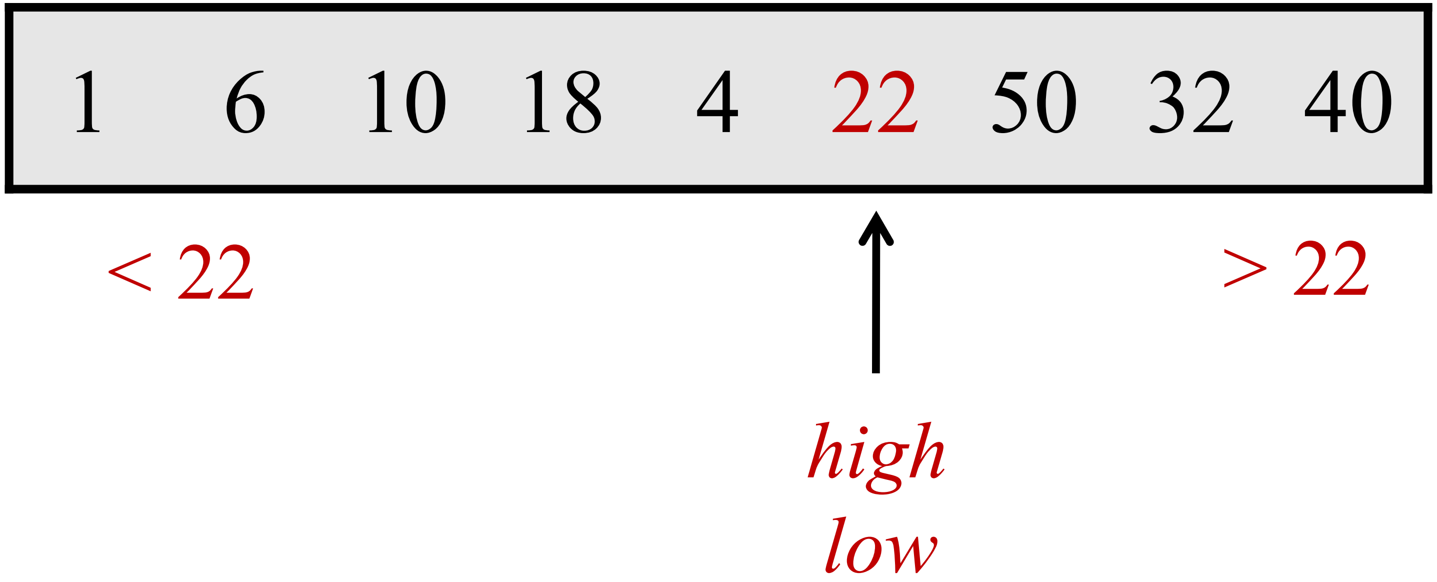
Partitioning an Array

Example: partition around 22



22	1	6	40	32	10	18	50	4

Output array:



1	6	10	18	4	22	50	32	40

< 22

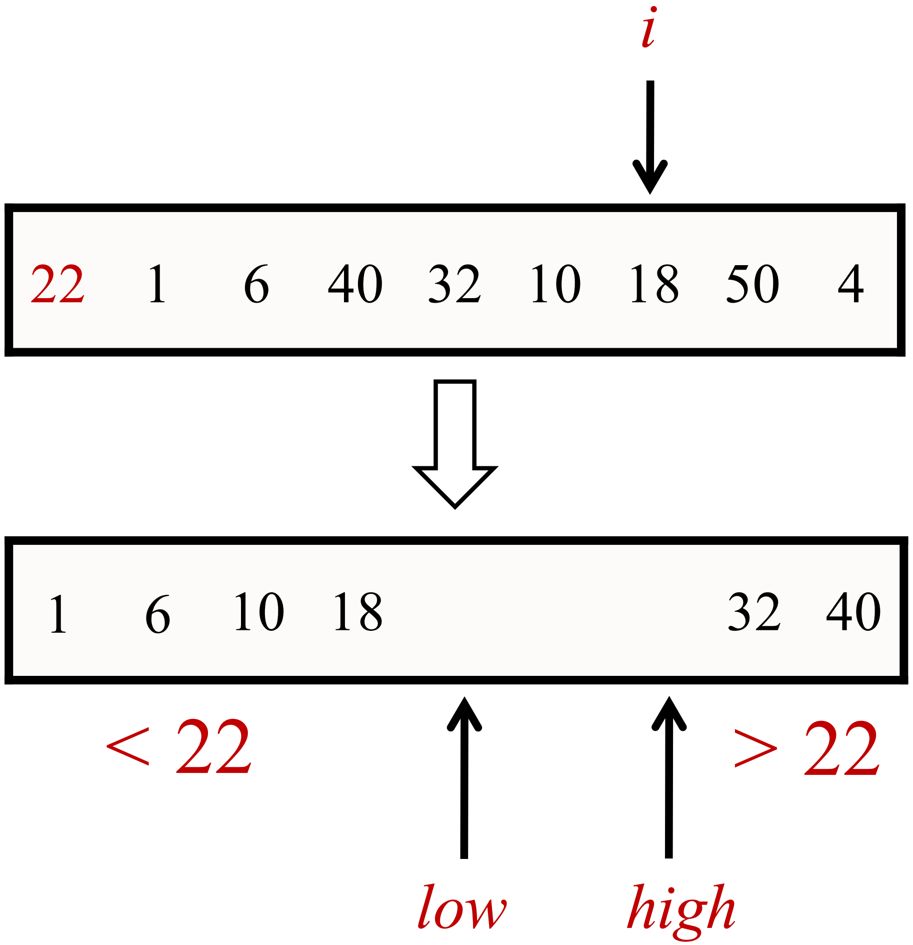
> 22

high

low

```
partition(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?

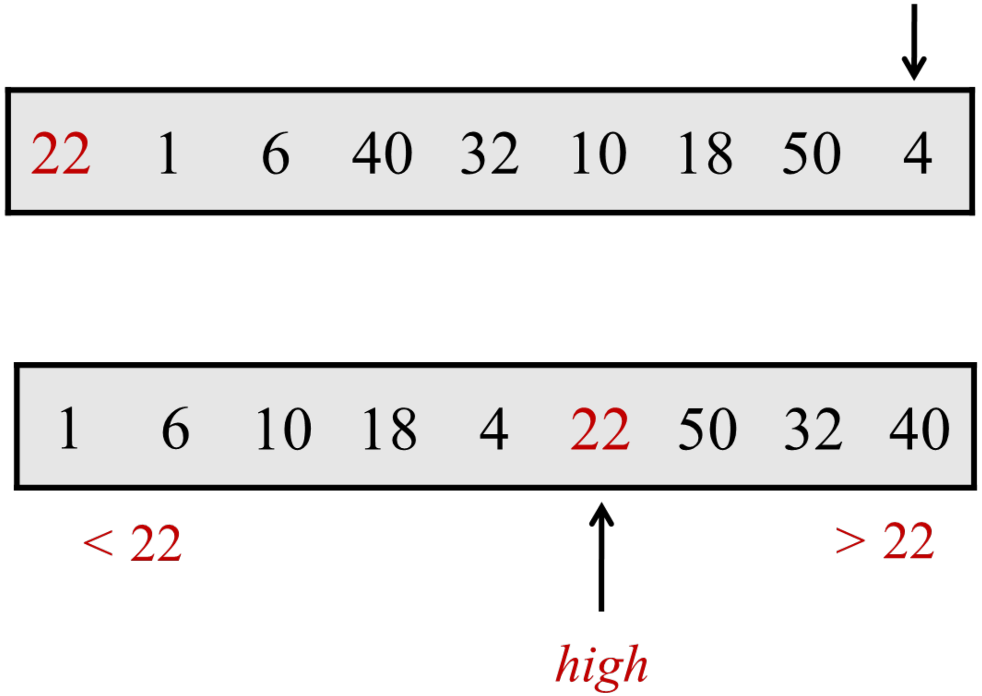
1. $O(\log n)$
2. ✓ $O(n)$
3. $O(n \log n)$
4. $O(n^2)$
5. I have no idea.

ARCHIPELAGO

is open

Any bugs?

Anything that can be improved?



22 1 6 40 32 10 18 50 4

1 6 10 18 4 22 50 32 40

< 22

> 22

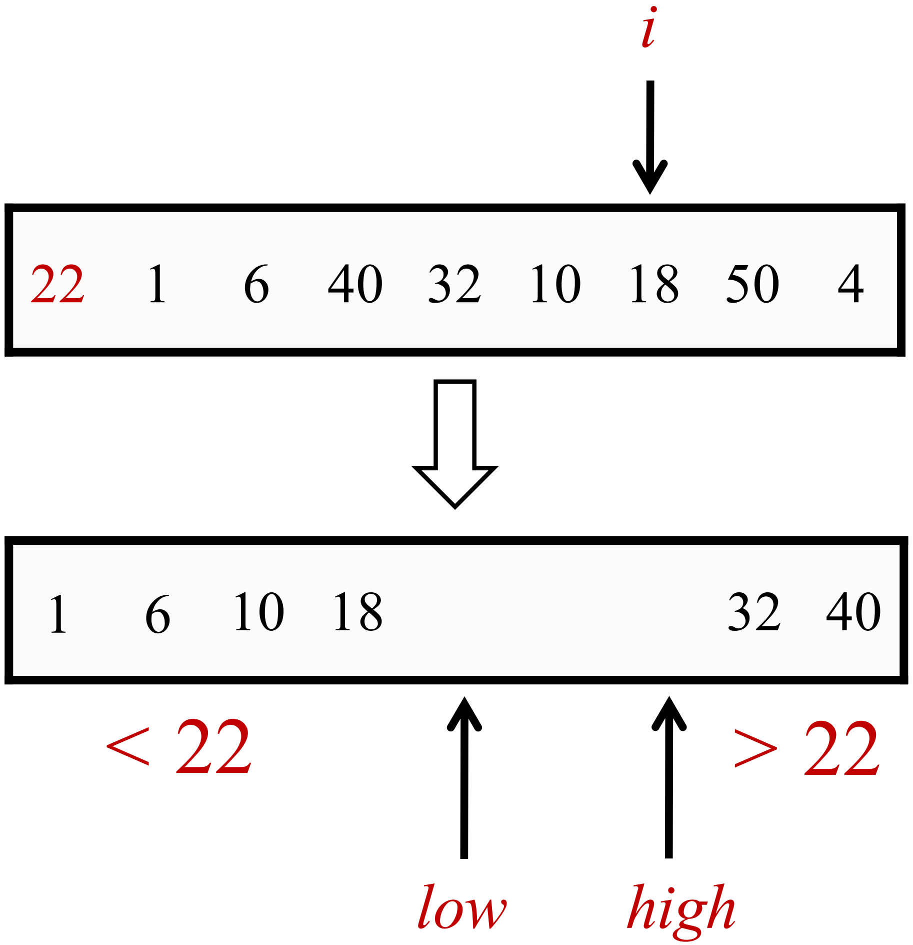
high

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



i

22 1 6 40 32 10 18 50 4

1 6 10 18 32 40

< 22

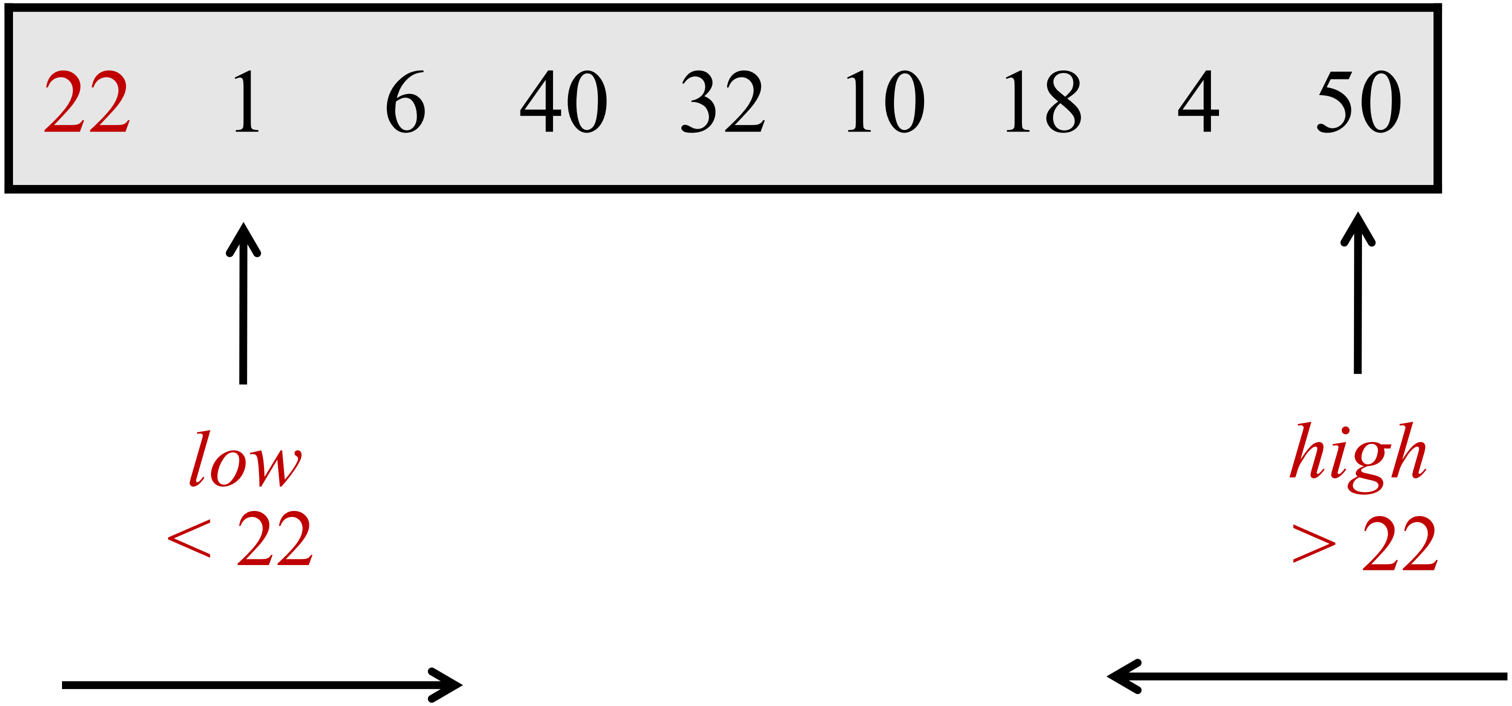
> 22

low

high

Partitioning an Array "in-place"

Example: partition around 22



22	1	6	40	32	10	18	4	50

low
< 22

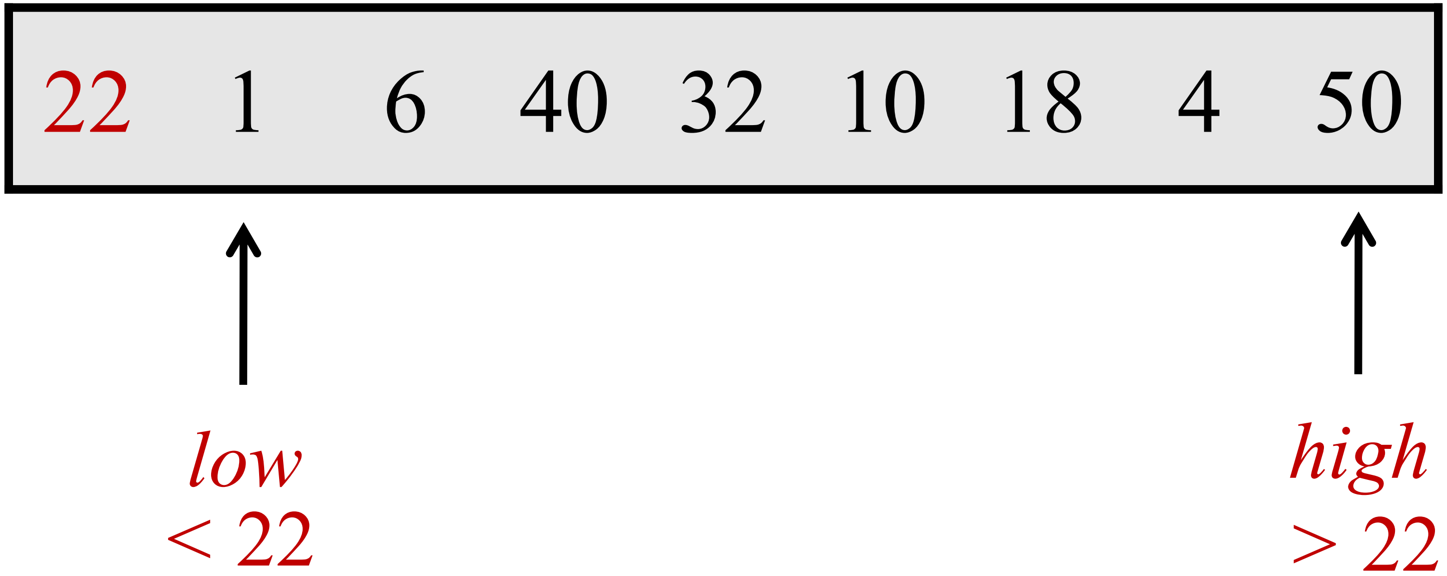
high
> 22

Move until it's bigger than the pivot

Move until it's less than the pivot

Partitioning an Array

Example: partition around 22



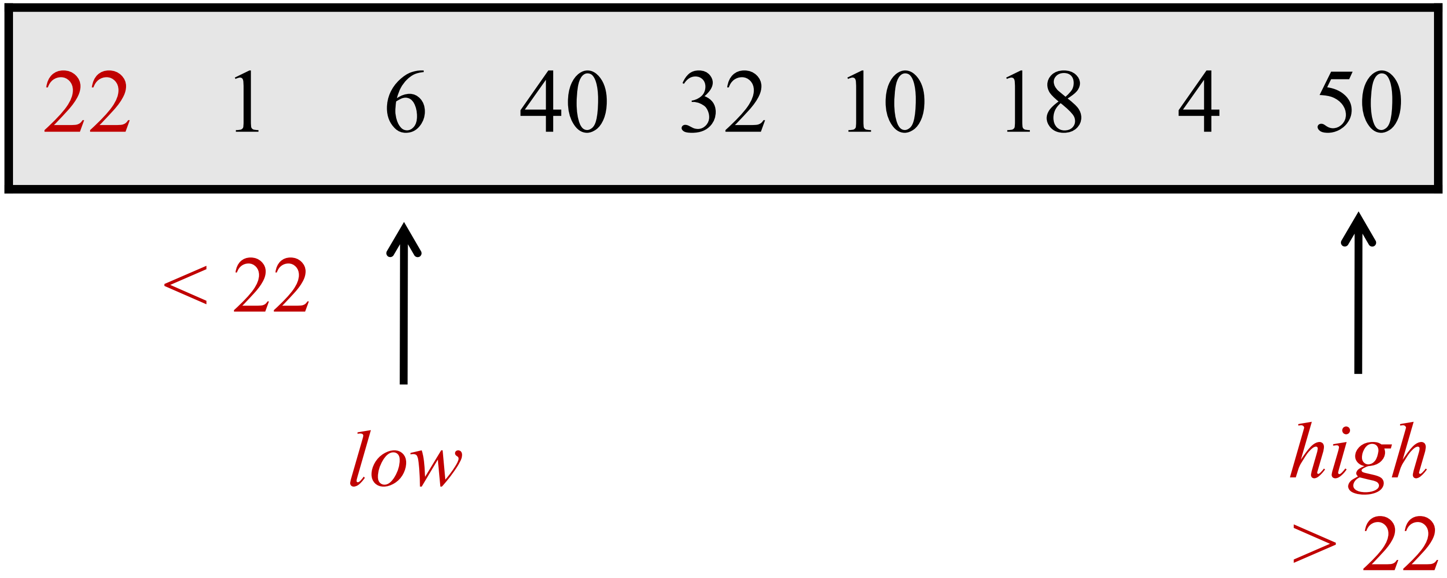
22	1	6	40	32	10	18	4	50

low
< 22

high
> 22

Partitioning an Array

Example: partition around 22



22	1	6	40	32	10	18	4	50

< 22

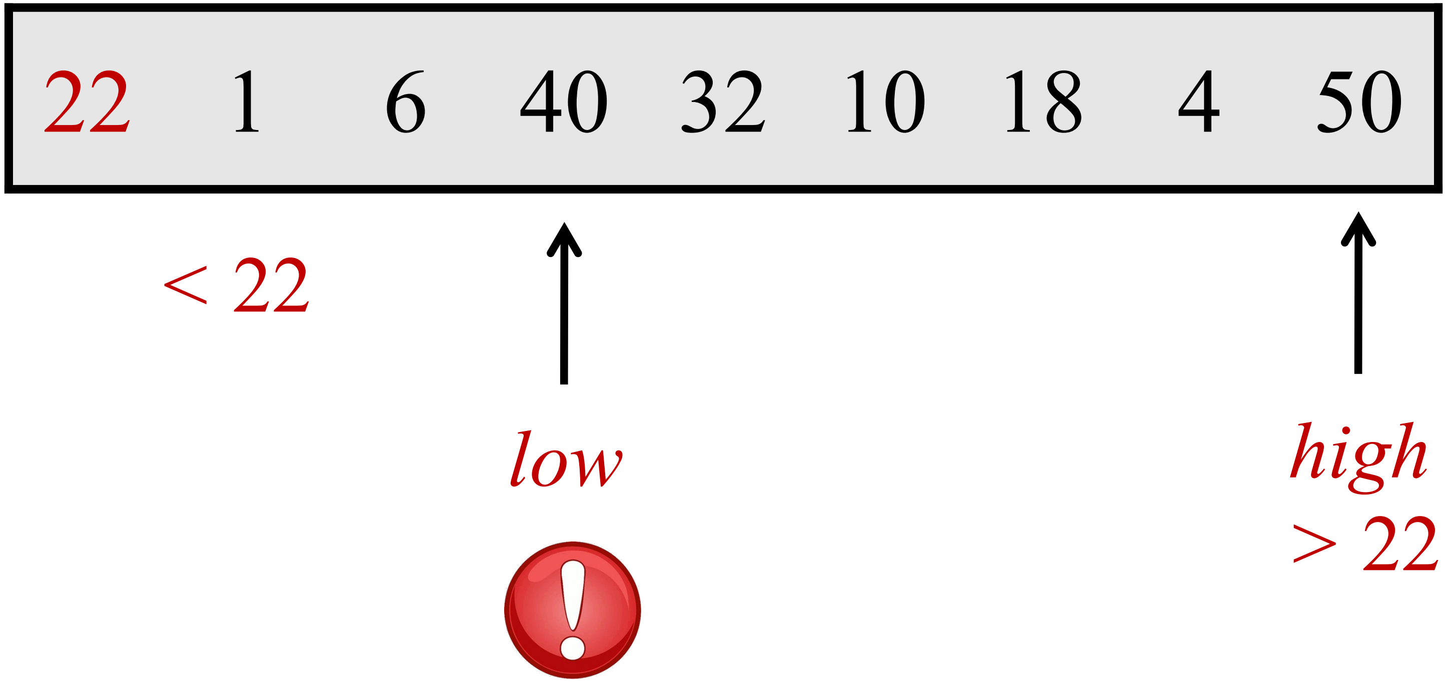
low

high

> 22

Partitioning an Array

Example: partition around 22



22 1 6 40 32 10 18 4 50

< 22

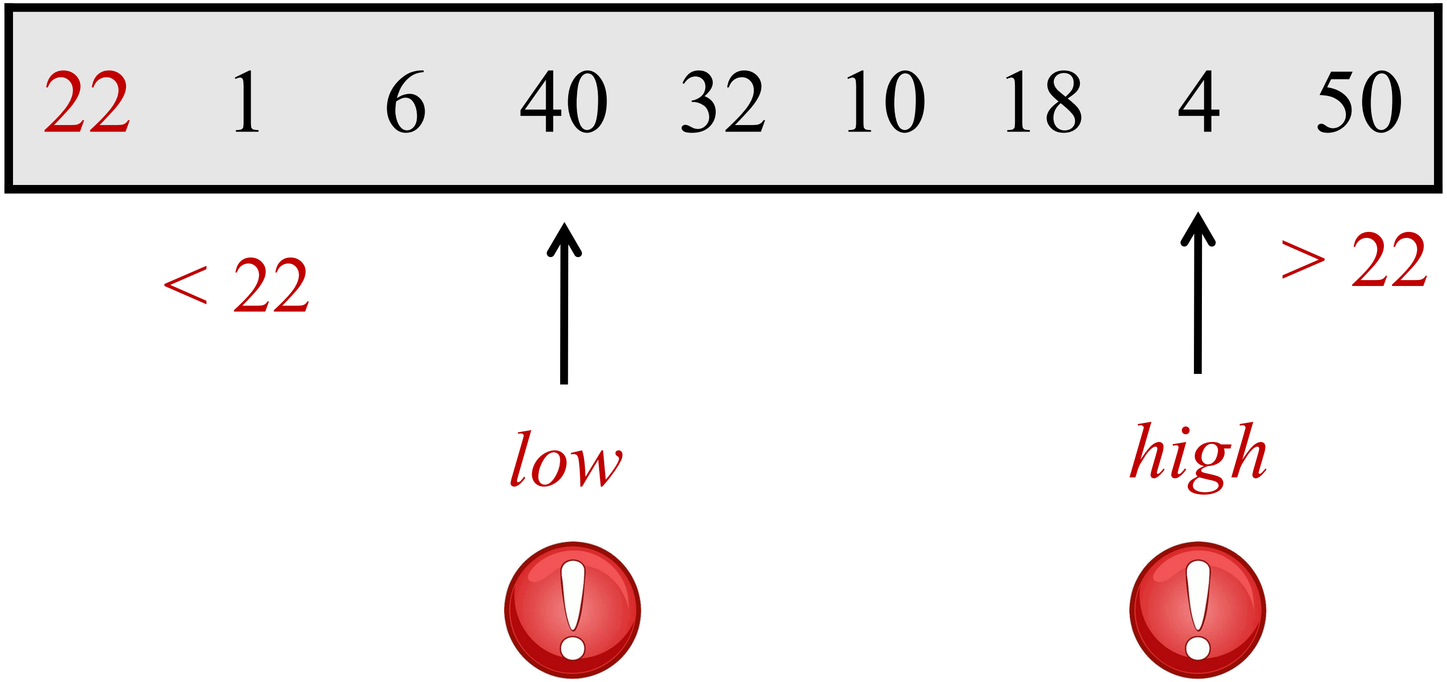
low

high

> 22

Partitioning an Array

Example: partition around 22



22 1 6 40 32 10 18 4 50

< 22

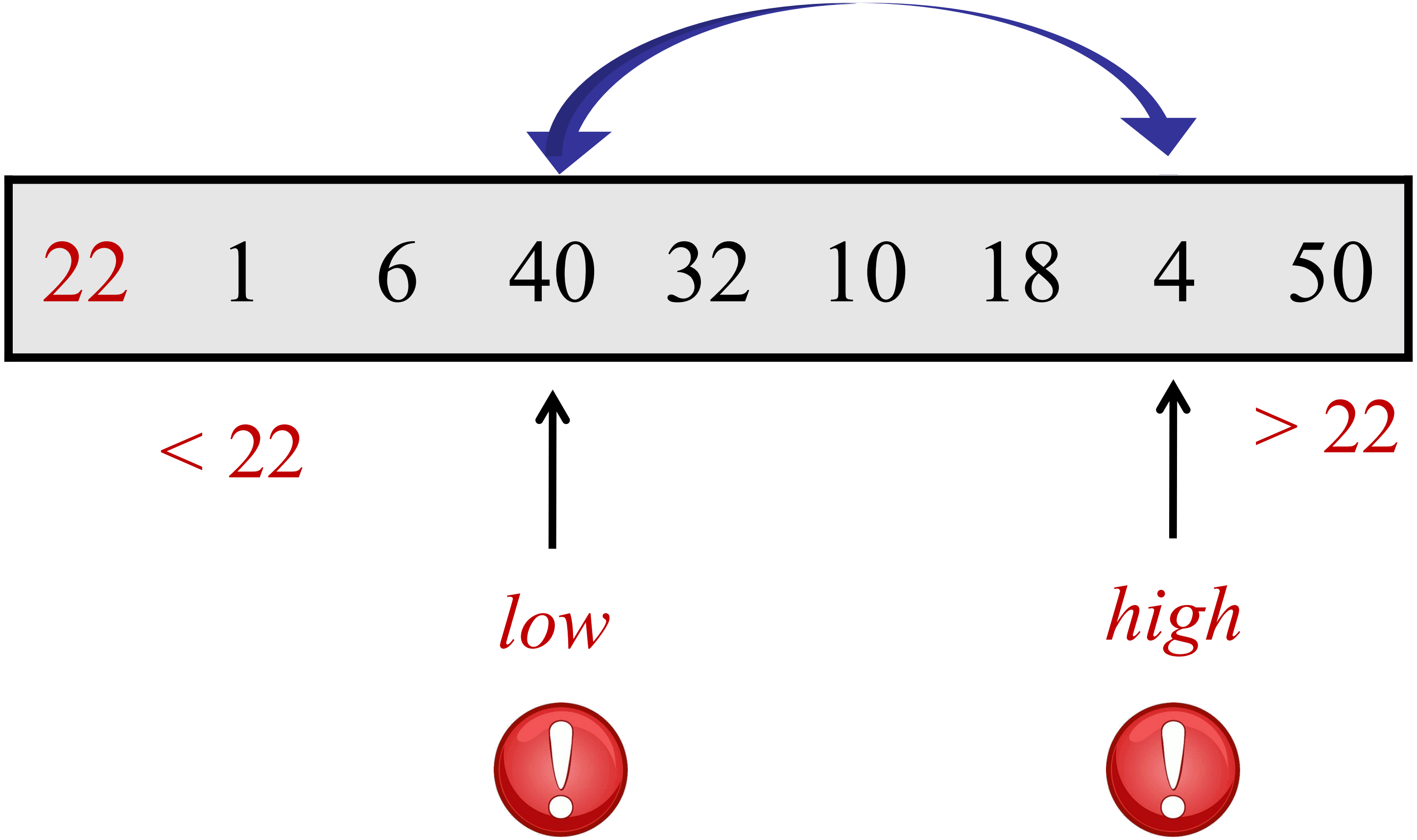
> 22

low

high

Partitioning an Array

Example: partition around 22



22	1	6	40	32	10	18	4	50

< 22

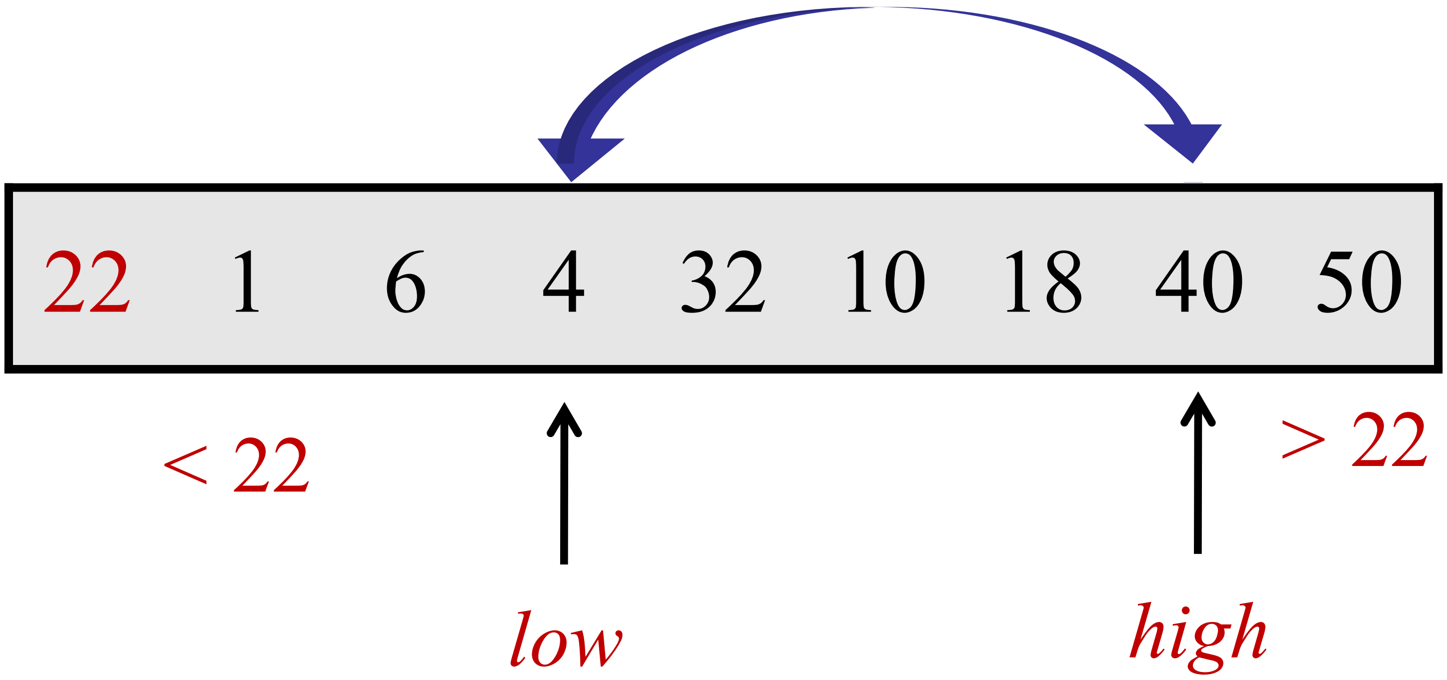
low

high

> 22

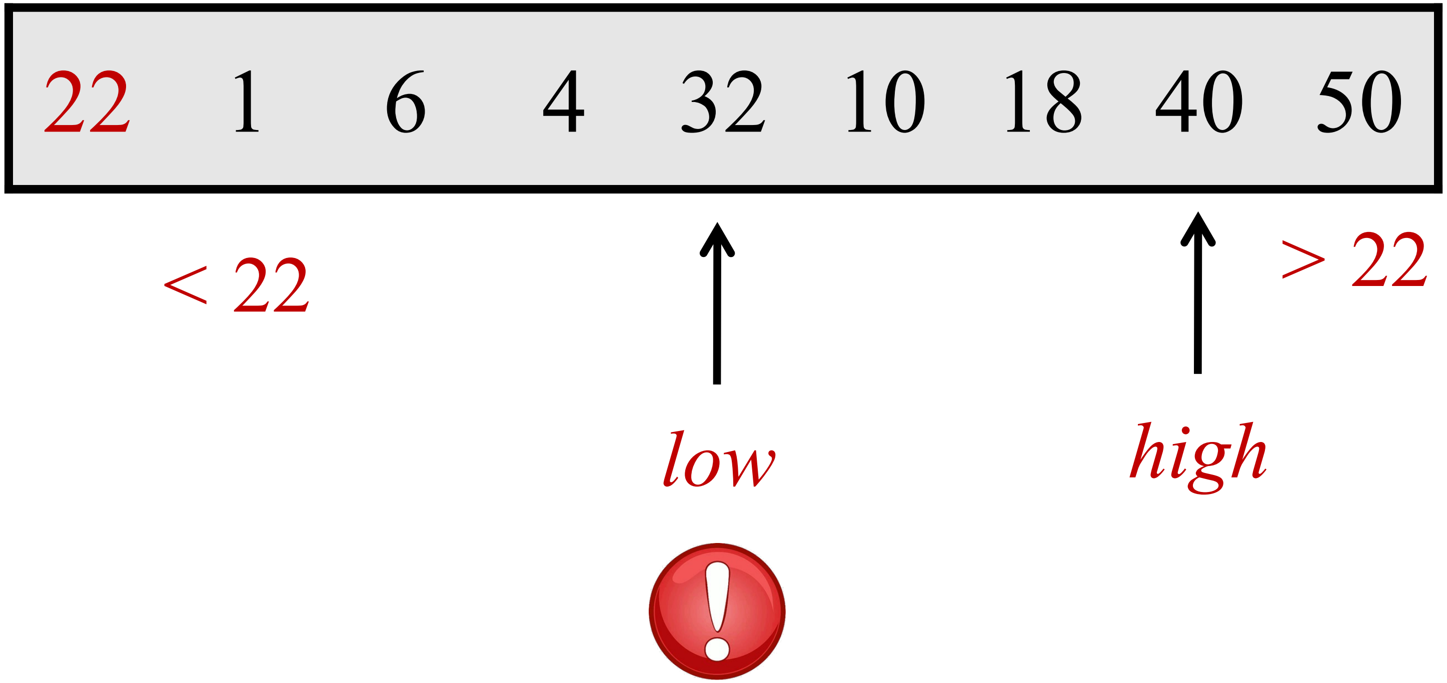
Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22



22 1 6 4 32 10 18 40 50

< 22

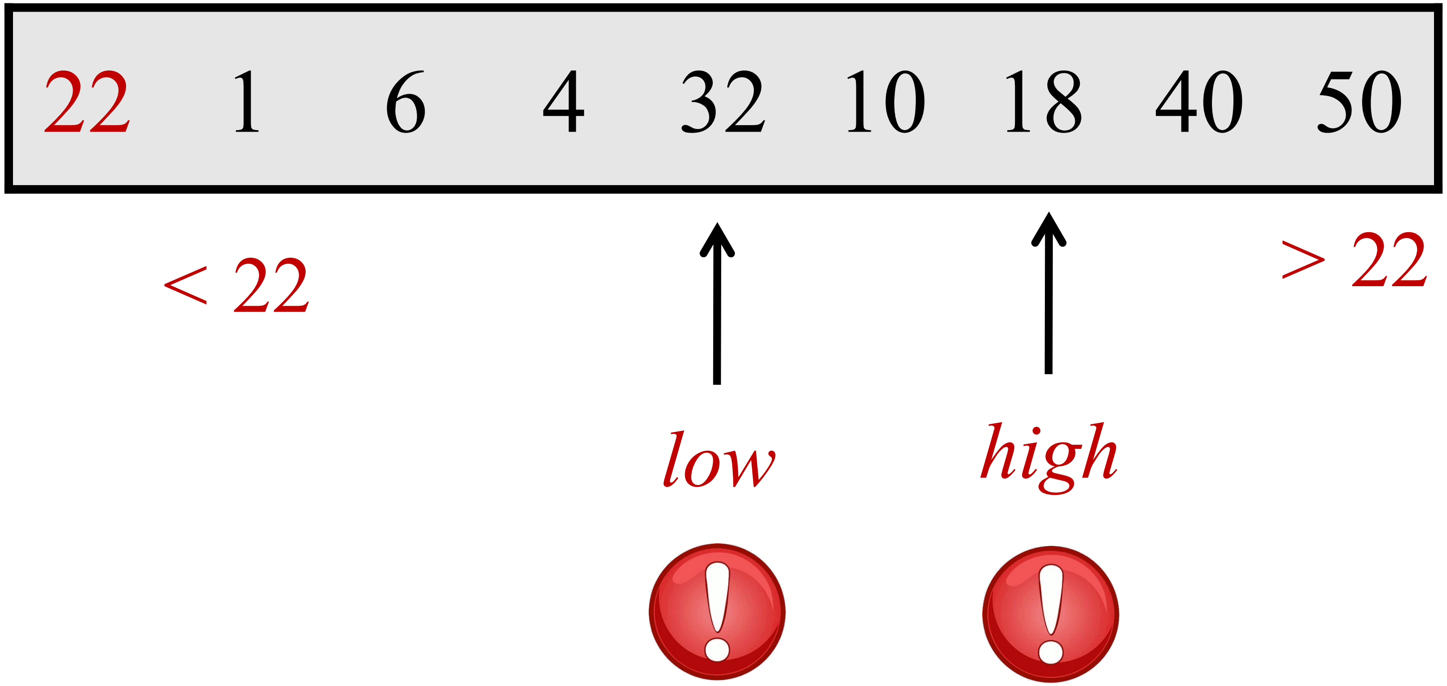
> 22

low

high

Partitioning an Array

Example: partition around 22



22	1	6	4	32	10	18	40	50

< 22

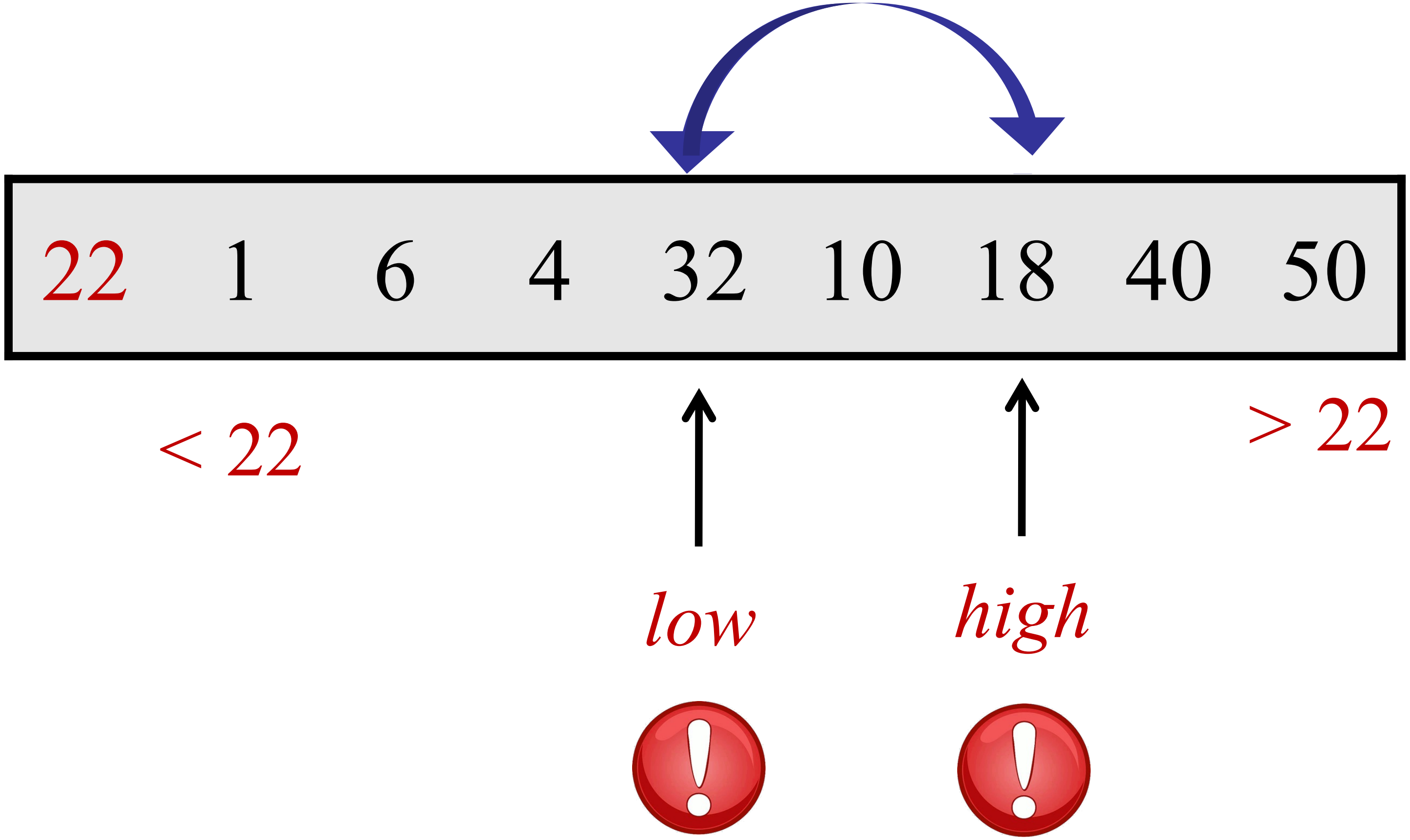
> 22

low

high

Partitioning an Array

Example: partition around 22



22 1 6 4 32 10 18 40 50

< 22

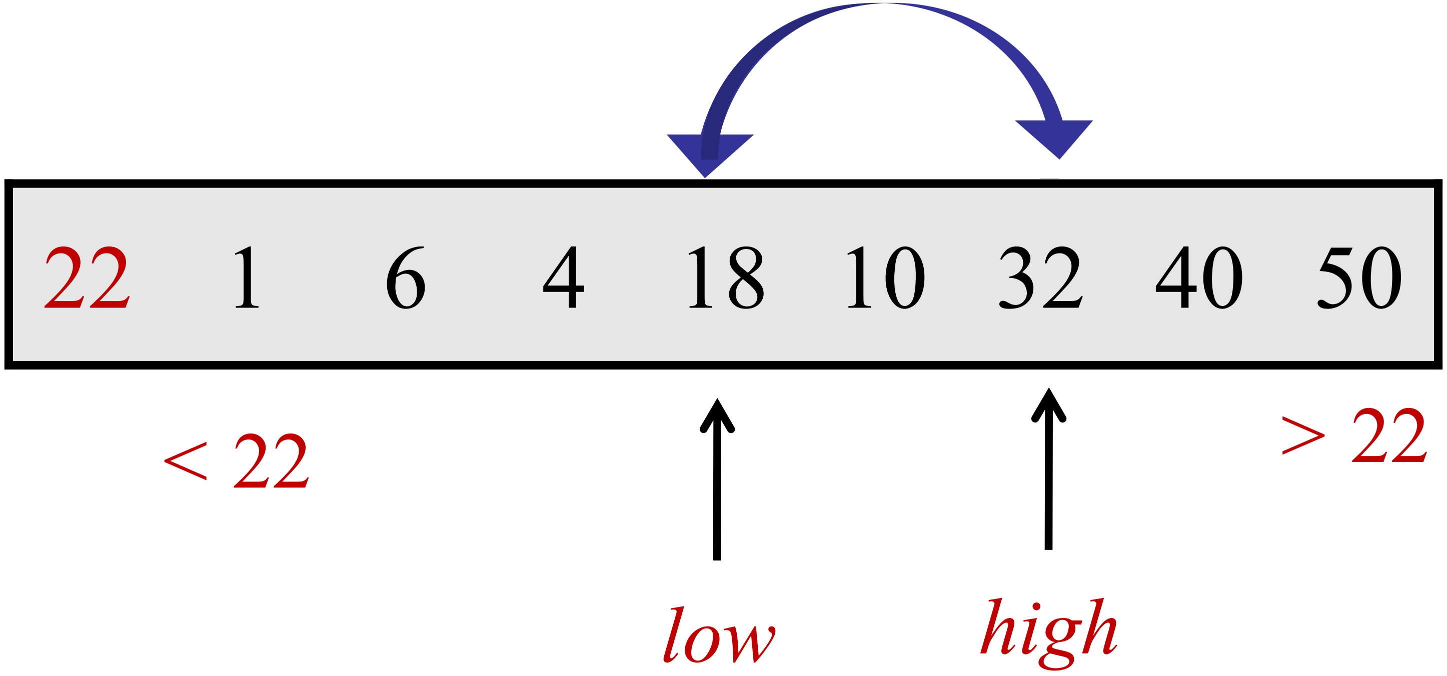
> 22

low

high

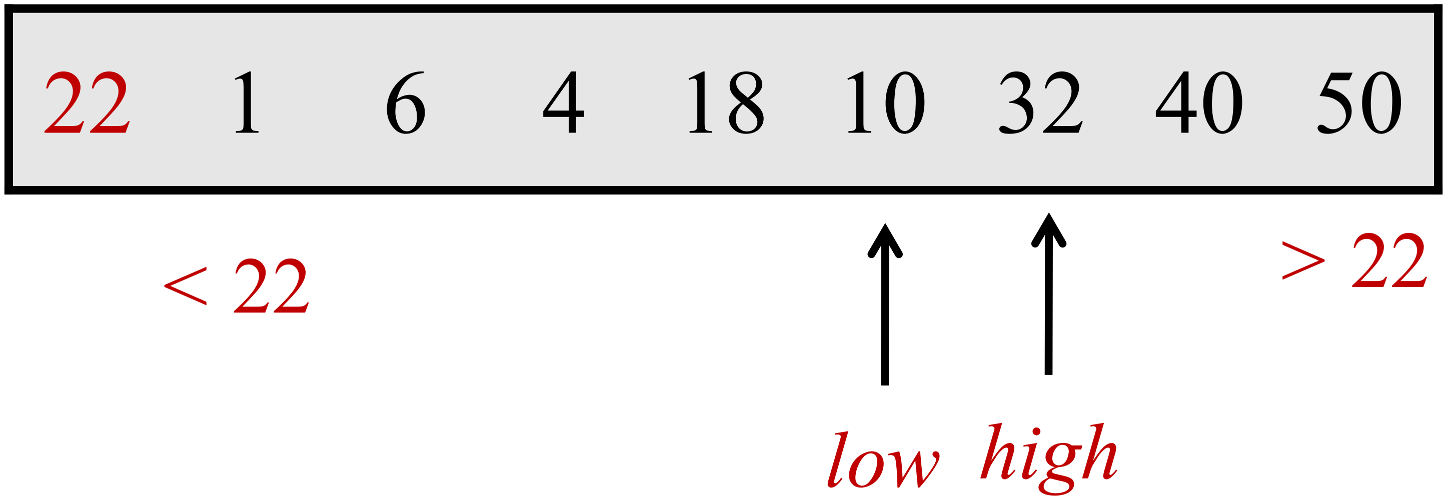
Partitioning an Array

Example: partition around 22



Partitioning an Array

Example: partition around 22



22	1	6	4	18	10	32	40	50

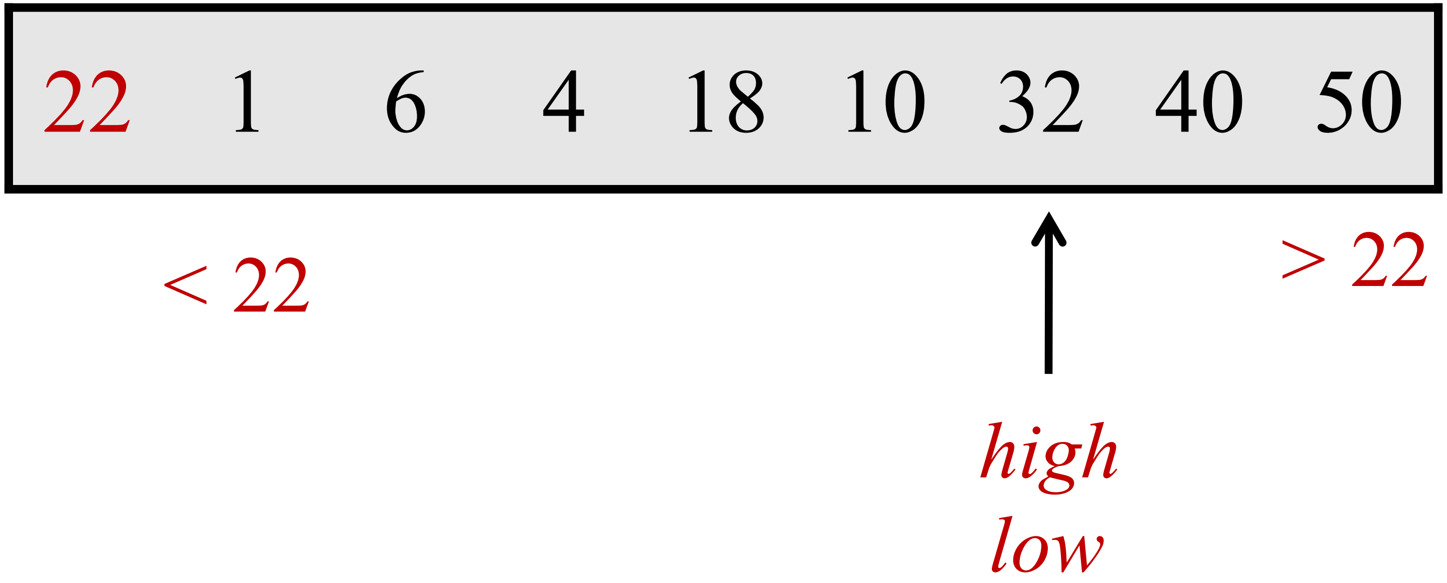
< 22

> 22

low *high*

Partitioning an Array

Example: partition around 22



22 1 6 4 18 10 32 40 50

< 22

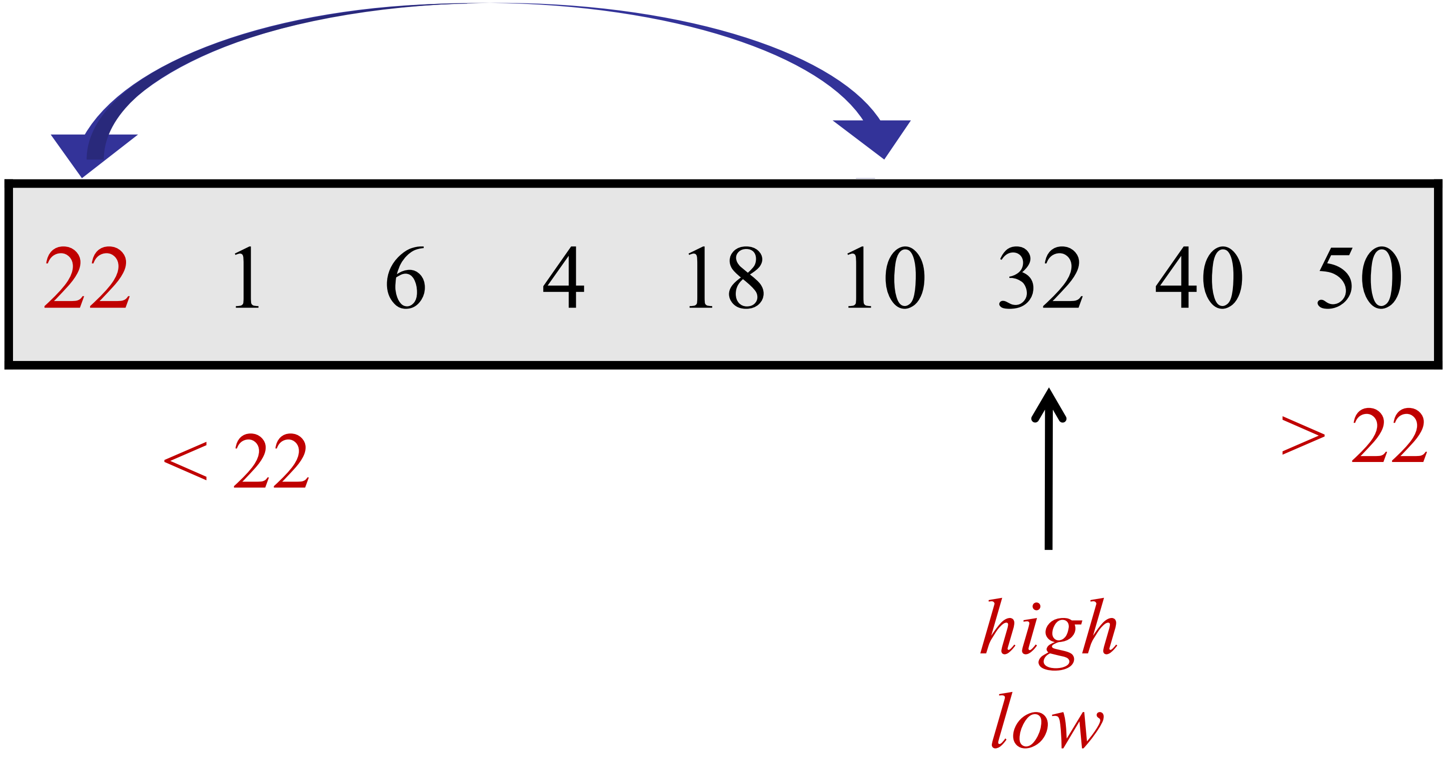
> 22

high

low

Partitioning an Array

Example: partition around 22



22	1	6	4	18	10	32	40	50

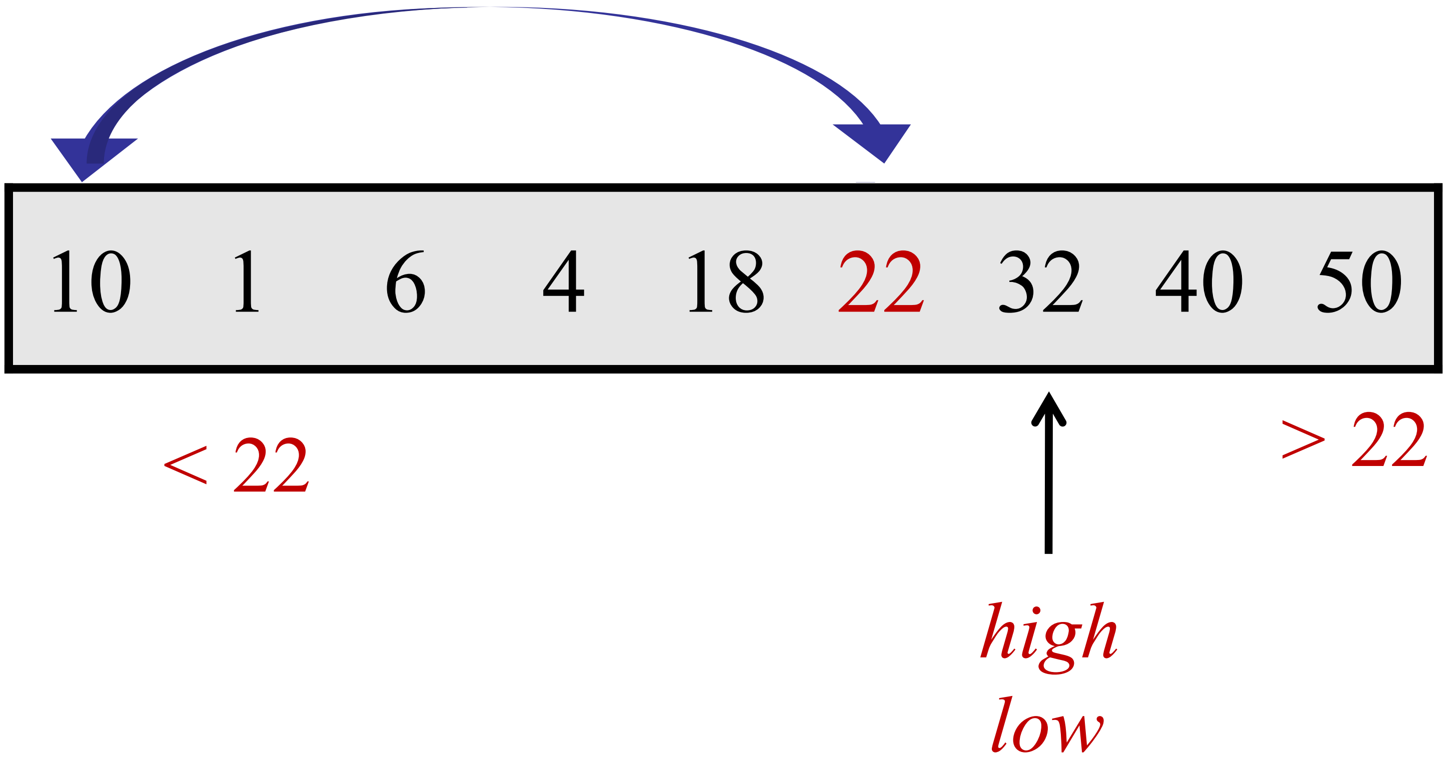
< 22

> 22

high
low

Partitioning an Array

Example: partition around 22



10 1 6 4 18 22 32 40 50

< 22

> 22

high

low

```
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: $A[n+1] = \infty$
    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;
```

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.

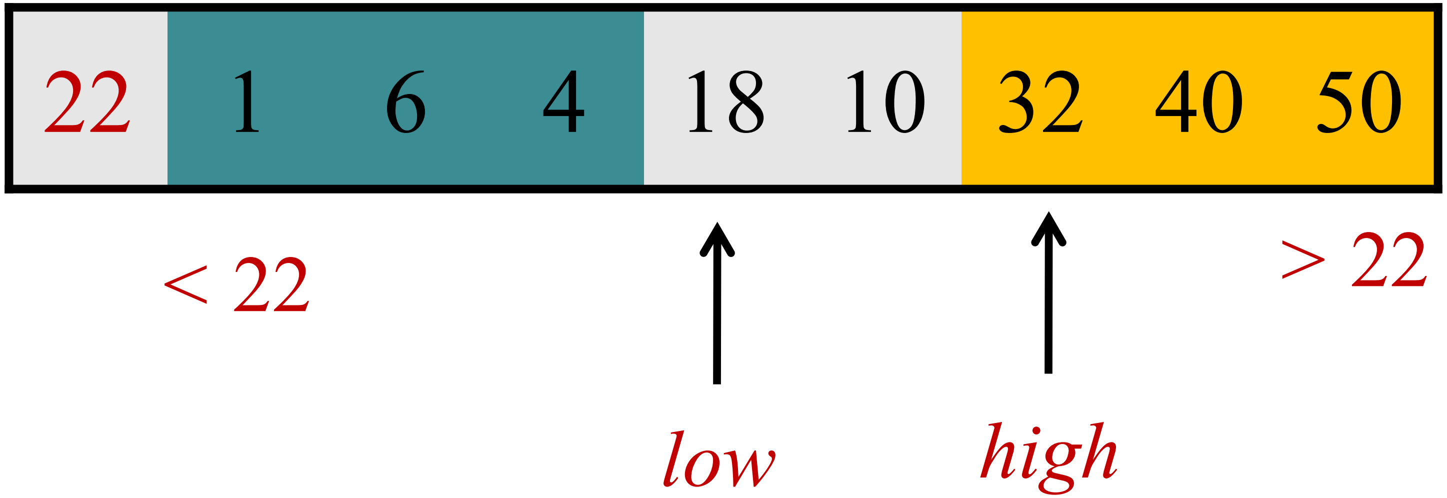
```
partition($A[1..n]$, $n$, $pIndex$)          // Assume no duplicates,  $n$>1
    $pivot = A[pIndex]$;                      // pIndex is the index of pivot
    swap($A[1]$, $A[pIndex]$);                // store pivot in $A[1]$
    $low = 2$;                                // start after pivot in $A[1]$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

Partition

Invariant: At the end of every loop iteration:

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

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



22 1 6 4 18 10 32 40 50

< 22

> 22

low

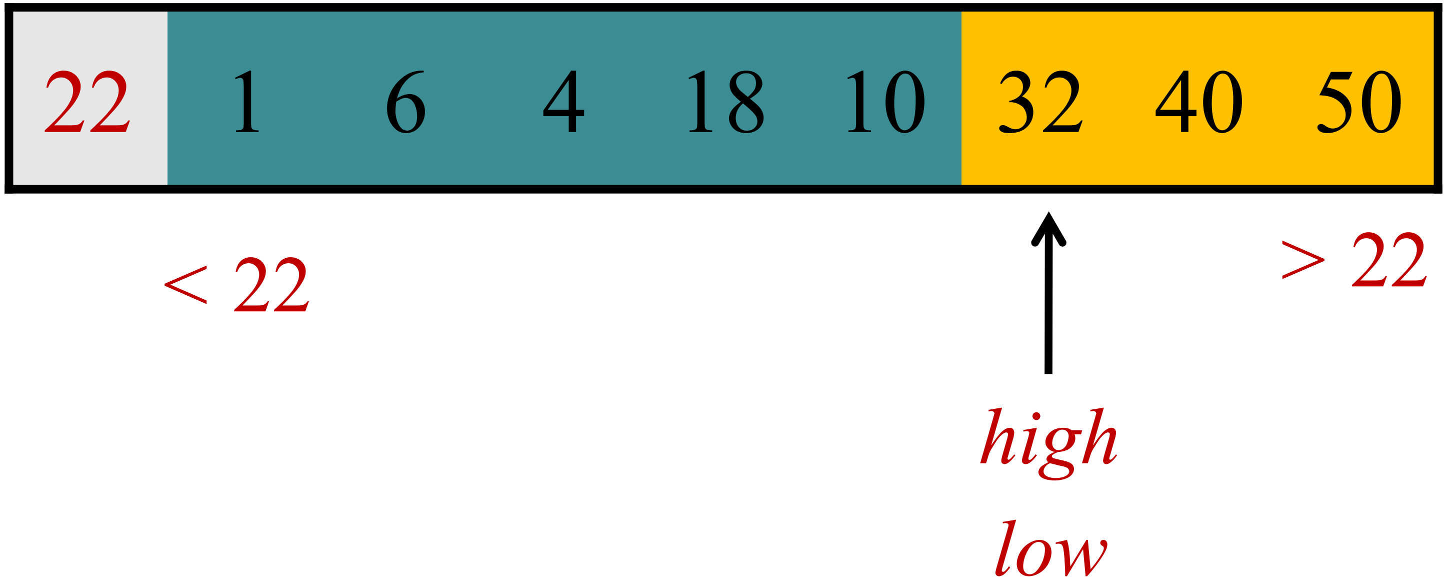
high

Partition

Invariant: At the end of every loop iteration:

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

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



22 1 6 4 18 10 32 40 50

< 22

> 22

high

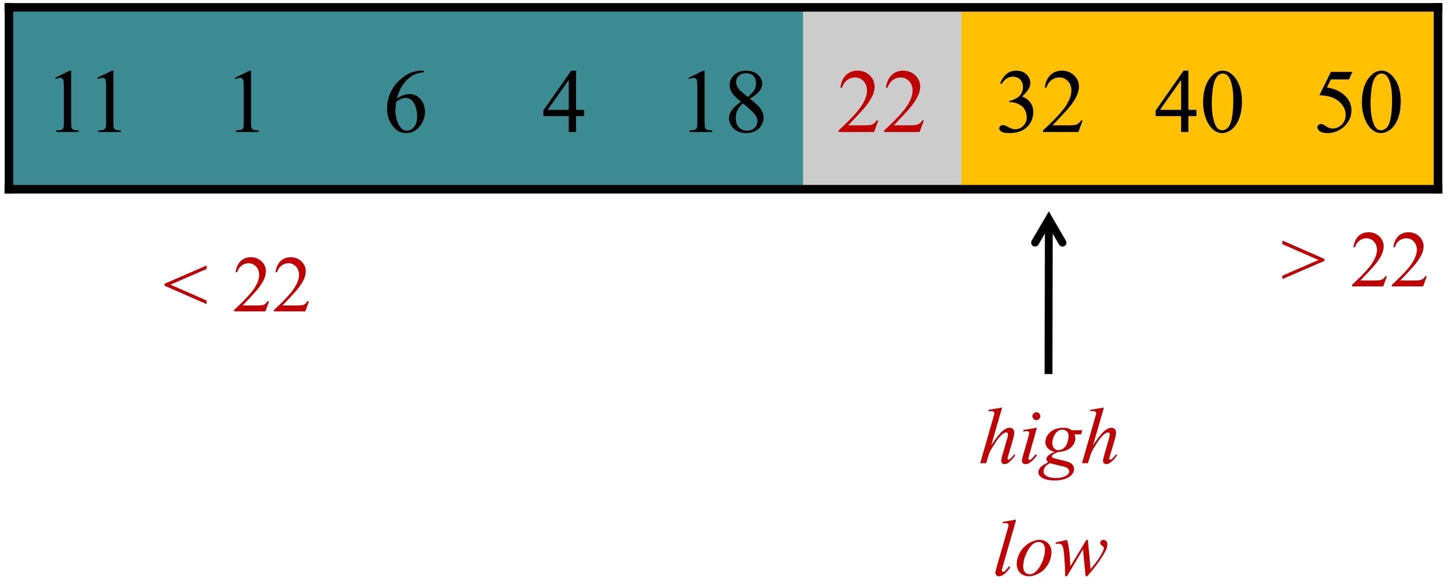
low

Partition

Claim: At the end of every loop iteration:

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

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



11	1	6	4	18	22	32	40	50

< 22

> 22

high

low

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: $A[n+1] = \infty$
    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  $low-1$;
```

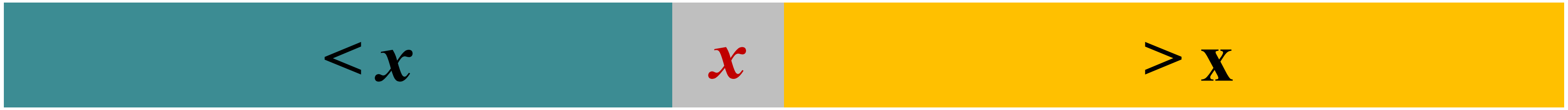
```
partition($A[1..n]$, $n$, $pIndex$)
    $pivot = A[pIndex]$;
    swap($A[1]$, $A[pIndex]$);
    $low = 2$;
    $high = n+1$;
    while ($low < high$)
        while ($A[low] < pivot$) and ($low < high$) do $low$++;
        while ($A[high] > pivot$) and ($low < high$) do $high$– – ;
        if ($low < high$) 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($A[p+1..\mathrm{n}]$, $n-p$)
```



$< x$

x

$>$ x

Sorting, continued

QuickSort

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

QuickSort

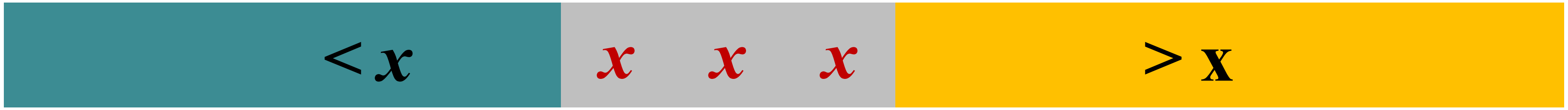
What happens if there are duplicates?

ARCHIPELAGO

is open

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

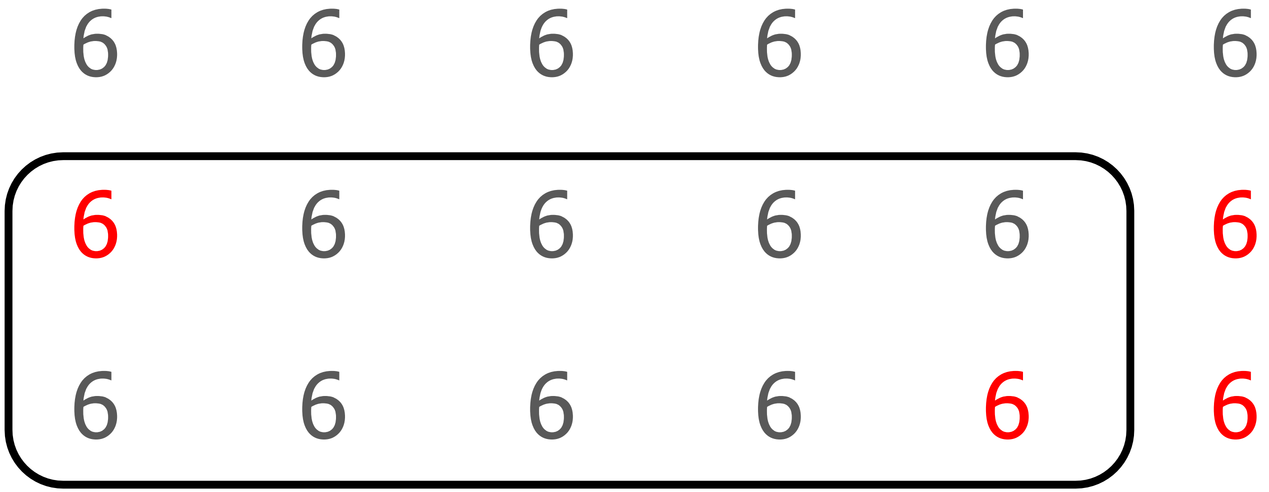
Example:

6 6 6 6 6 6

6 6 6 6 6 6

Quicksort

Example:



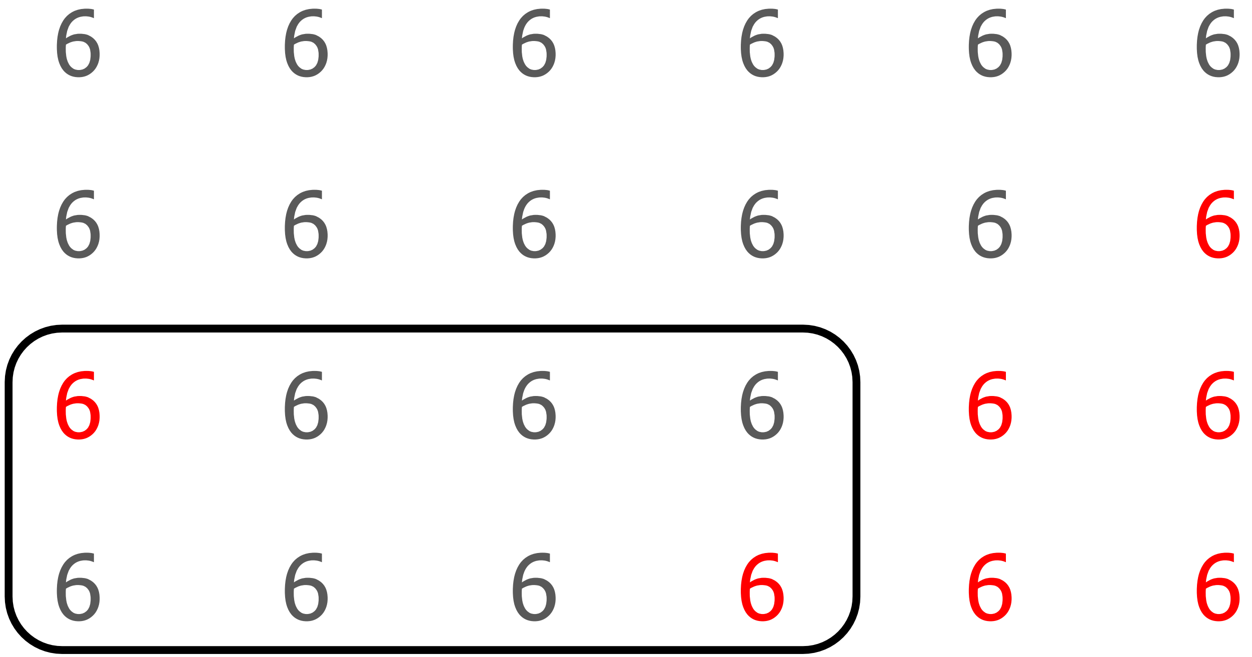
6 6 6 6 6 6

6 6 6 6 6 6

6 6 6 6 6 6

Quicksort

Example:



6 6 6 6 6 6

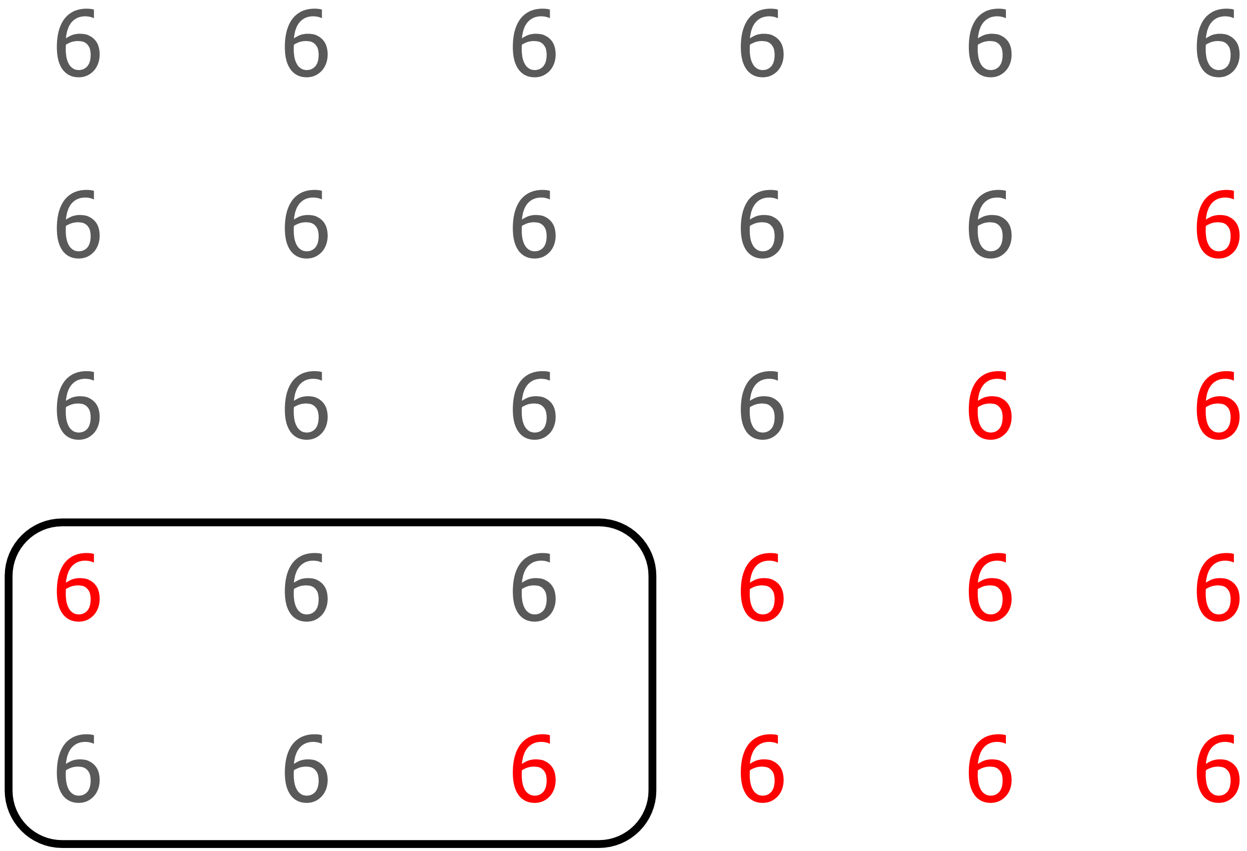
6 6 6 6 6 6

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

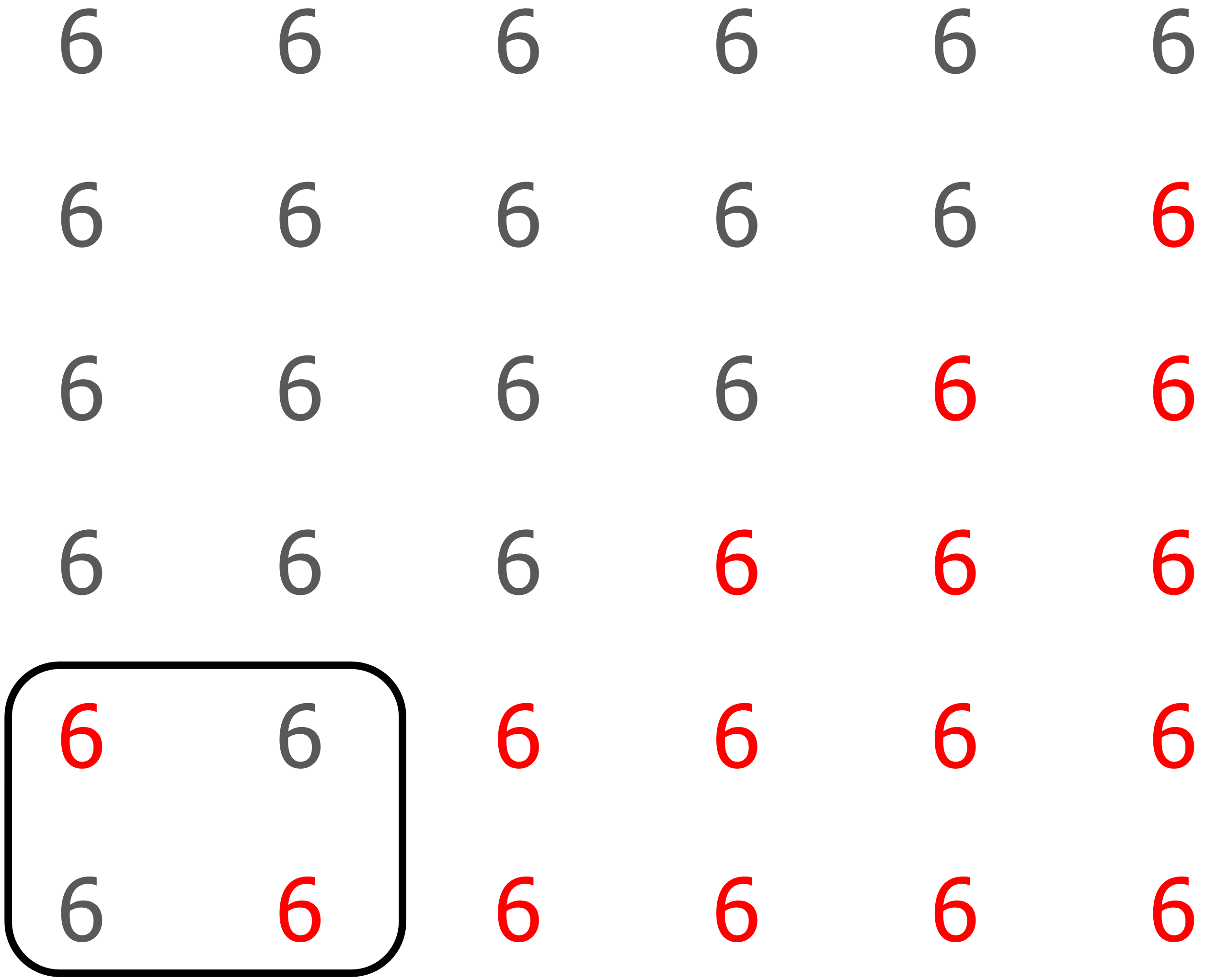
6 6 6 6 6 6

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
```

ARCHIPELAGO

is open

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: $A[n+1] = \infty$
    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  $low-1$;
```

Sorting, continued

QuickSort

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