CS2040S
Data Structures and Algorithms

Welcome!

Admin

Recorded recitation this week!

Recorded tutorial this week!

- Part 1: Review (more this week)
- Part 2: Harder questions (only one optional this week)

  - Check with your tutor on room scheduling.
  - Do prepare in advance.
  - Do have questions.
  - Do take advantage of tutorial to get to know your tutor and other students in your class

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!

operties:

mance

bility

erformance on special inputs

It ran the fastest so it must be QuickSort.

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

- 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 co... ...ared the ...peed of A an... ...d B was much f... ...it must b... insertion... ...rt.

It ran th... f... ...est so it must b... ...ickSort.

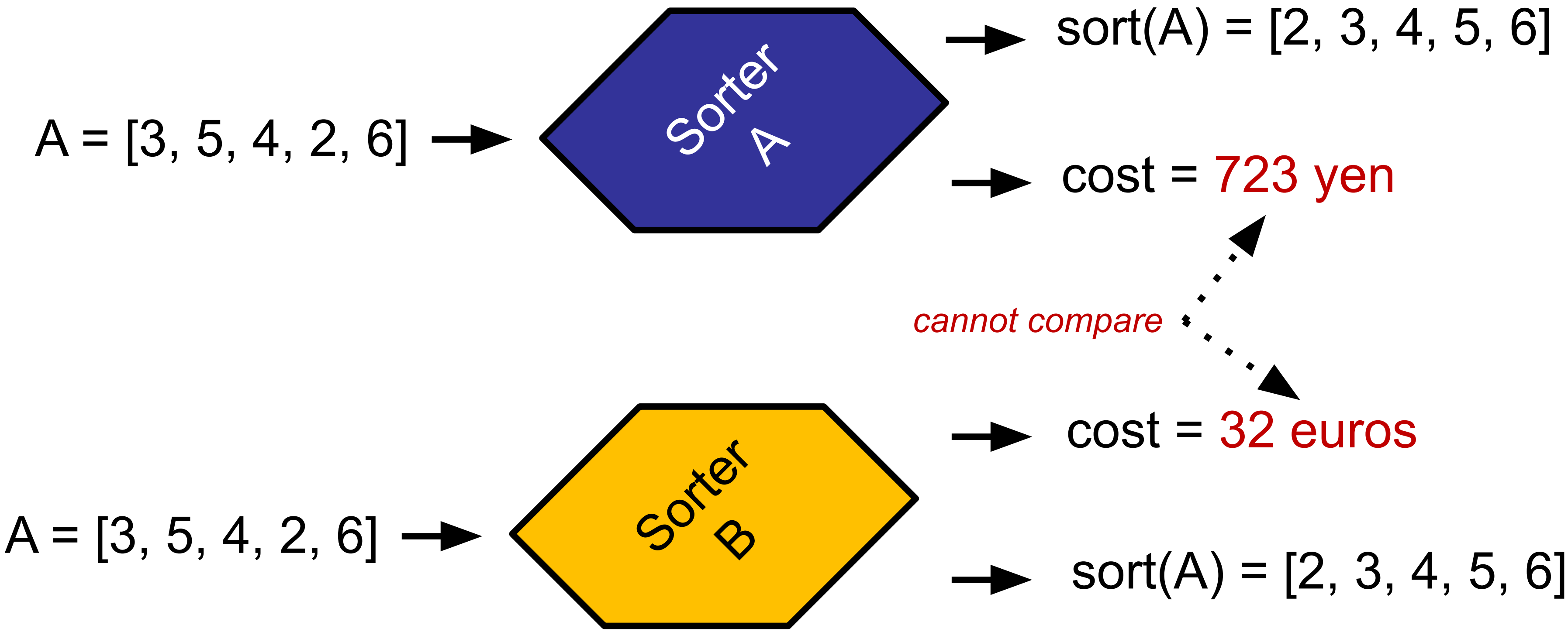
...operties:

- ...mance
- ...ability
- ...erformance on special inputs

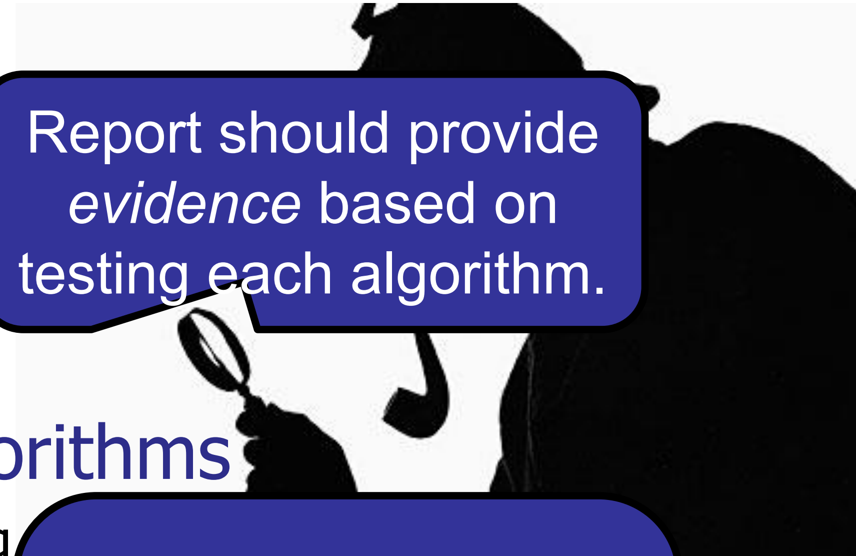
– Absolute speed is not a good reason…

Problem Set 3

Sorting Detective



Problem Set 3



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…

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 next week…

Problem Set 3

Sorting Detective

- Six suspicious sorting algorithms

Pset 3 will be delayed until next week!!

Warning: we cover QuickSort next week…

Today: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

Key questions:

How to analyze a sorting algorithm?

Invariants

Trade-offs: how to decide which algorithm to use for which problem?

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]

Sorting

```
public interface ISort{

    public void sort(int[] dataArray);

}
```

Aside: BogoSort

`BogoSort(A[1..n])`

Repeat:

a) Choose a random permutation of the array A.

b) If A is sorted, return A.

What is the expected running time of BogoSort?

Aside: BogoSort

`BogoSort(A[1..n])`

Repeat:

a) Choose a random permutation of the array A.

b) If A is sorted, return A.

What is the expected running time of BogoSort?

$\boxed{O(n \cdot n!)}$

Aside: BogoSort

`QuantumBogoSort(A[1..n])`

a) Choose a random permutation of the array A.
b) If A is sorted, return A.
c) If A is not sorted, destroy the universe.

What is the expected running time of Quantum BogoSort?

(Remember QuantumBogoSort when you learn about non-deterministic Turing Machines.)

Aside: MaybeBogoSort

MaybeBogoSort(A[1..n])

1. Choose a random permutation of the array A.
2. If A[1] is the minimum item in A then:

   MaybeBogoSort(A[2..n])

   Else

   MaybeBogoSort(A[1..n])

What is the expected running time of MaybeBogoSort?

Today: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

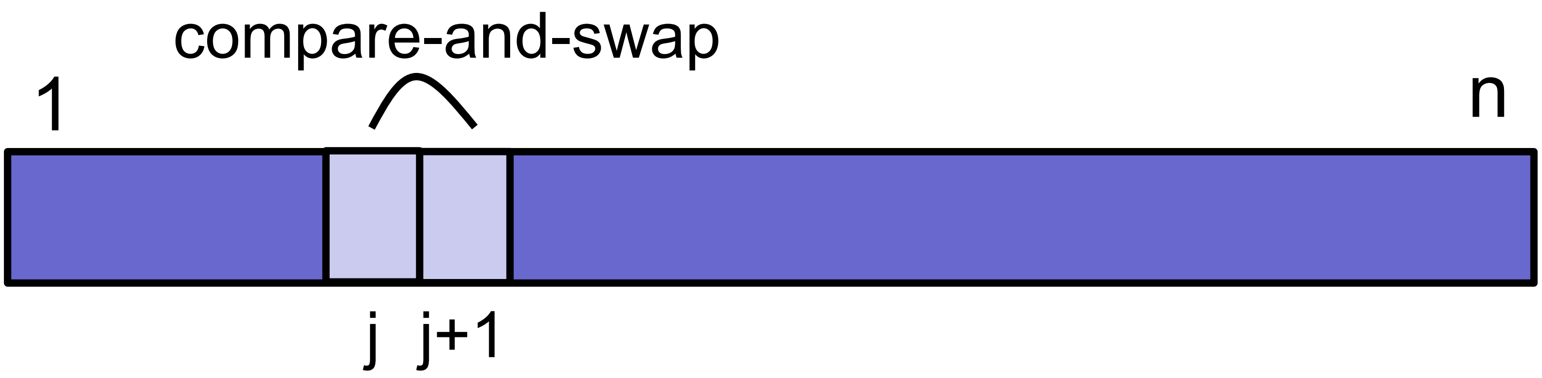
BubbleSort

BubbleSort(A, n)

repeat n **times:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])



BubbleSort

Example: 8 2 4 9 3 6

BubbleSort

Example:	**8**	**2**	4	9	3	6
	2	**8**	4	9	3	6

BubbleSort

Example:

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

BubbleSort

Example:

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

BubbleSort

Example:	8	2	4	9	3	6
	2	8	4	9	3	6
	2	4	8	9	3	6
	2	4	8	**9**	**3**	6
	2	4	8	**3**	**9**	6

BubbleSort

Example:

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

BubbleSort

Example:

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

BubbleSort

Pass 2:

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

BubbleSort

Pass 3:

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

BubbleSort

Pass 4:

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

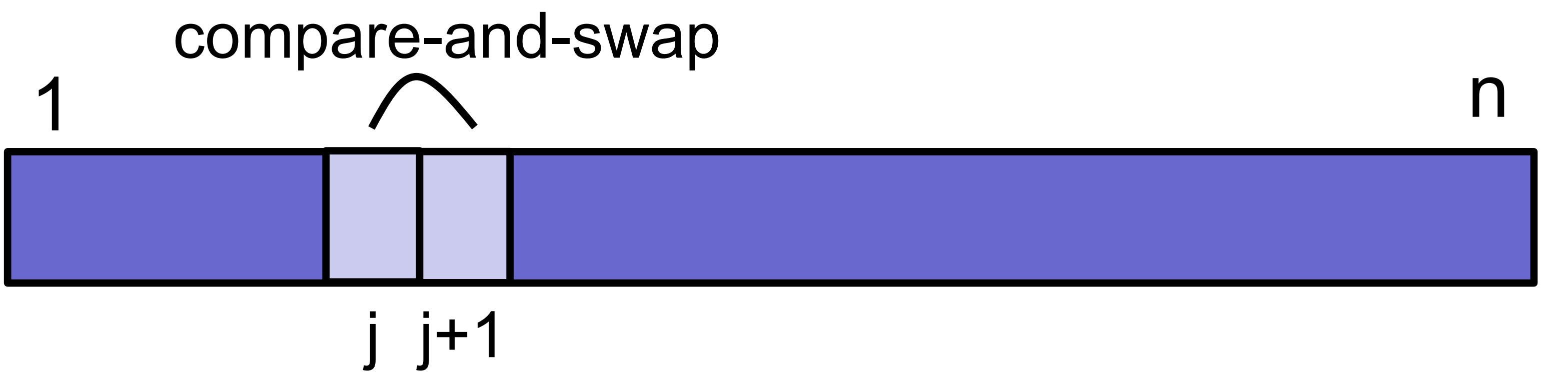
BubbleSort

BubbleSort(A, n)

repeat n **times:**

  for j ← 1 **to** n-1

    if A[j] > A[j+1] **then** swap(A[j], A[j+1])



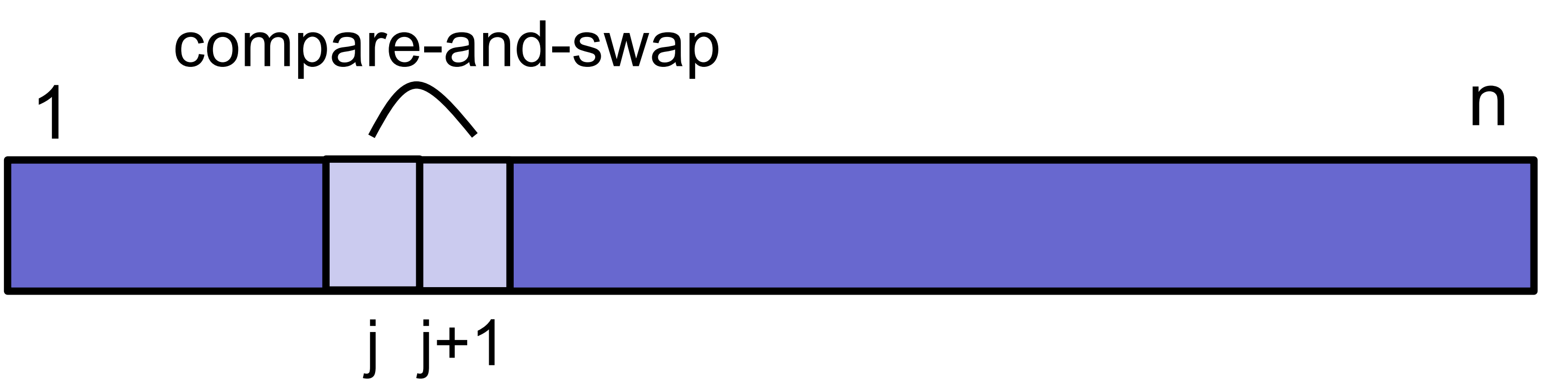
BubbleSort

BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

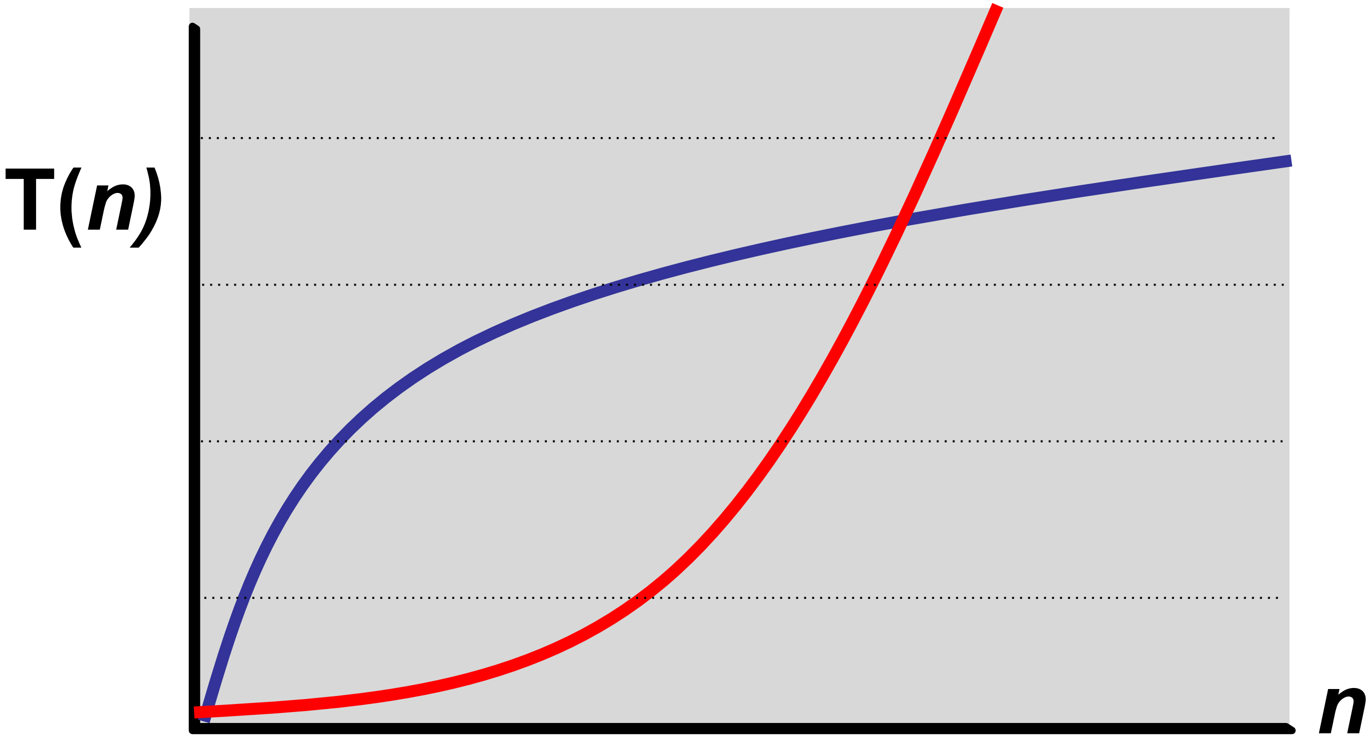
if A[j] > A[j+1] **then** swap(A[j], A[j+1])



Big-O Notation

How does an algorithm scale?

- For large inputs, what is the running time?
- T(n) = running time on inputs of size n



What is the running time of BubbleSort?

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

BubbleSort

Running time:

- Depends on the input!

BubbleSort

Example:

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

BubbleSort

Running time:

- Depends on the input!

Best-case:

- Already sorted: $O(n)$

BubbleSort

Best-case:

- Already sorted: O(n)

Average-case:

- Assume inputs are chosen at random.

Worst-case:

- Max running time over all possible inputs.

BubbleSort

Best-case:

- Already sorted: O(n)

Average-case:

- Assume inputs are chosen at random.

Worst-case:

Unless otherwise specified, in CS2040S, we focus on worst-case

- Max running time over all possible inputs.

BubbleSort Analysis

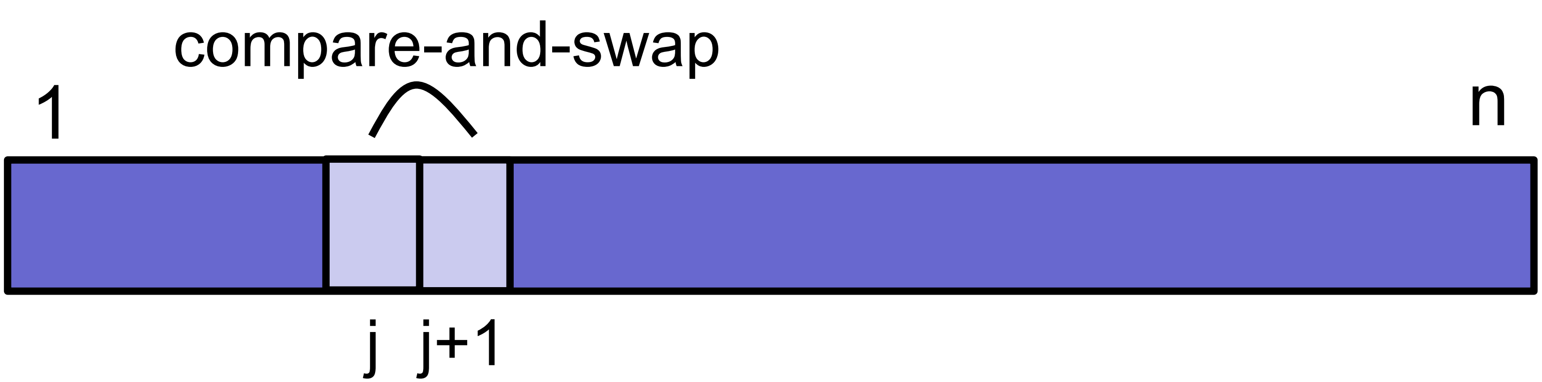
How many iterations do we need?

BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])



BubbleSort Analysis

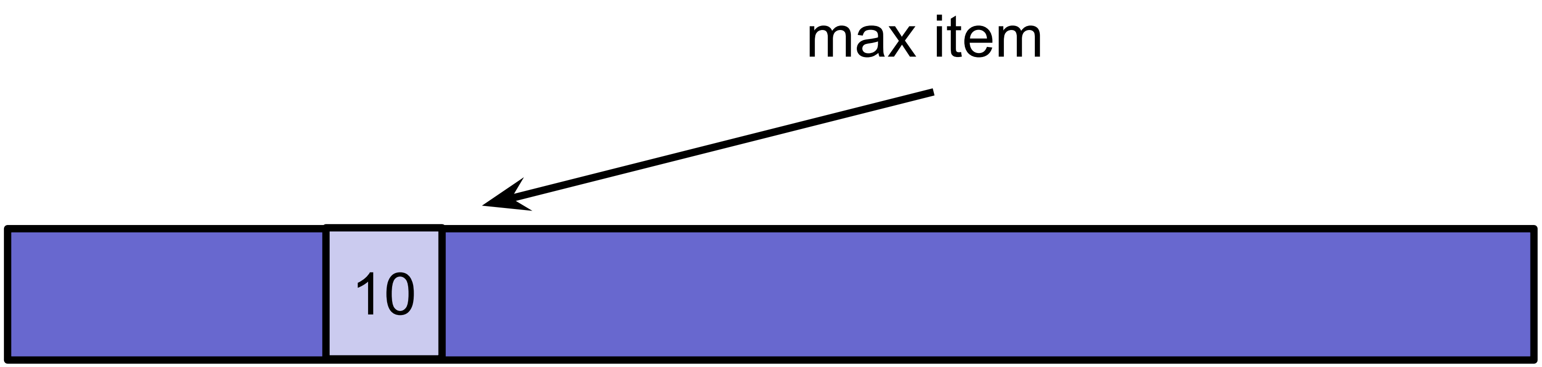
BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])

What is a good loop invariant for BubbleSort?



BubbleSort Analysis

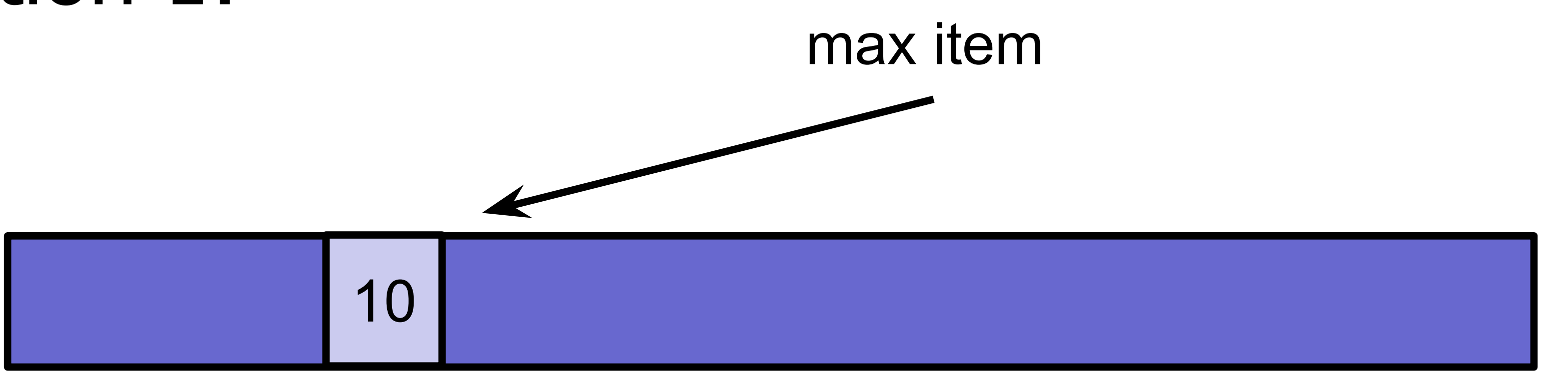
BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])

Iteration 1:



BubbleSort Analysis

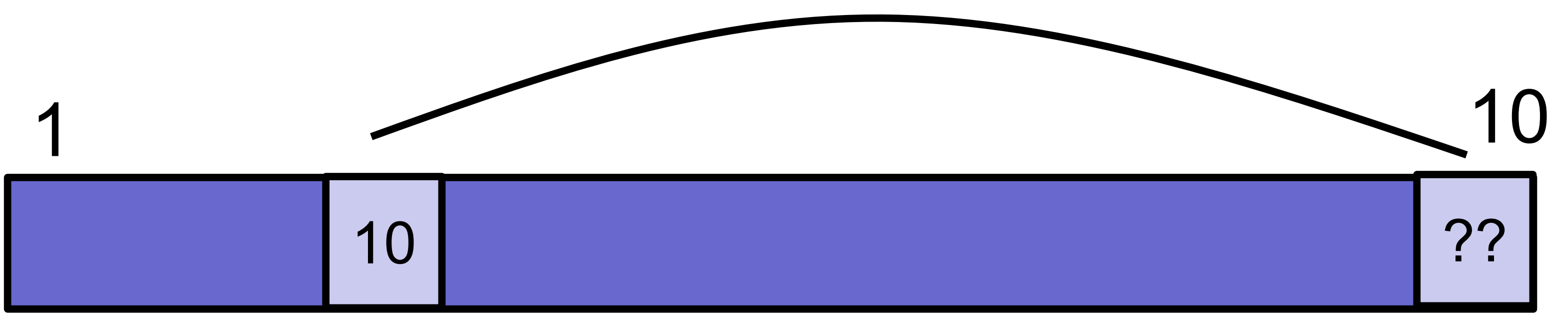
BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])

Iteration 1:



BubbleSort Analysis

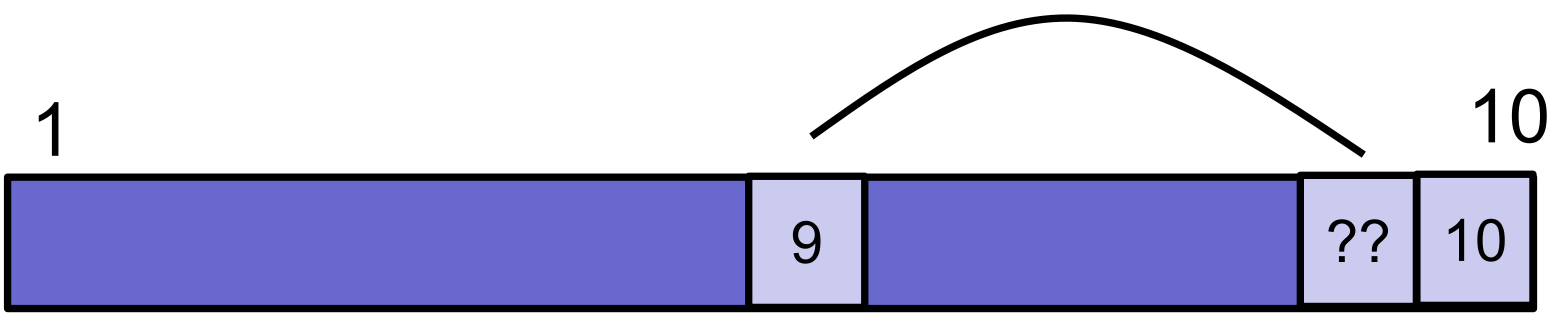
BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

if A[j] > A[j+1] **then** swap(A[j], A[j+1])

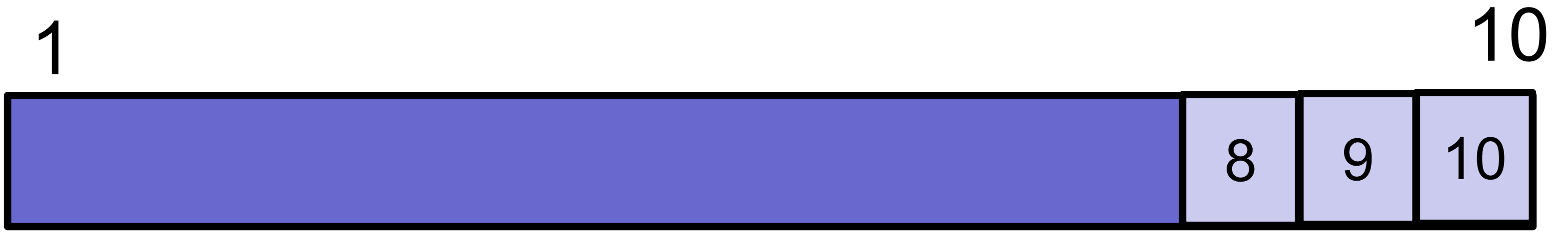
Iteration 2:



BubbleSort Analysis

Loop invariant:

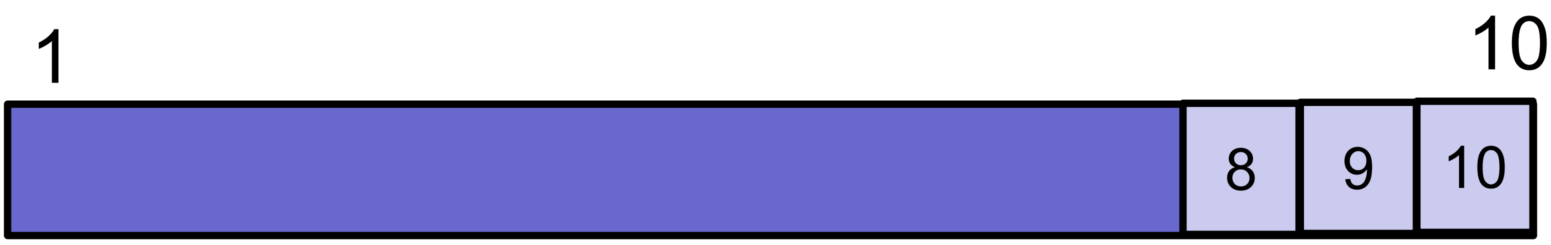
At the end of iteration j: ???



BubbleSort Analysis

Loop invariant:

At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array.

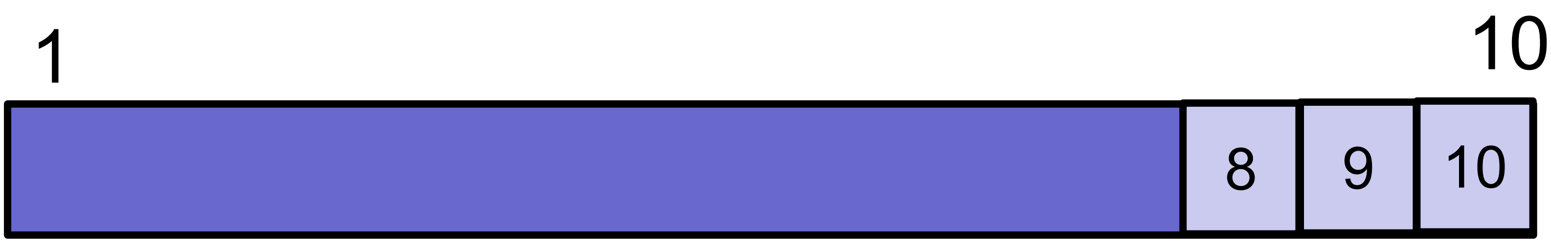


BubbleSort Analysis

Loop invariant:

At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array.

Correctness: after n iterations → sorted

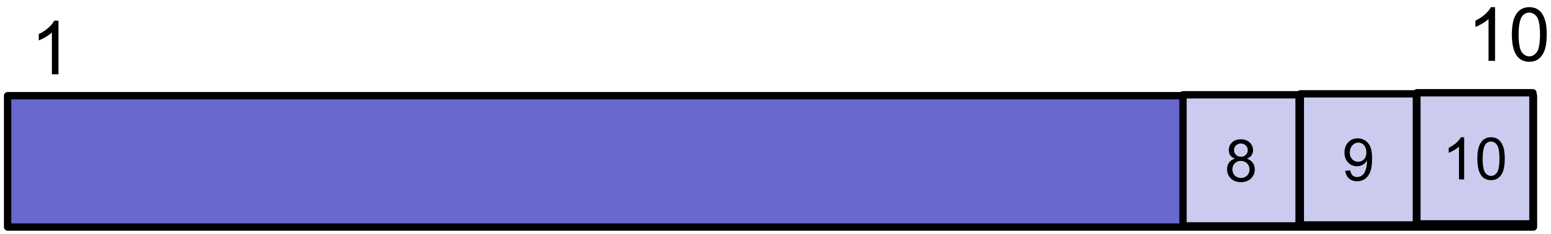


BubbleSort Analysis

Loop invariant:

At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array.

Worst case: n iterations

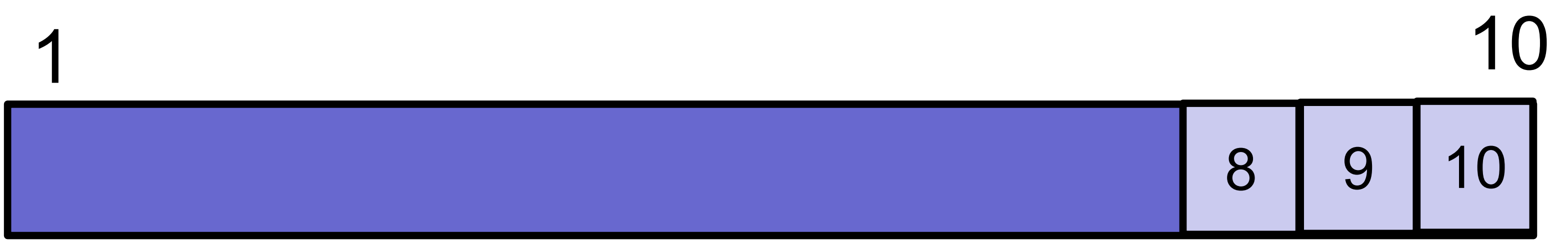


BubbleSort Analysis

Loop invariant:

At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array.

Worst case: n iterations → $O(n^2)$ time



BubbleSort

Best-case: $O(n)$

- Already sorted

Average-case: $O(n^2)$

- Assume inputs are chosen at random…

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

- Bound on how long it takes.

Today: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

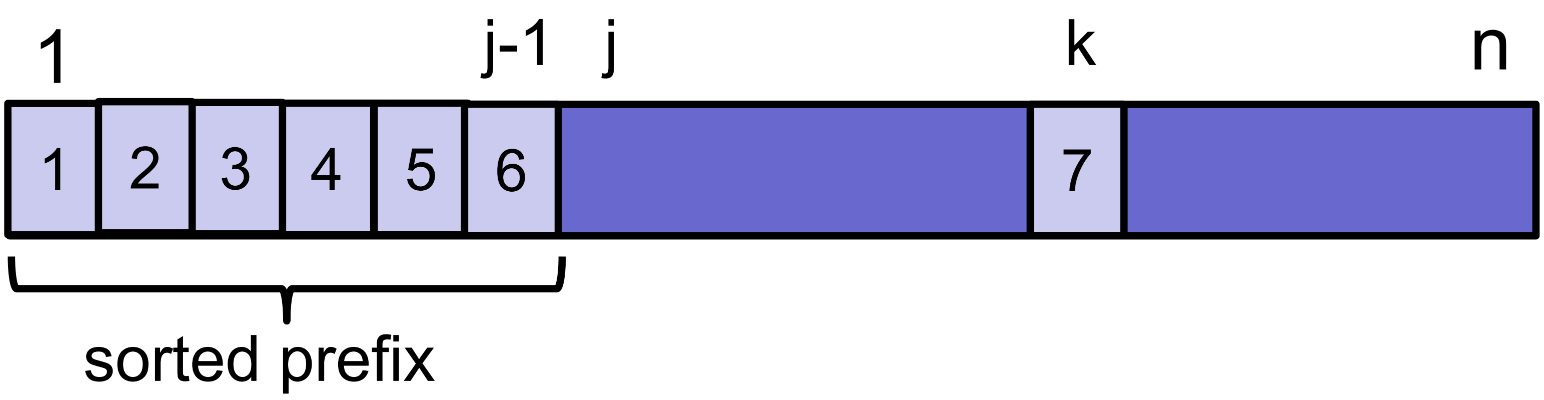
SelectionSort

SelectionSort(A, n)

for j ← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])



SelectionSort

Example: 8 2 4 9 3 6

SelectionSort

Example: 8 **2** 4 9 3 6

SelectionSort

Example:	8	**2**	4	9	3	6
	2	8	4	9	3	6

SelectionSort

Example:	8	**2**	4	9	3	6
	2	8	4	9	**3**	6

SelectionSort

Example:

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

SelectionSort

Example:					
8	**2**	4	9	3	6
2	8	4	9	**3**	6
2	3	**4**	9	8	6

SelectionSort

Example:

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

SelectionSort

Example:

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

SelectionSort

Example:

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

What is the (worst-case) running time of SelectionSort?

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

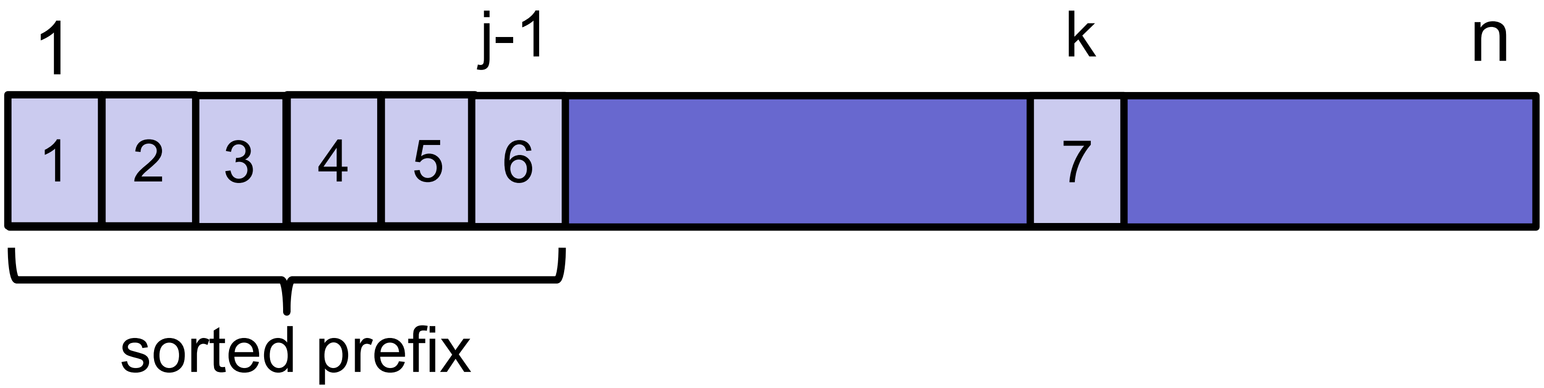
SelectionSort

SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])



SelectionSort

Time: $(n - j)$

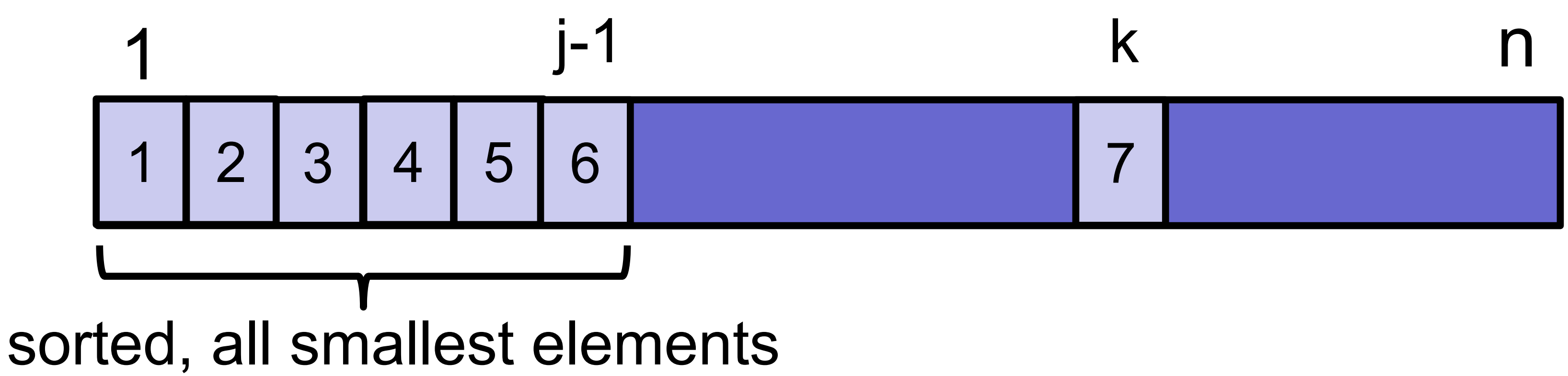
SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])

Running time: $n + (n-1) + (n-2) + (n-3) + \ldots$



SelectionSort

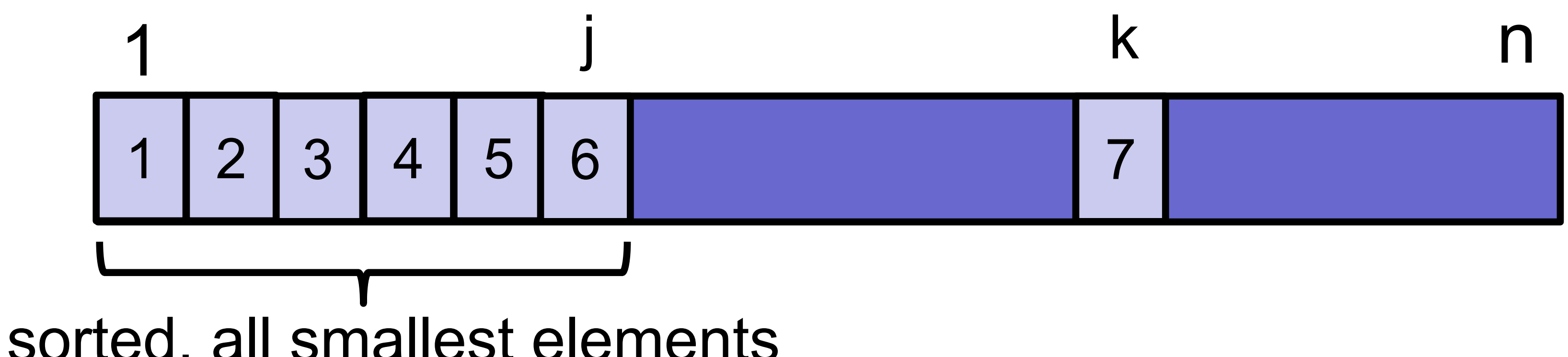
Time: $(n - j)$

SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])



Basic facts

n

first iteration

Basic facts

$n + (n - 1)$

second iteration

Basic facts

$$n + (n - 1) + (n - 2)$$

third iteration

Basic facts

$$n + (n - 1) + (n - 2) + (n - 3) + \ldots + 1 \quad =$$

Basic facts

$$n + (n - 1) + (n - 2) + (n - 3) + \ldots + 1 \quad = (n)(n+1)/2$$

$$=$$

Basic facts

$$n + (n-1) + (n-2) + (n-3) + \ldots + 1 \quad = (n)(n+1)/2$$

$$= \Theta(n^2)$$

SelectionSort

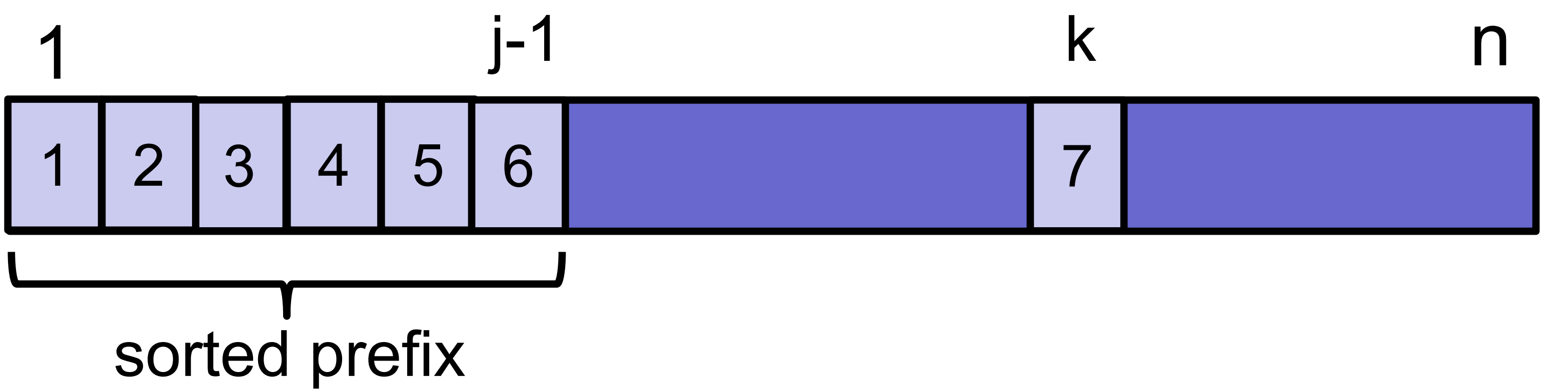
SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])

Running time: $O(n^2)$



What is the BEST CASE running time of SelectionSort?

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

SelectionSort

SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])

Running time: $O(n^2)$ (always; in the worst-case) and $\Omega(n^2)$ (even in the best case)

SelectionSort

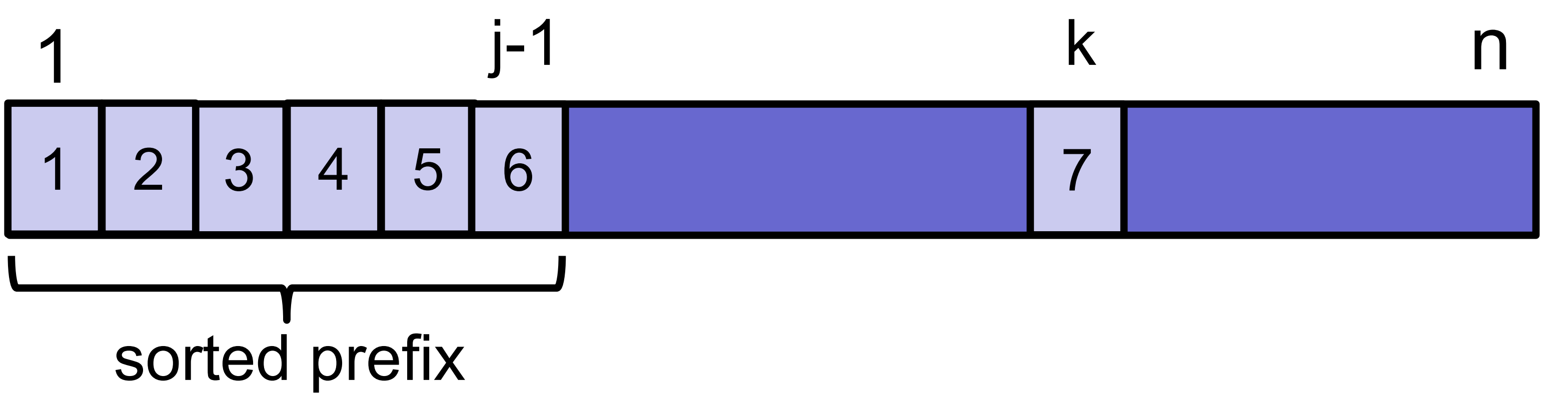
SelectionSort(A, n)

for j← 1 **to** n-1**:**

find minimum element A[j] in A[j..n]

swap(A[j], A[k])

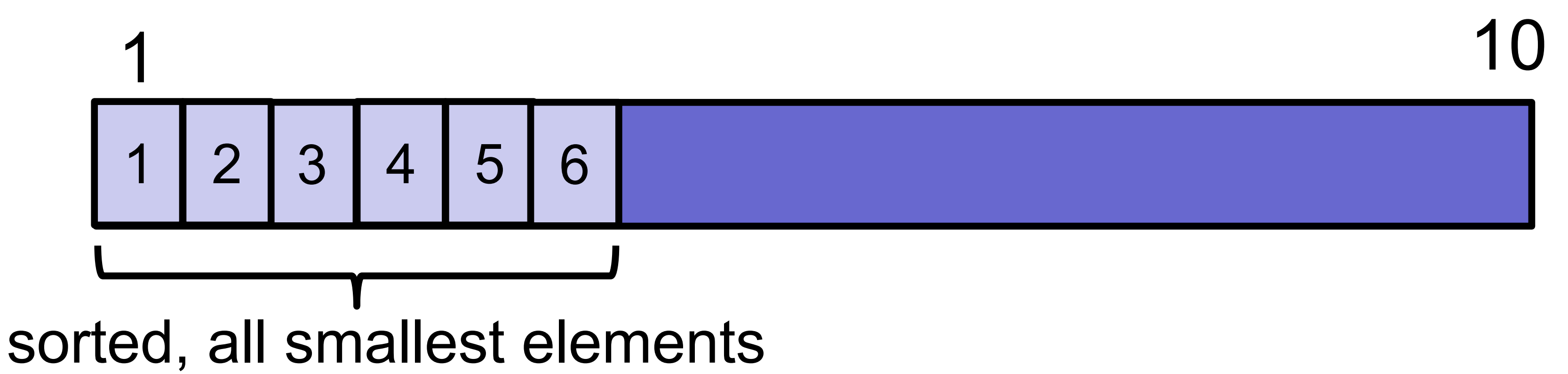
What is a good loop invariant for SelectionSort?



SelectionSort Analysis

Loop invariant:

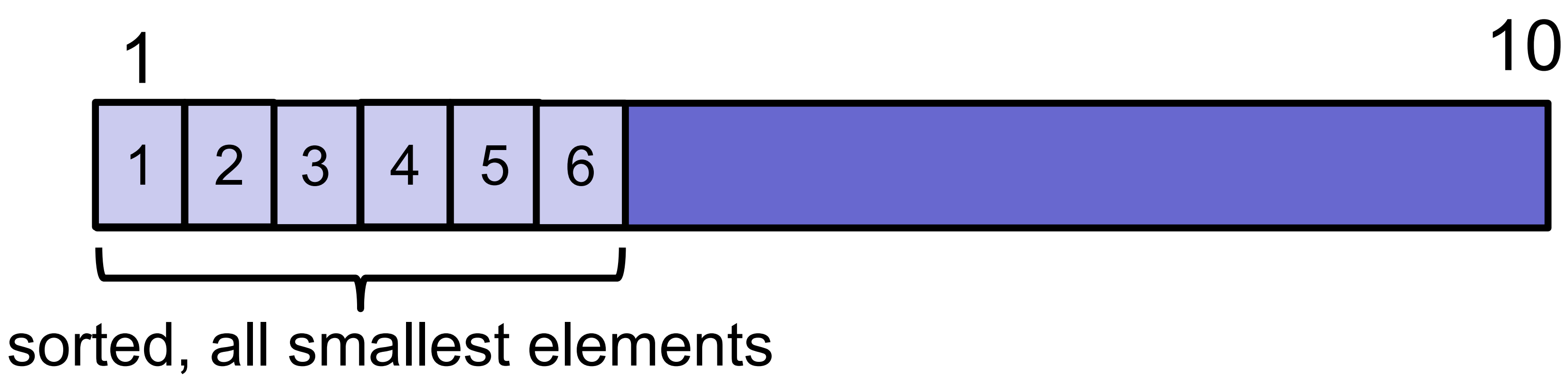
At the end of iteration j: the smallest j items are correctly sorted in the first j positions of the array.



SelectionSort Analysis

Loop invariant: (Alternative)

At the **end** of iteration j, for all i <= j,
A[i] is the ith smallest element of the entire array.



Today: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

Insertion Sort

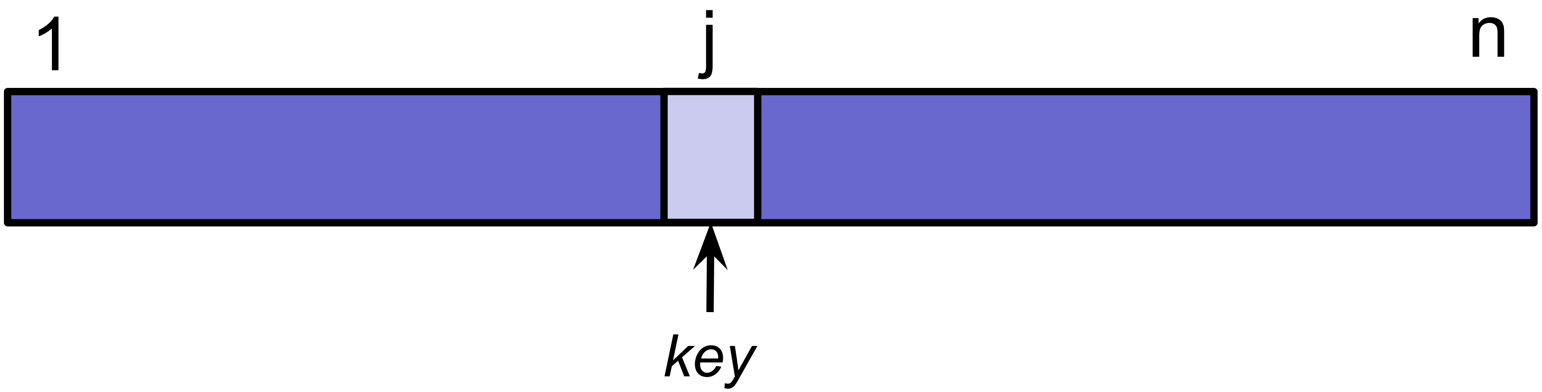
InsertionSort(A, n)

for j ← 2 **to** n

key ← A[j]

Insert key into the sorted array A[1..j-1]

Illustration: At iteration j



Insertion Sort

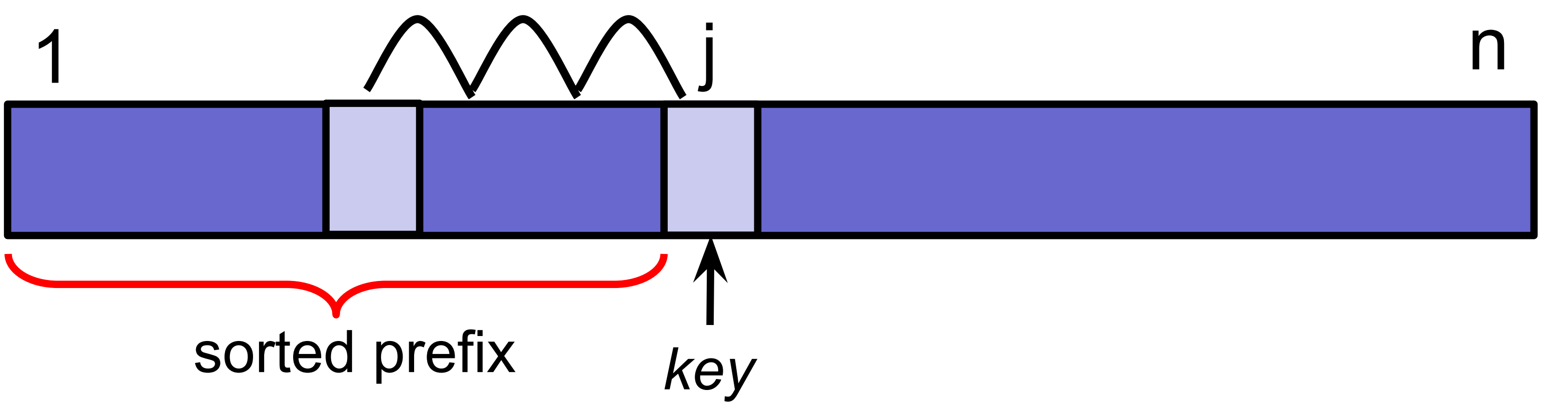
InsertionSort(A, n)

for j ← 2 **to** n

key ← A[j]

Insert key into the sorted array A[1..j-1]

Illustration: At iteration j



Insertion Sort

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

Insertion Sort

Example: **8 2 4 9 3 6**

swap

Insertion Sort

Example: 8 2 | 4 9 3 6

2 8 | 4 9 3 6

swap

Insertion Sort

Example:

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

swap

Insertion Sort

Example:

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

swap

Insertion Sort

Example:

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

Insertion Sort

Example:

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

What is the (worst-case) running time of InsertionSort?

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

Insertion Sort

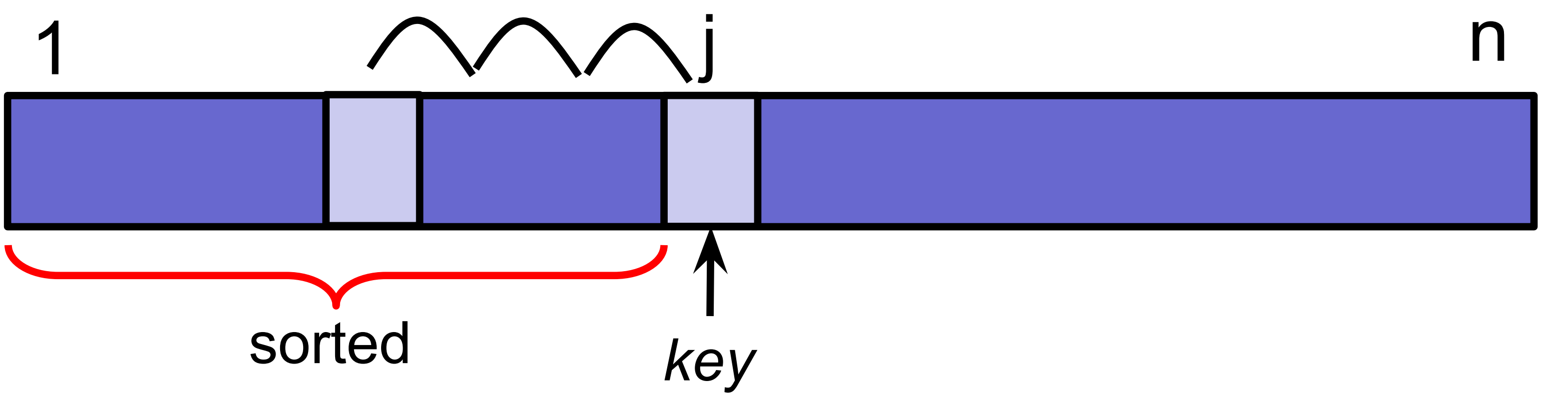
We need to analyse this step:

Insertion-Sort(A, n)

for j ← 2 **to** n

key ← A[j]

Insert key into the sorted array A[1..j-1]



Insertion Sort

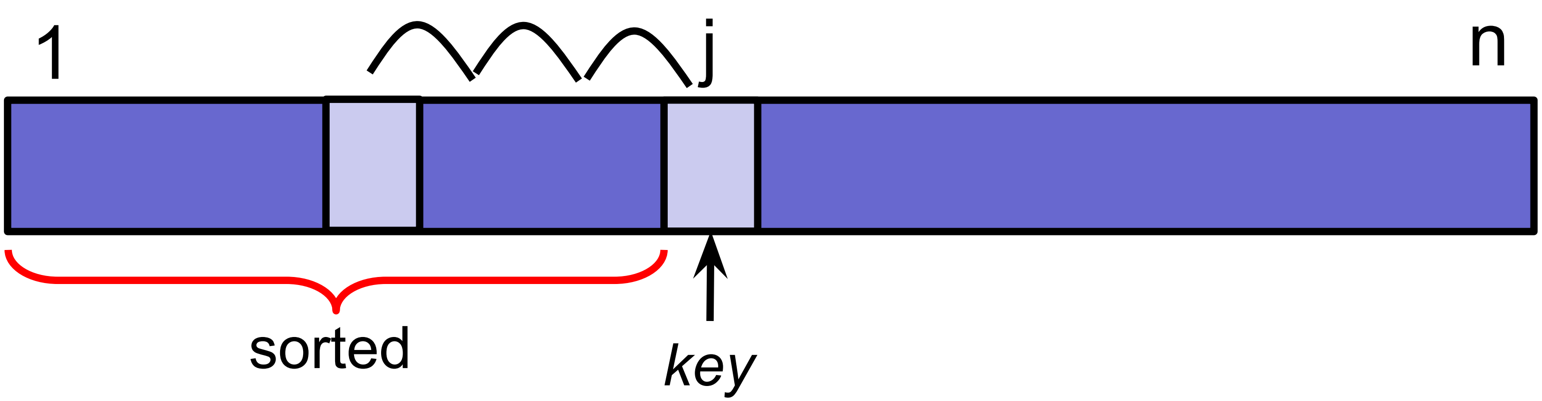
What is the max distance that the key needs to be moved?

Insertion-Sort(A, n)

    for j ← 2 **to** n

        key ← A[j]

        Insert key into the sorted array A[1..j-1]



Insertion Sort Analysis

Insertion-Sort(A, n)

    for j ← 2 **to** n

        key ← A[j]

        i ← j–1

        while (i > 0) **and** (A[i] > *key*)

            A[i+1] ← A[i]

            i ← i–1

        A[i+1] ← *key*

Repeat at most j times.

Basic facts

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) + n \quad = (n)(n+1)/2$$

$$= \Theta(n^2)$$

Insertion Sort

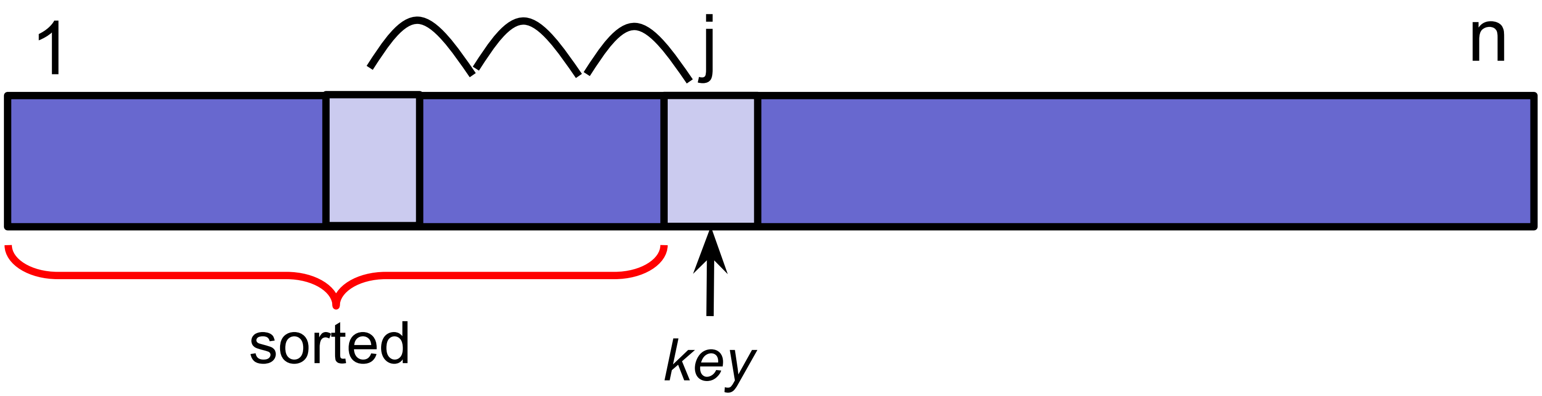
Insertion-Sort(A, n)

for j ← 2 **to** n

key ← A[j]

Insert key into the sorted array A[1..j-1]

Running time: $O(n^2)$



Insertion Sort

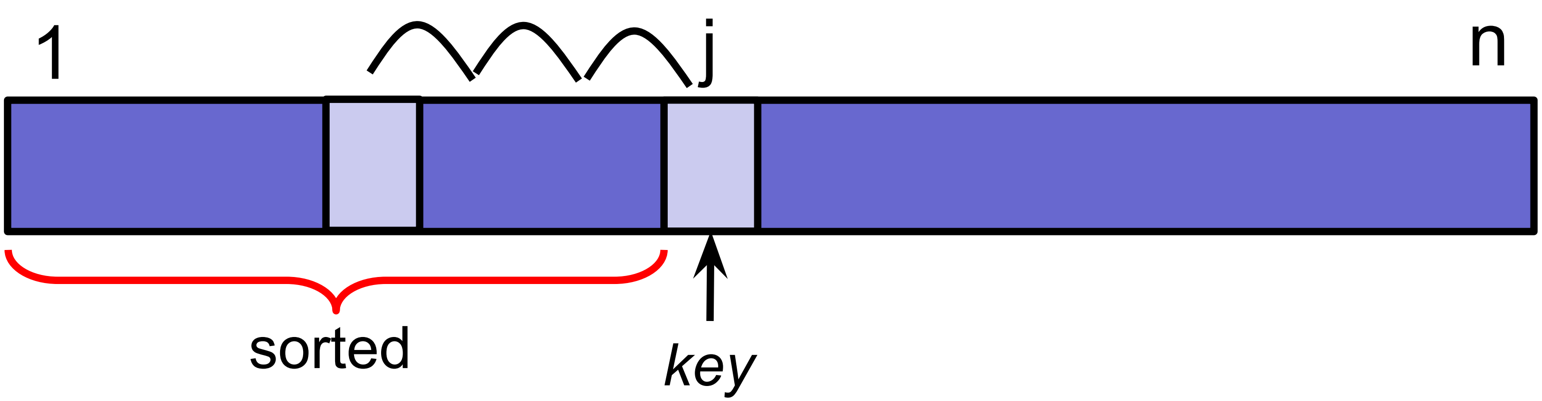
Insertion-Sort(A, n)

for j ← 2 **to** n

key ← A[j]

Insert key into the sorted array A[1..j-1]

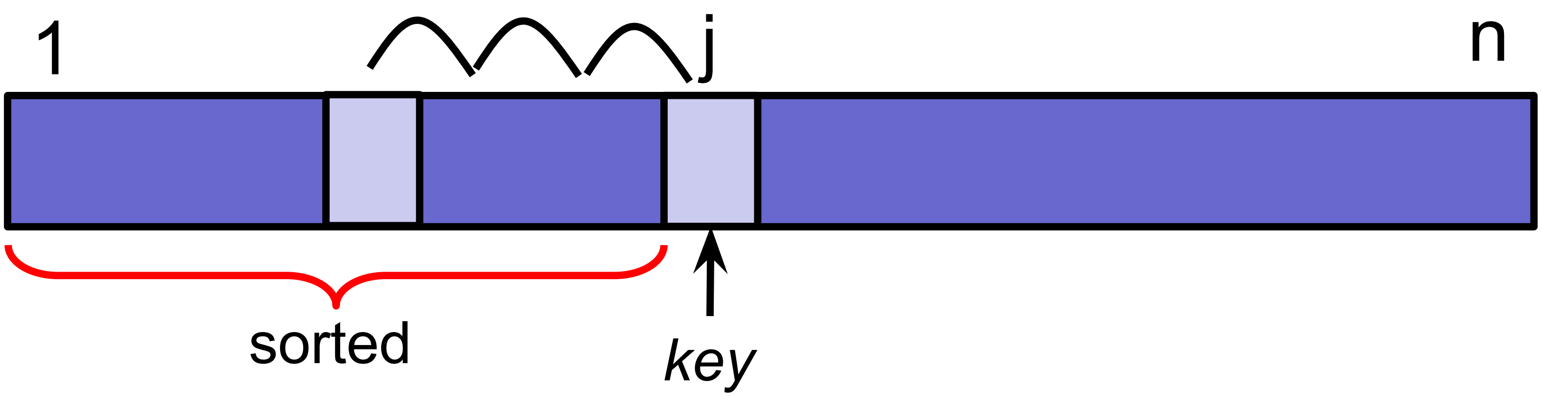
What is a good loop invariant for InsertionSort?



Insertion Sort

Loop invariant:

At the end of iteration j: the first j items in the array are in sorted order.



Insertion Sort

Best-case:

Average-case:

– Random permutation

Worst-case:

Insertion Sort

Best-case:

- Already sorted: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Average-case:

- Random permutation?

Worst-case:

- Inverse sorted: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Insertion Sort

Very fast!

Best-case: $O(n)$

- Already sorted: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Average-case:

- Random permutation?

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

- Inverse sorted: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Insertion Sort Analysis

On average, a key in position j needs to move j/2 slots backward (in expectation).

Average-case analysis:

- Assume all inputs equally likely

$$\sum_{j=2}^{n} \Theta\left({}^{j}\!/_{2} \right) = \Theta\left(n^2\right)$$

- In expectation, still $\Theta(\mathbf{n^2})$

Today: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

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!

Today: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

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]

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

How much space does a sorting algorithm need?

Space complexity

- Worst case: O(n)

Properties of Sorting Algorithms

Space complexity

- Worst case: O(n)

- An 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.

Clarification:

There are 2 options here,
they are slightly different.

Subtle issue:

How do you count space?

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

Clarification:

In CS2040S, we will use the second option

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

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

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
Value	a	b	**C**	g	h	**D**	j	k	l	m

⇩ STABLE

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

Which are stable?

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

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

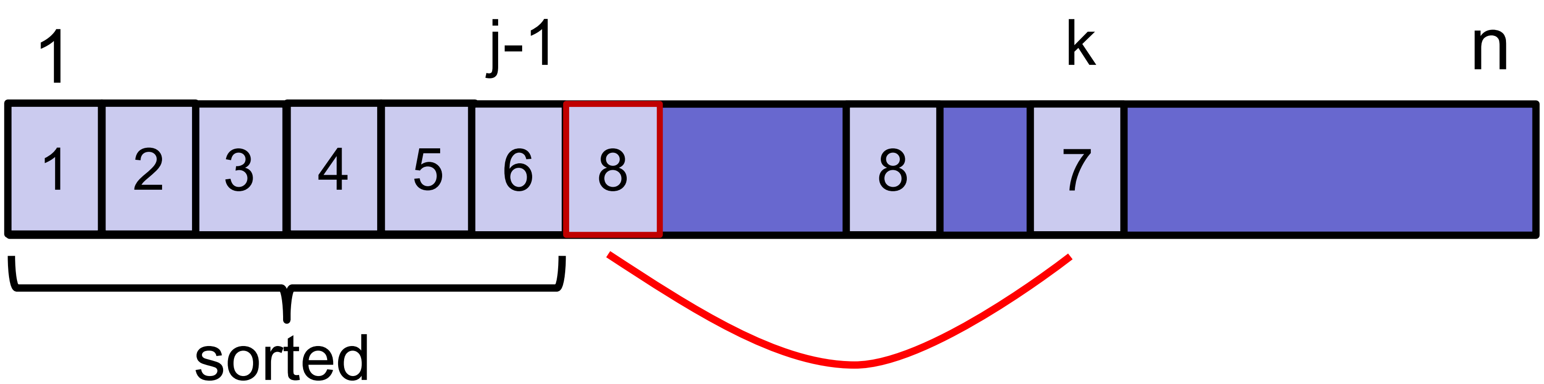
SelectionSort(A, n)

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

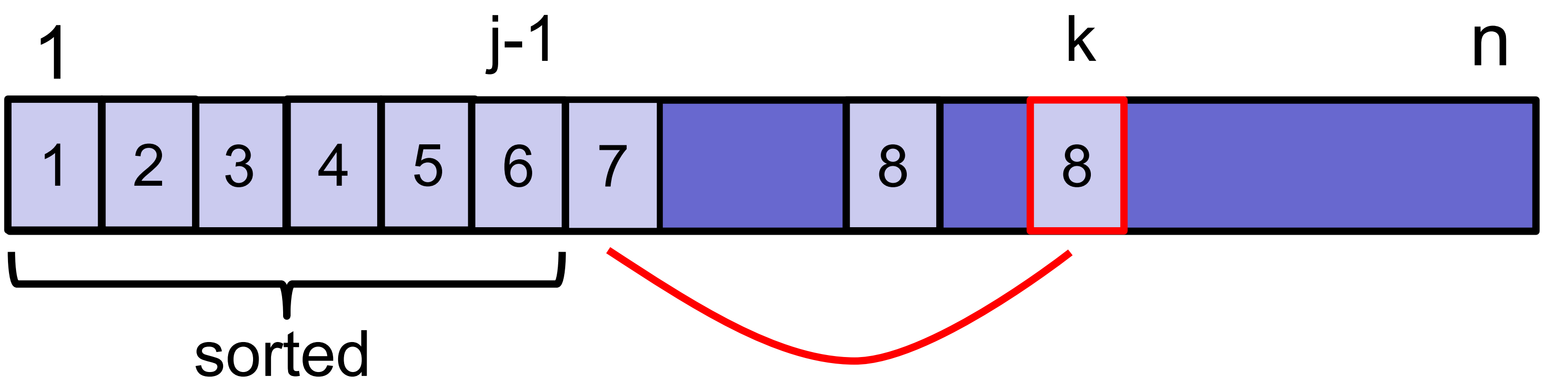
SelectionSort(A, n)

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

Today: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- 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.

<u>Advice:</u>

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;**

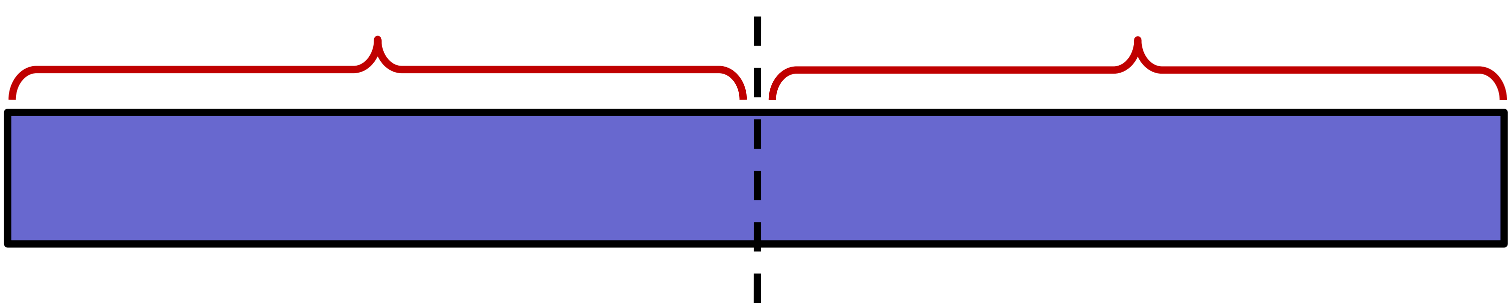
else:

X ←MergeSort**(**A[1..n/2], n/2**);**

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

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

return



MergeSort

Step 2:
Recursively sort the two halves.

MergeSort(A, n)

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

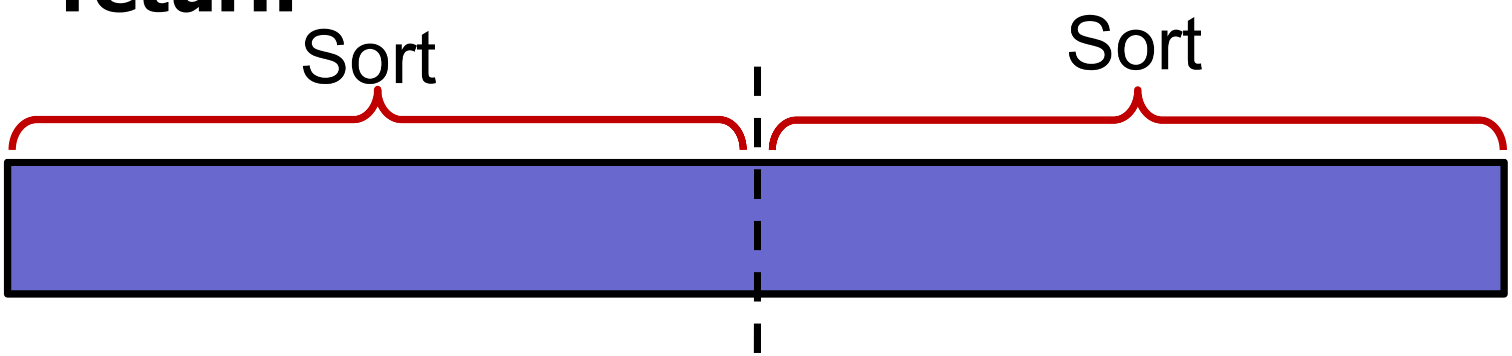
else:

X ←MergeSort(A[1..n/2], n/2);

Y ←MergeSort(A[n/2+1, n], n/2);

Merge (X,Y, n/2);

return



MergeSort

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

MergeSort(A, n)

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

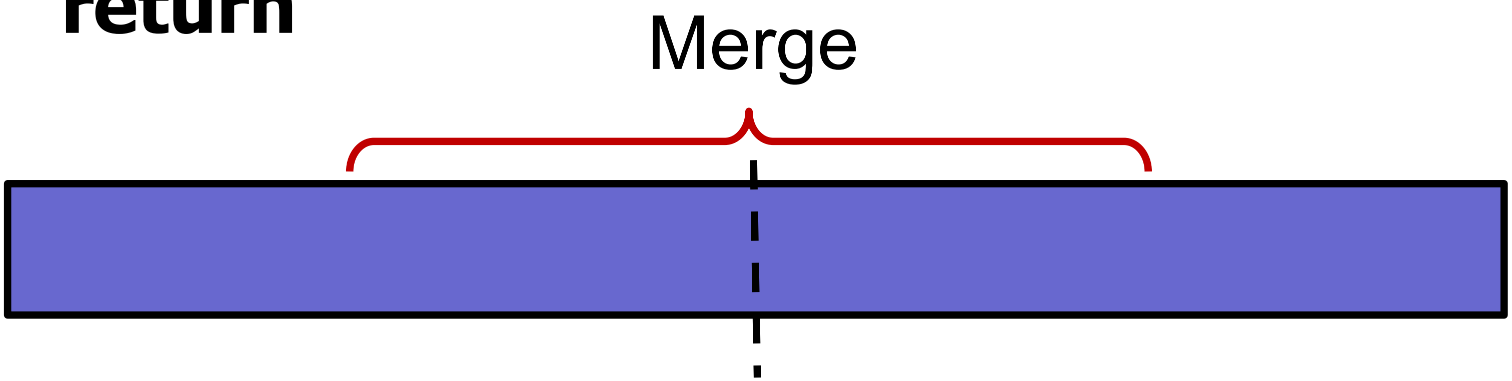
else:

X ←MergeSort**(**A[1..n/2], n/2**);**

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

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

return



MergeSort

MergeSort(A, n)

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

else:

X ←MergeSort**(**A[1..n/2], n/2**);**

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

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

return

Base case

Recursive “conquer” step

Combine solutions

The only “interesting” part is merging!

MergeSort

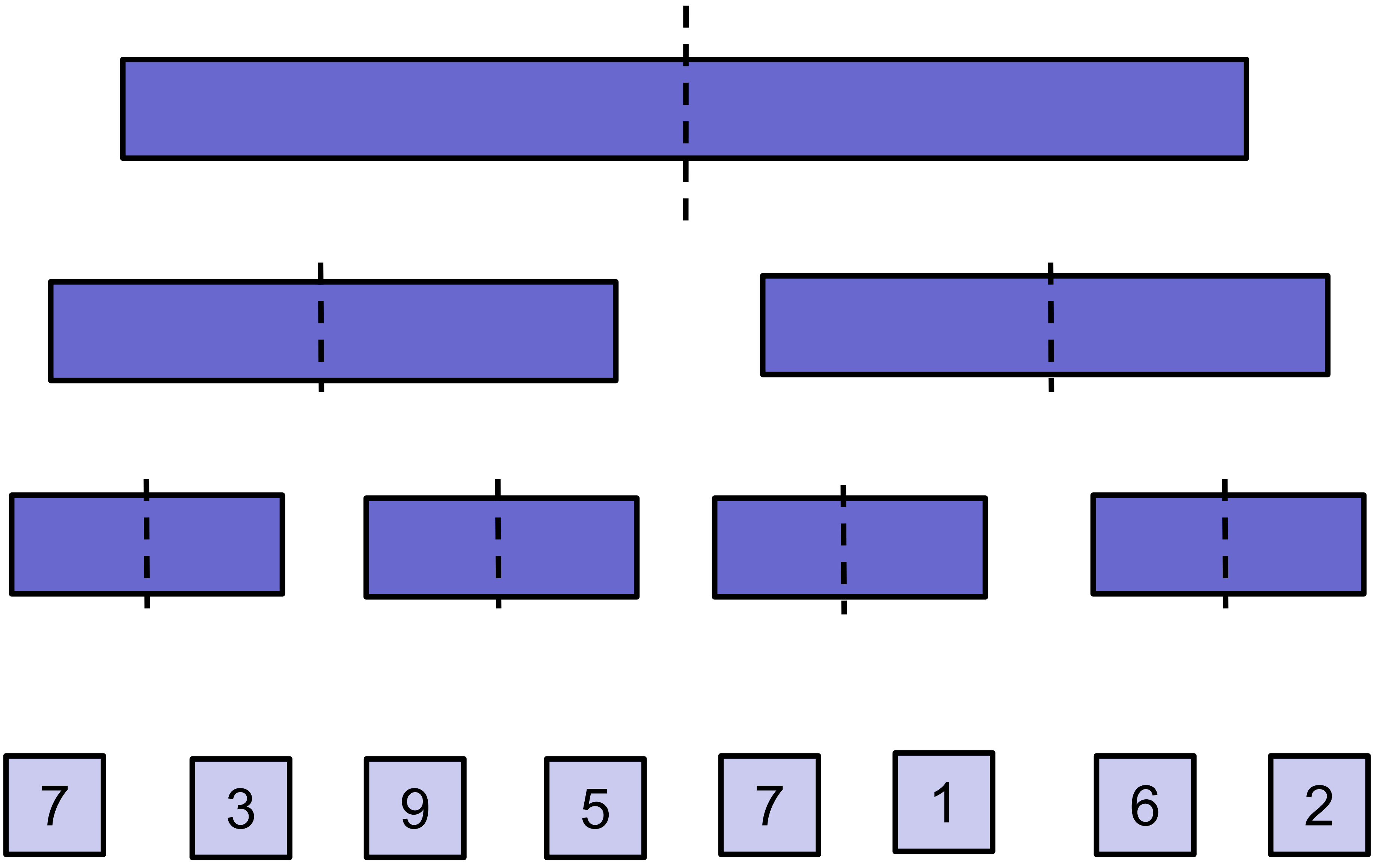
Divide-and-Conquer Sorting

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

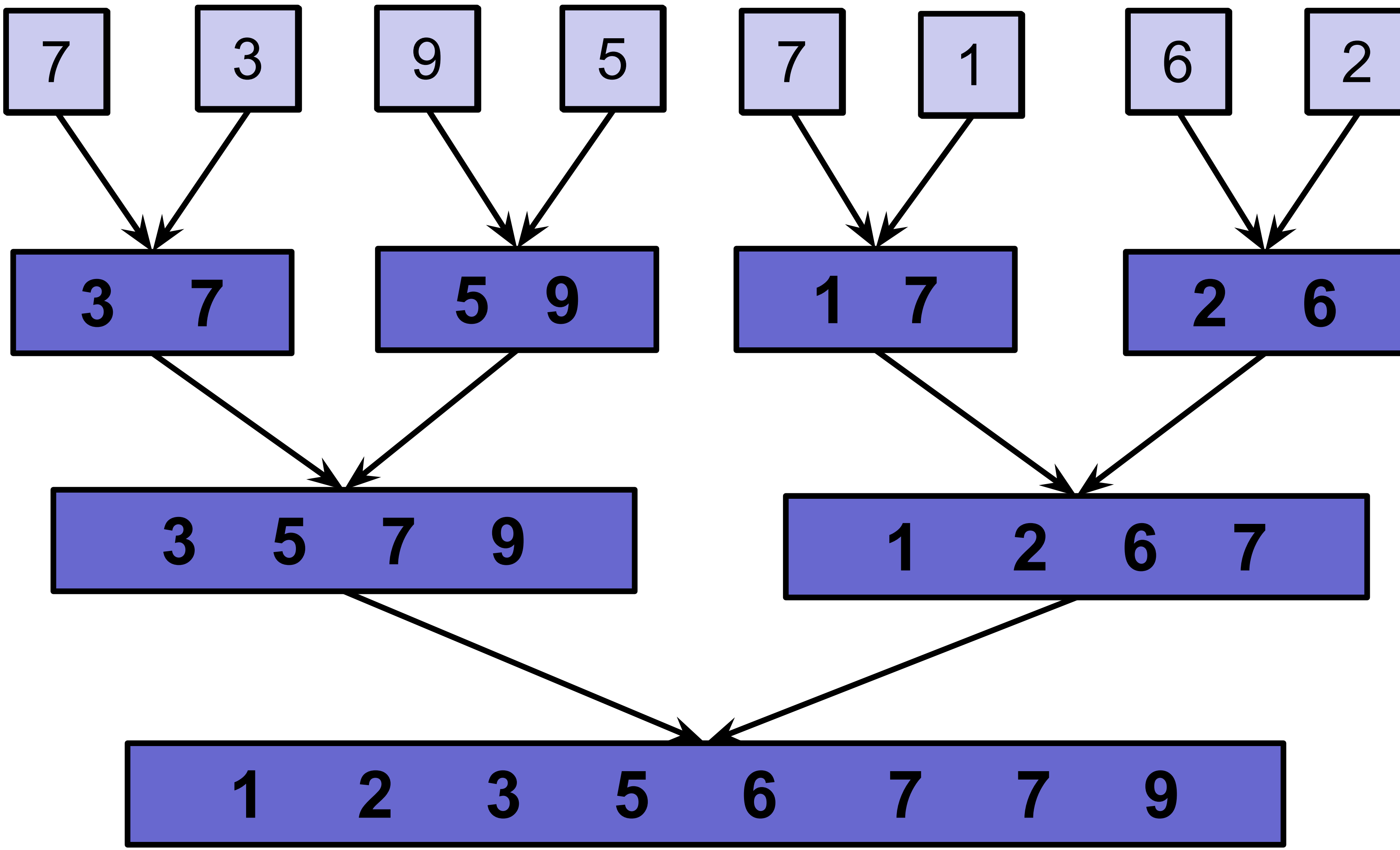
<u>Advice:</u>

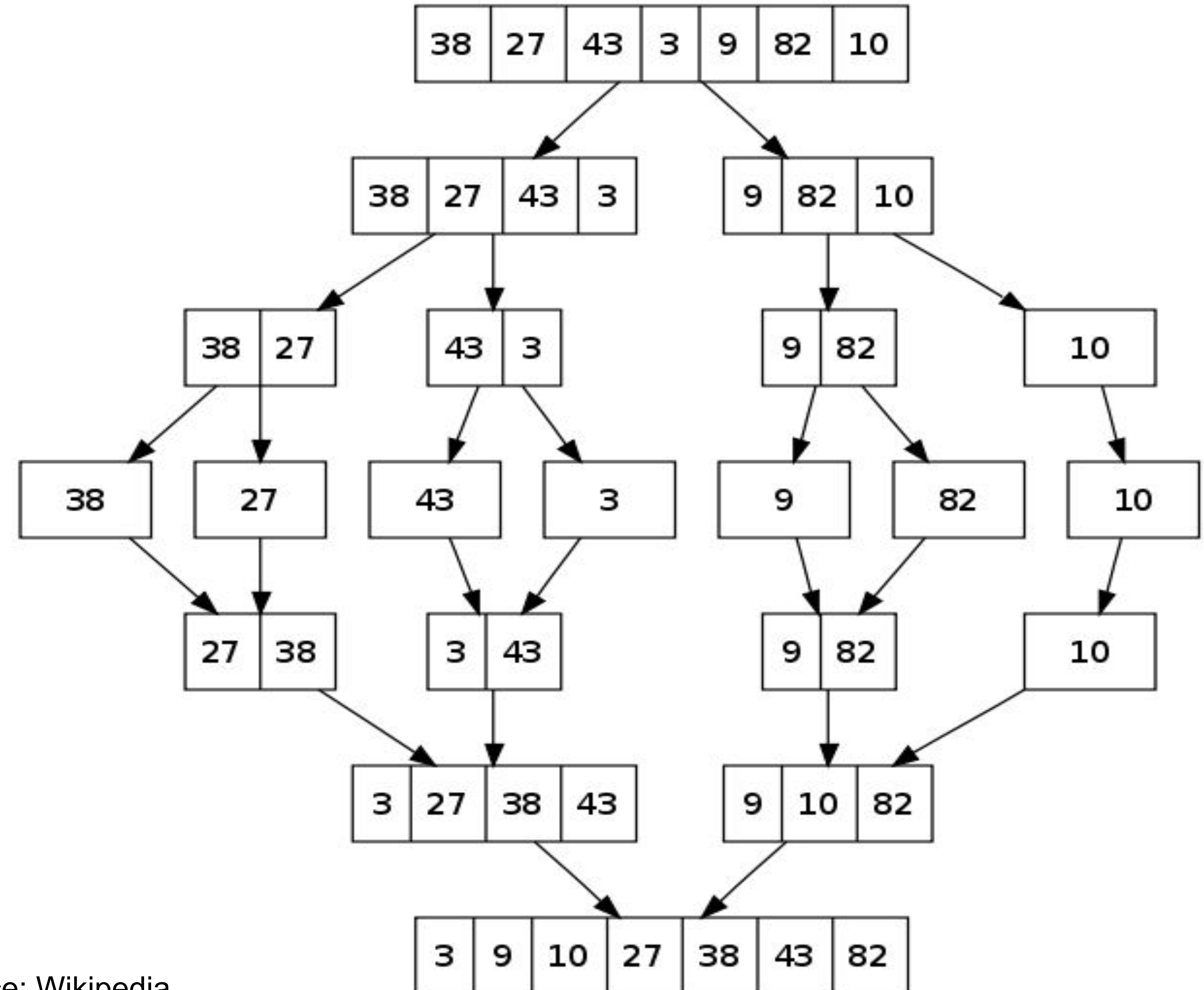
When thinking about recursion, do not "unroll" the recursion.
Treat the recursive call as a magic black box.
(But don't forget the base case.)

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

2	7	13	20

1	9	11	12

return result of left recursive call

return result of right recursive call

Merging Two Sorted Lists

2	7	13	20

1	9	11	12

Clarification:

We have this (only-once) allocated auxiliary array for the entire algorithm. Size: n

Total space complexity: O(n)

Merging Two Sorted Lists

2	7	13	20

1	9	11	12

Clarification:

We will merge the two lists into the allocated array.

Merging Two Sorted Lists

2	7	13	20

1	9	11	12

Clarification:

Then we can move the items back into the original array after we're done using it.

Merging Two Sorted Lists

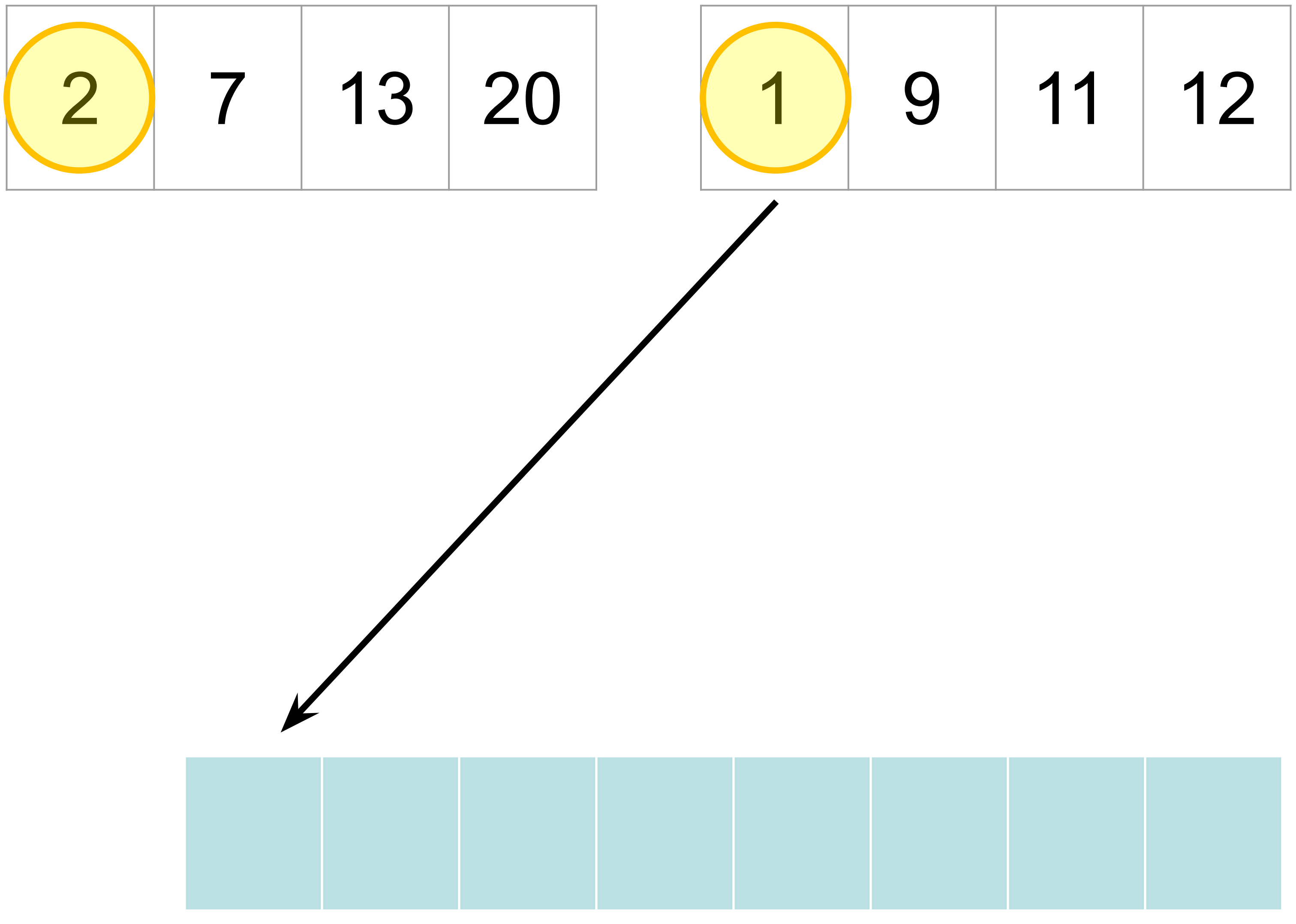
2	7	13	20

1	9	11	12

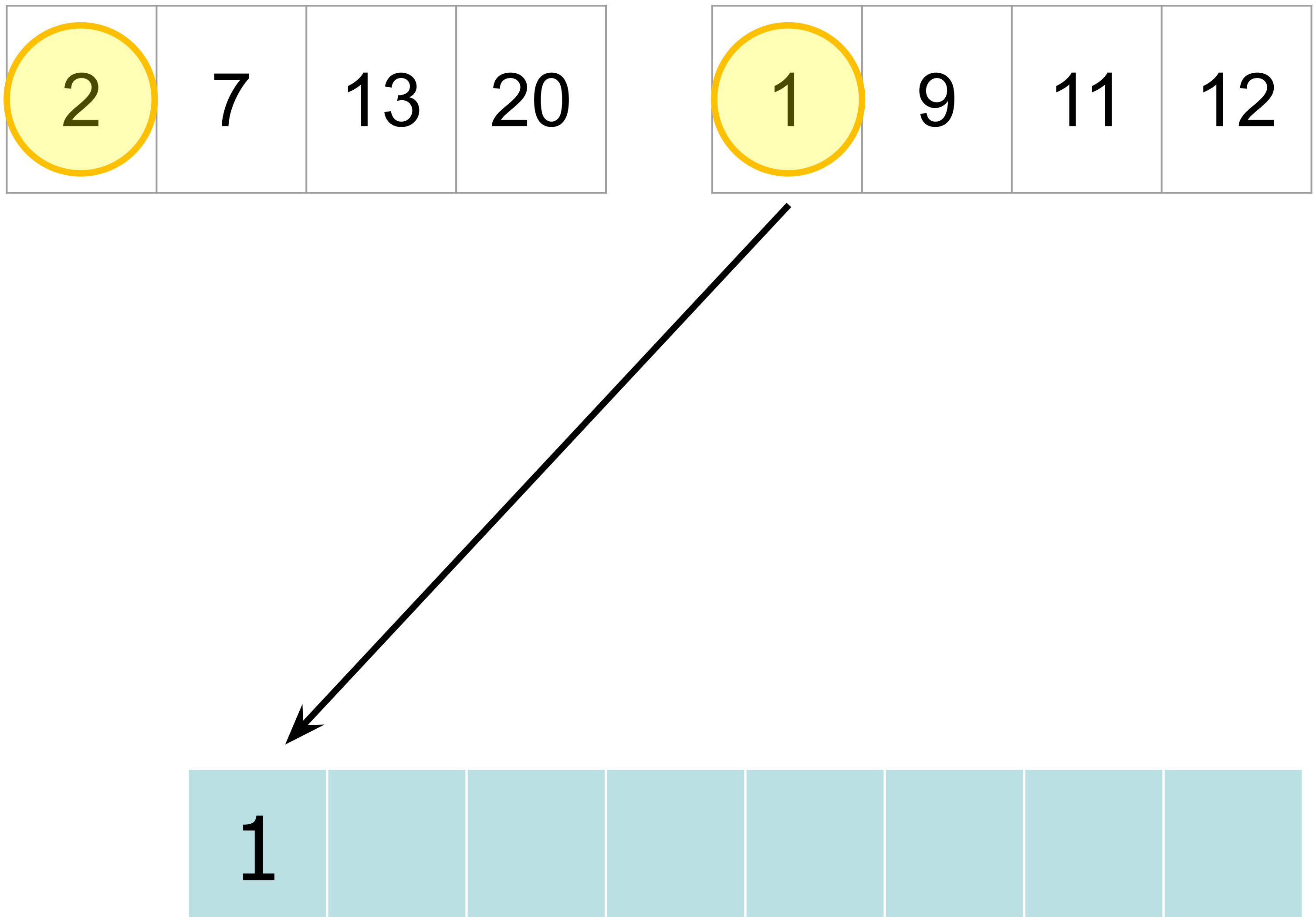
Clarification:

Then we can keep reusing the auxiliary array throughout all of the recursion.

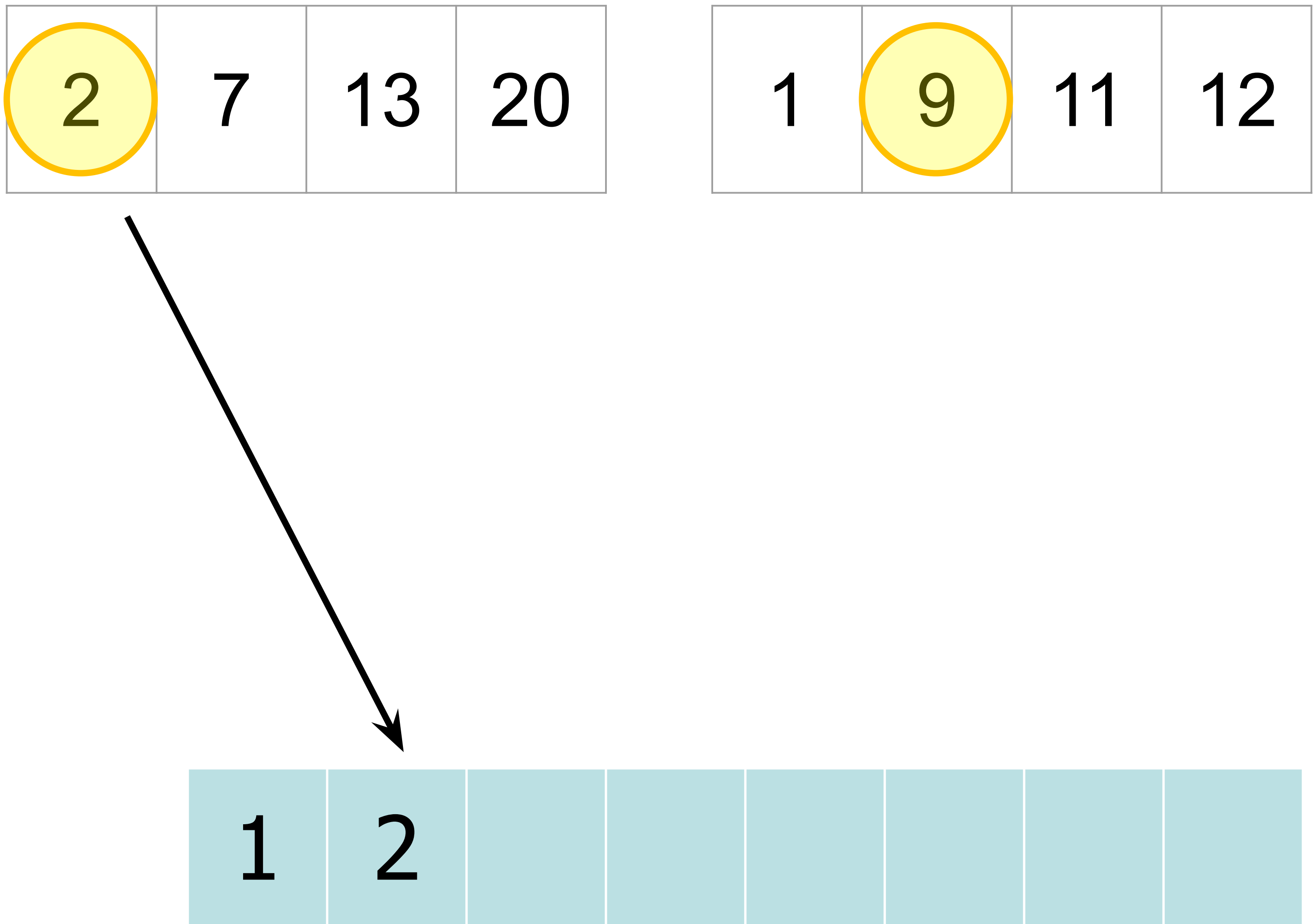
Merging Two Sorted Lists



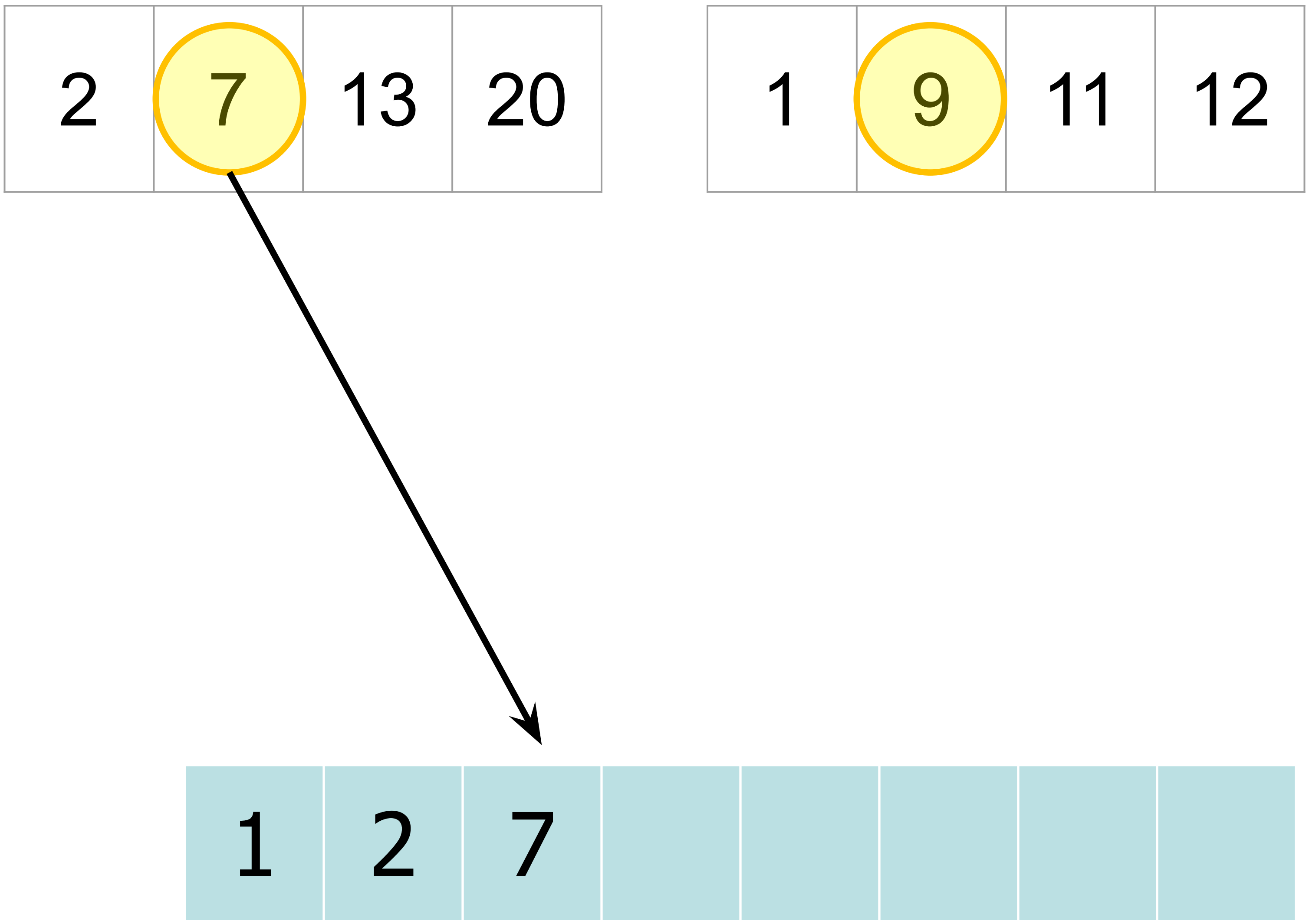
Merging Two Sorted Lists



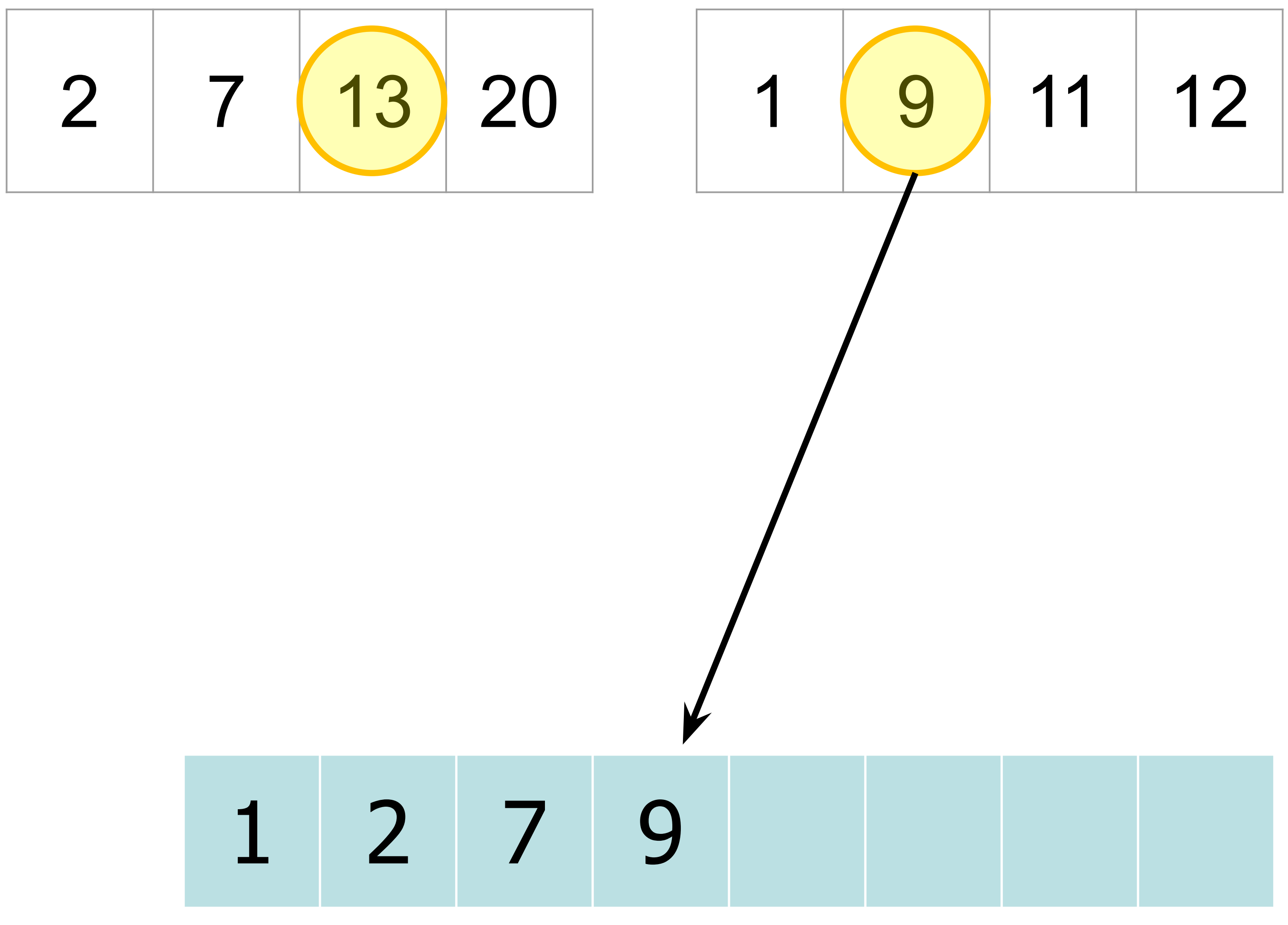
Merging Two Sorted Lists



Merging Two Sorted Lists



Merging Two Sorted Lists



Merging Two Sorted Lists

2	7	13	20

1	9	11	12

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: ??

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:

$X \leftarrow$ Merge-Sort**(**...**);** ←-------- $T(n/2)$

$Y \leftarrow$ 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.

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

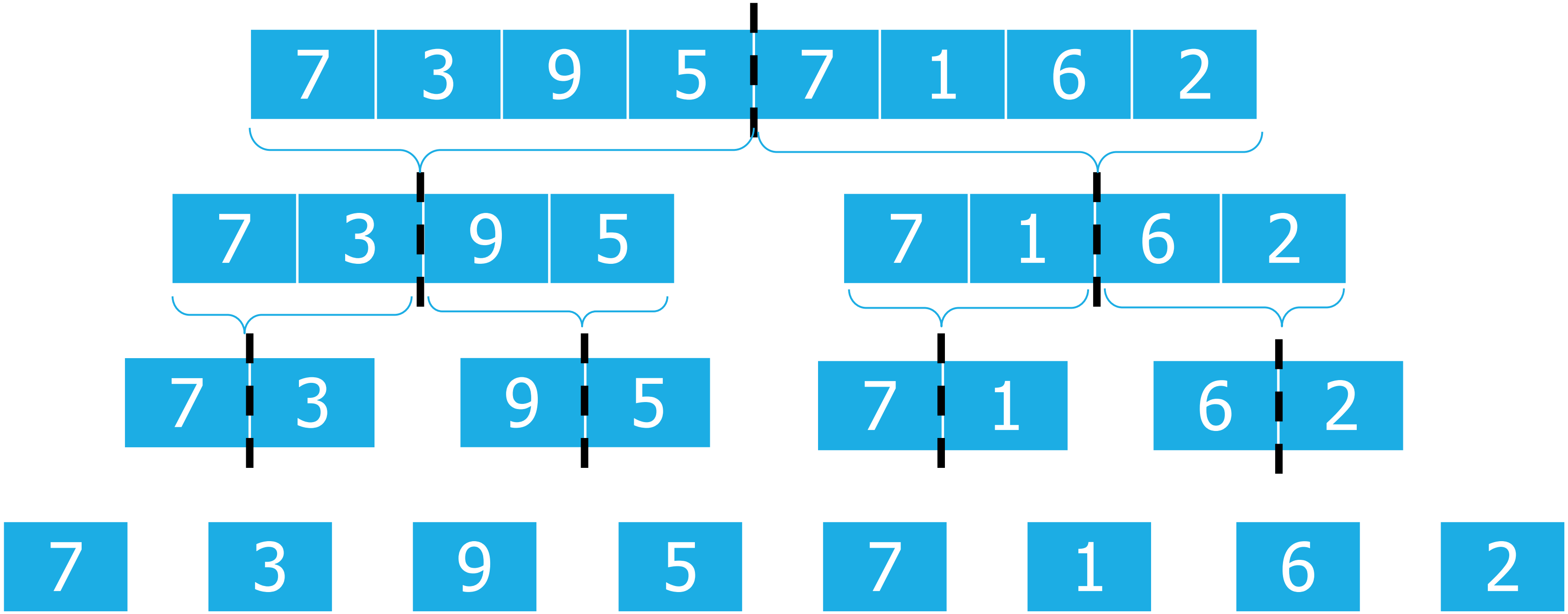
Techniques for Solving Recurrences

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

recursive sort

cn

recursive sort

T(n/2)

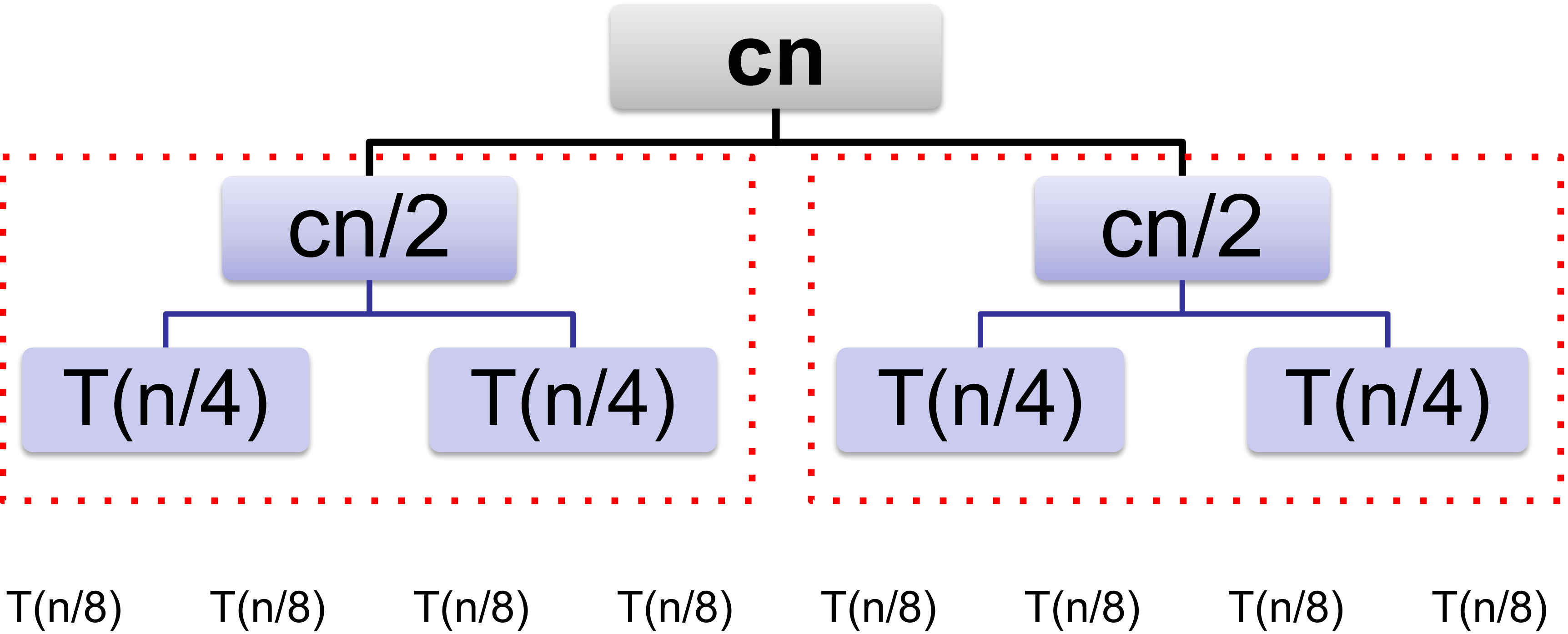
T(n/2)

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)

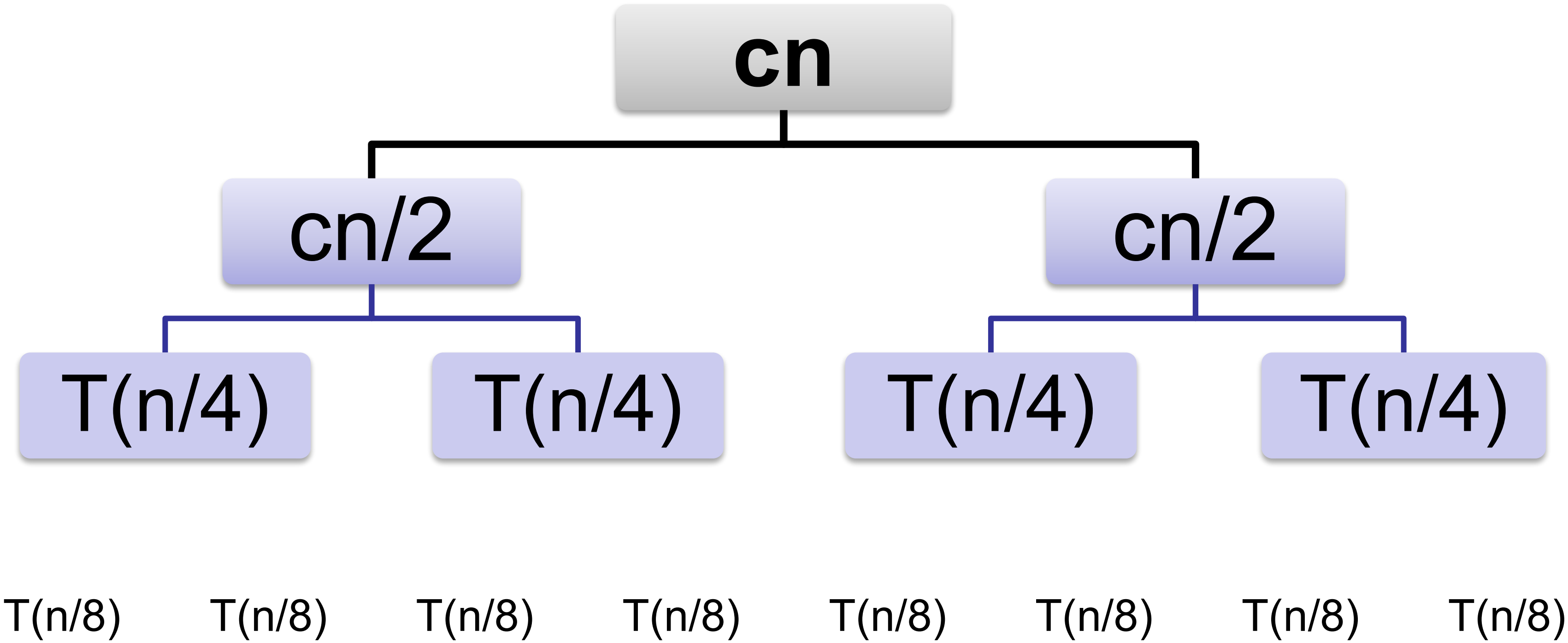
MergeSortAnalysis

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



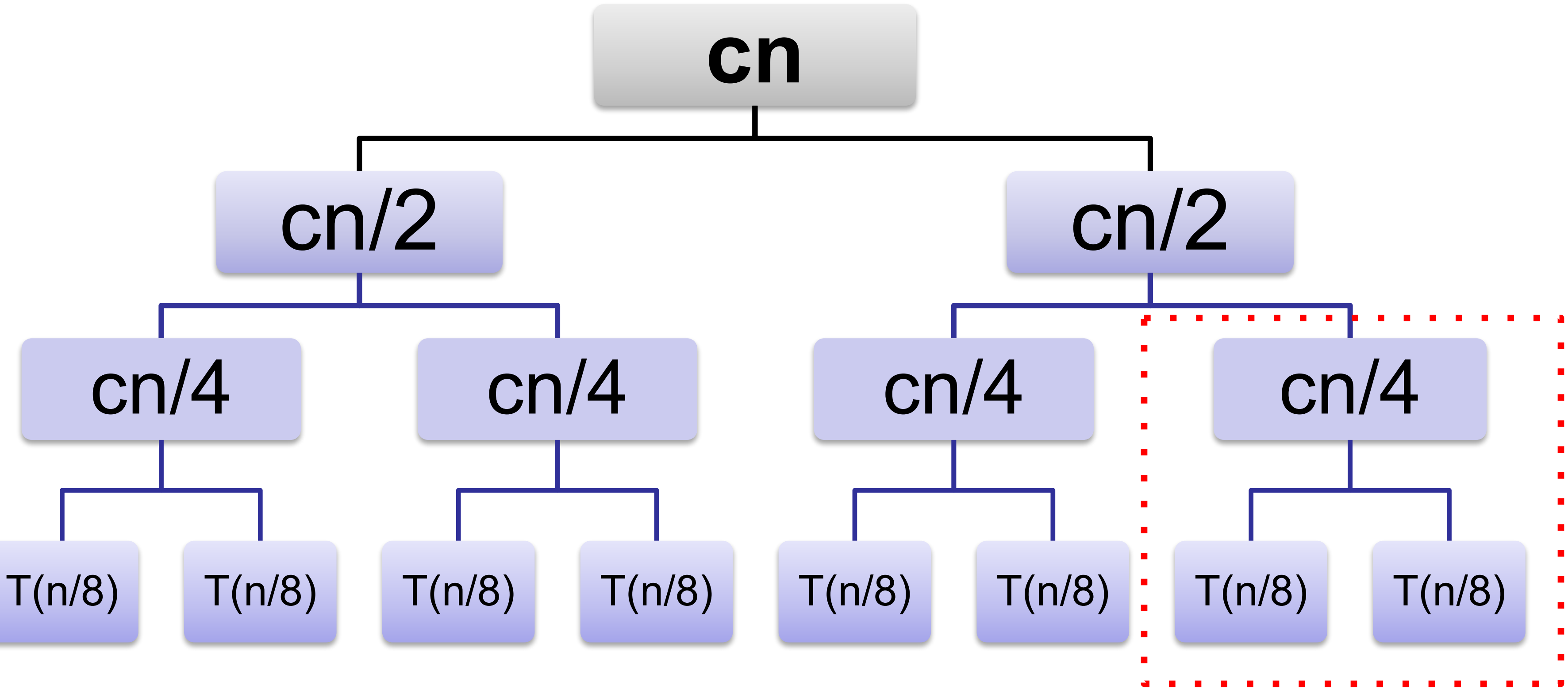
MergeSortAnalysis

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



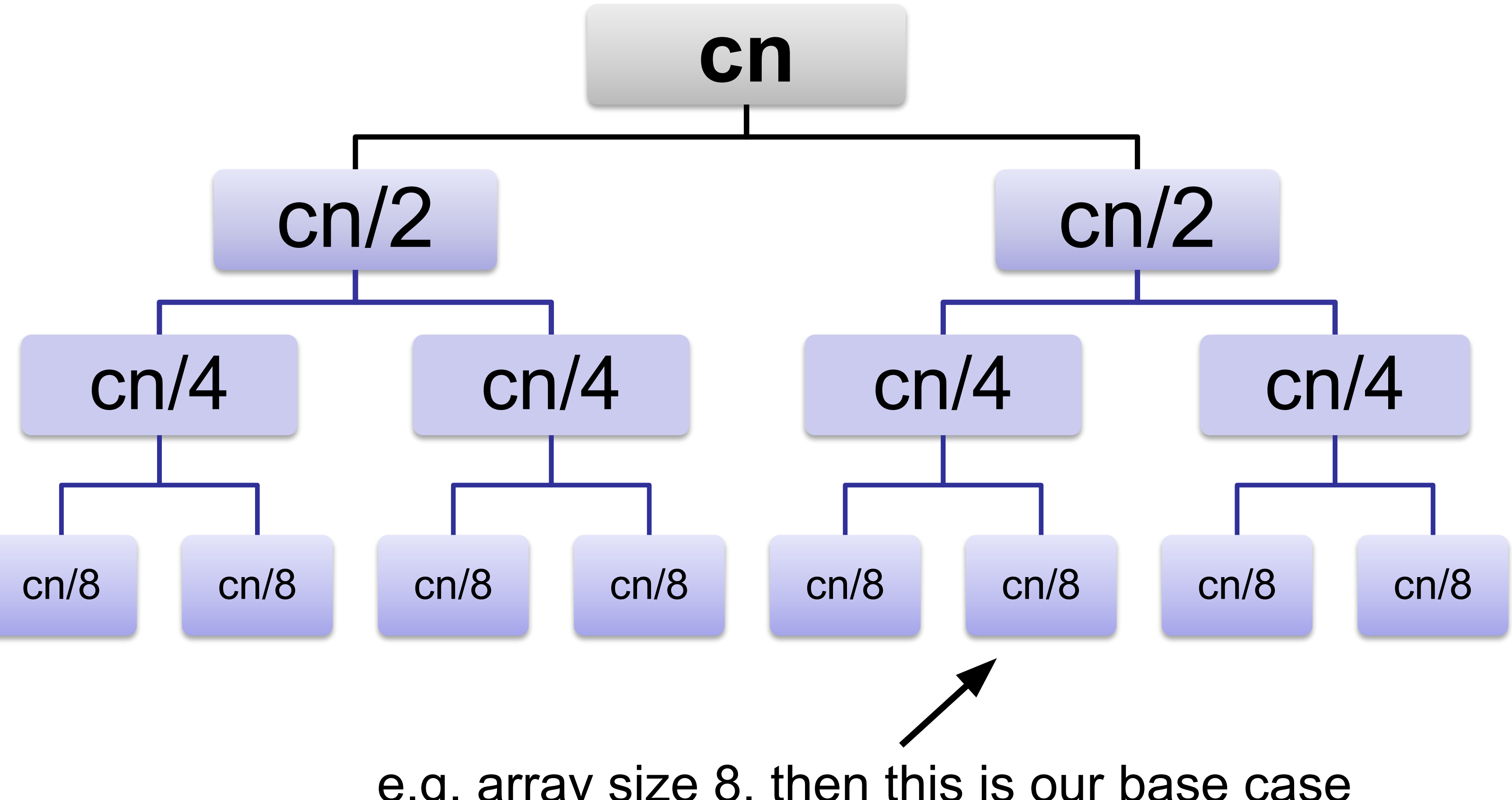
MergeSort Analysis

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



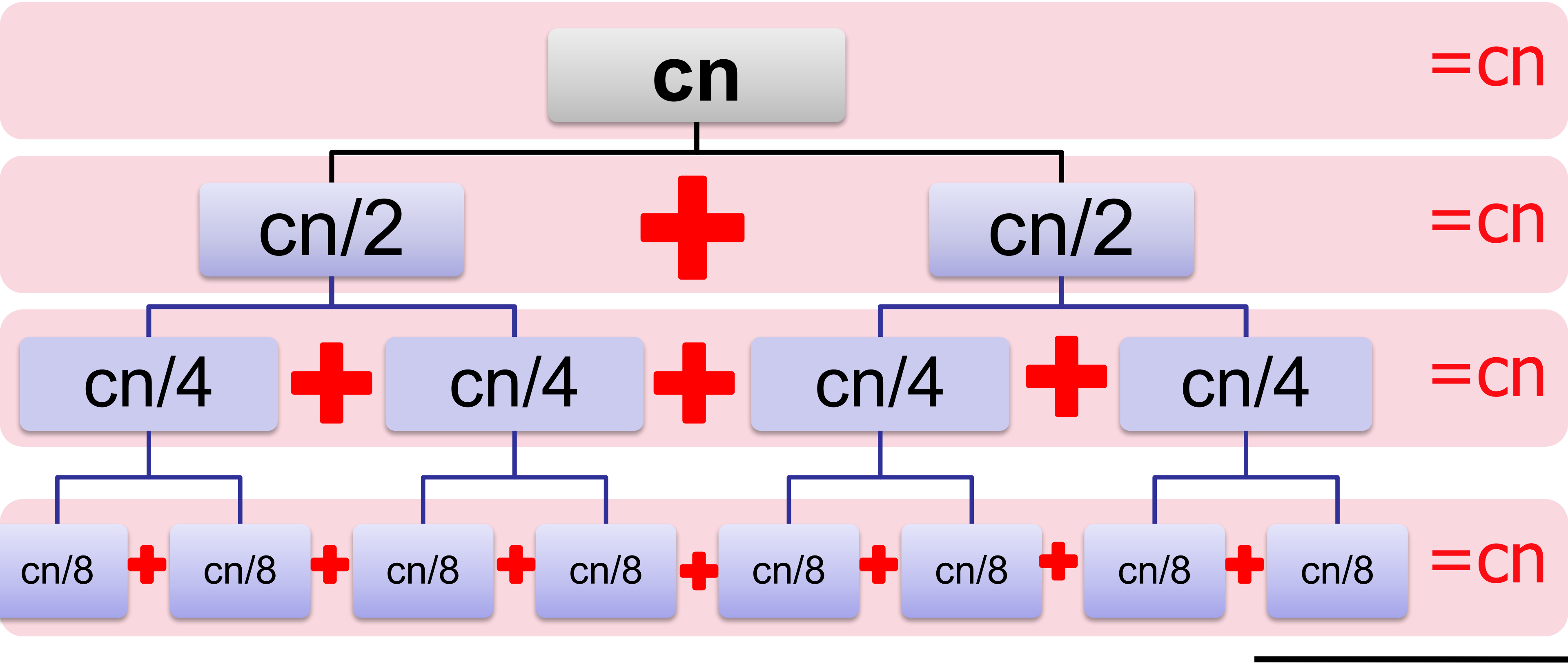
MergeSort Analysis

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



MergeSort Analysis

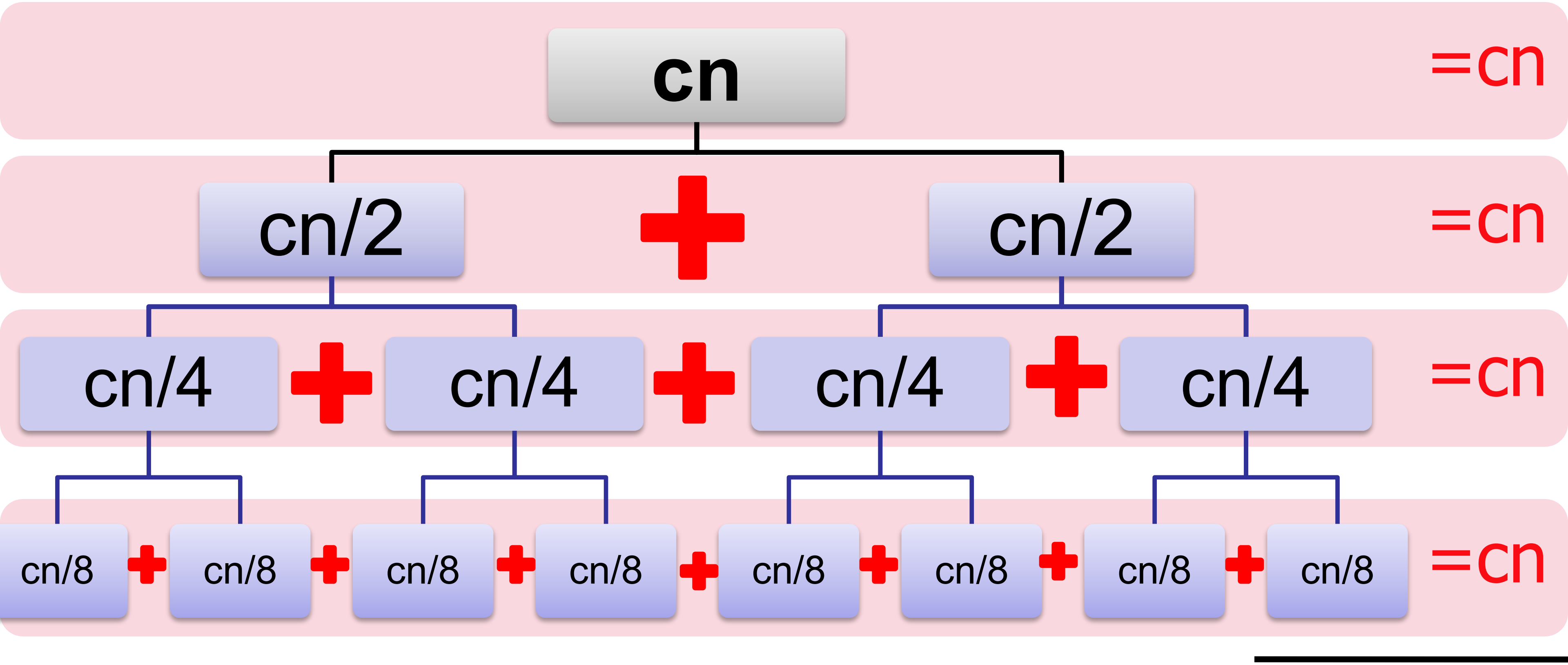
$T(n) = 2T(n/2) + cn$



Each level, we do O(n) work, regardless of level

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*

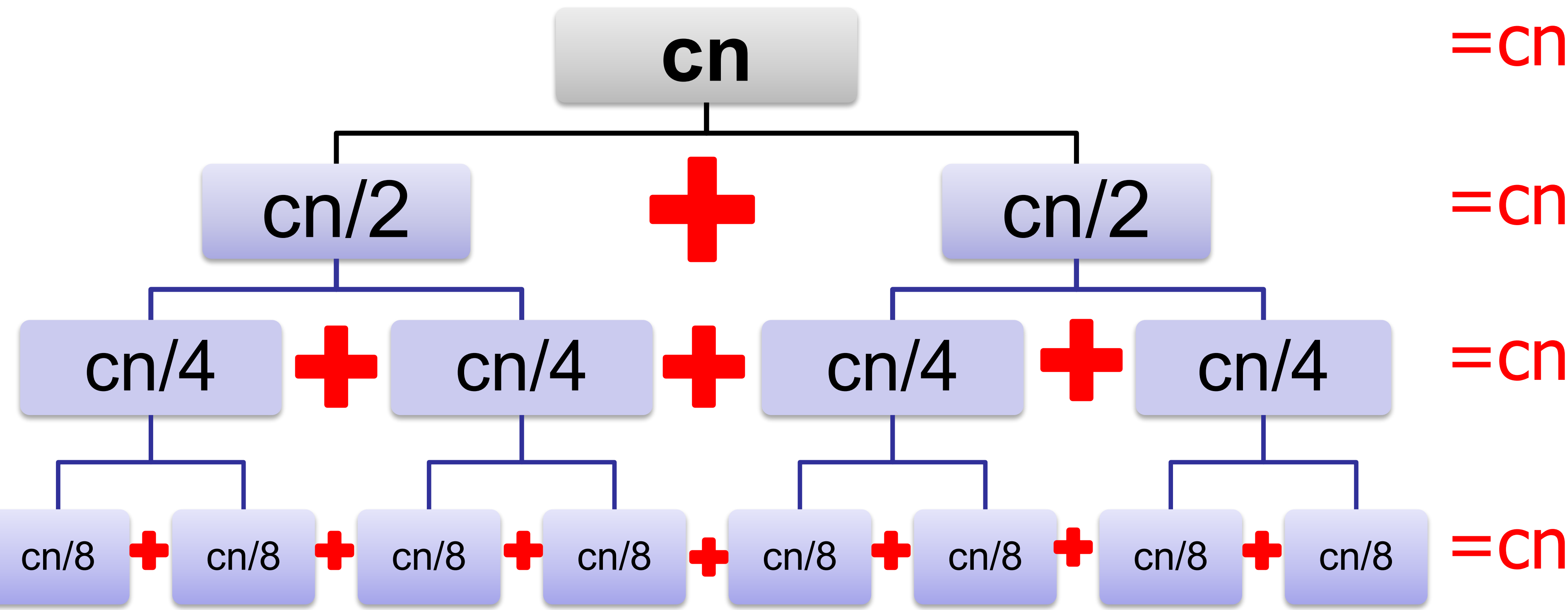
$$\text{number} = 2^{\text{level}}$$

$$n = 2^h$$

$$\log n = h$$

MergeSort Analysis

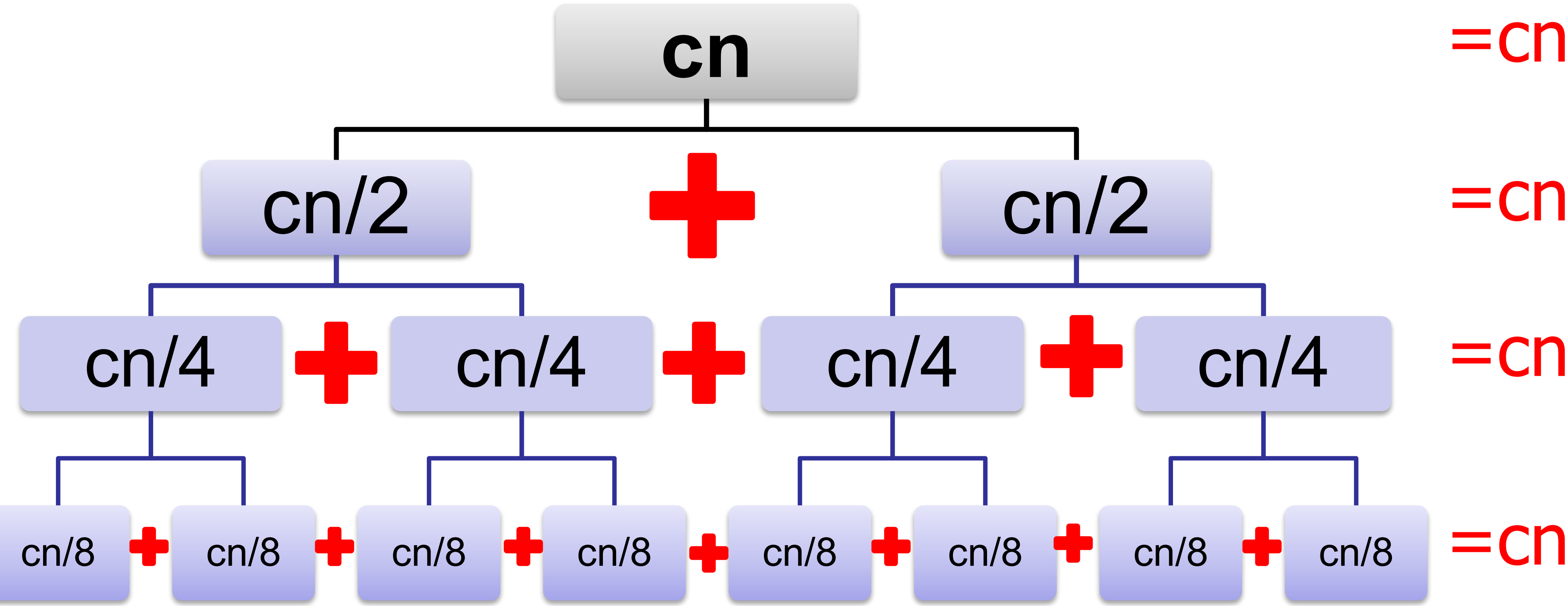
$$T(n) = 2T(n/2) + cn$$



Total work: work at level 1 + work at level 2 + …

MergeSort Analysis

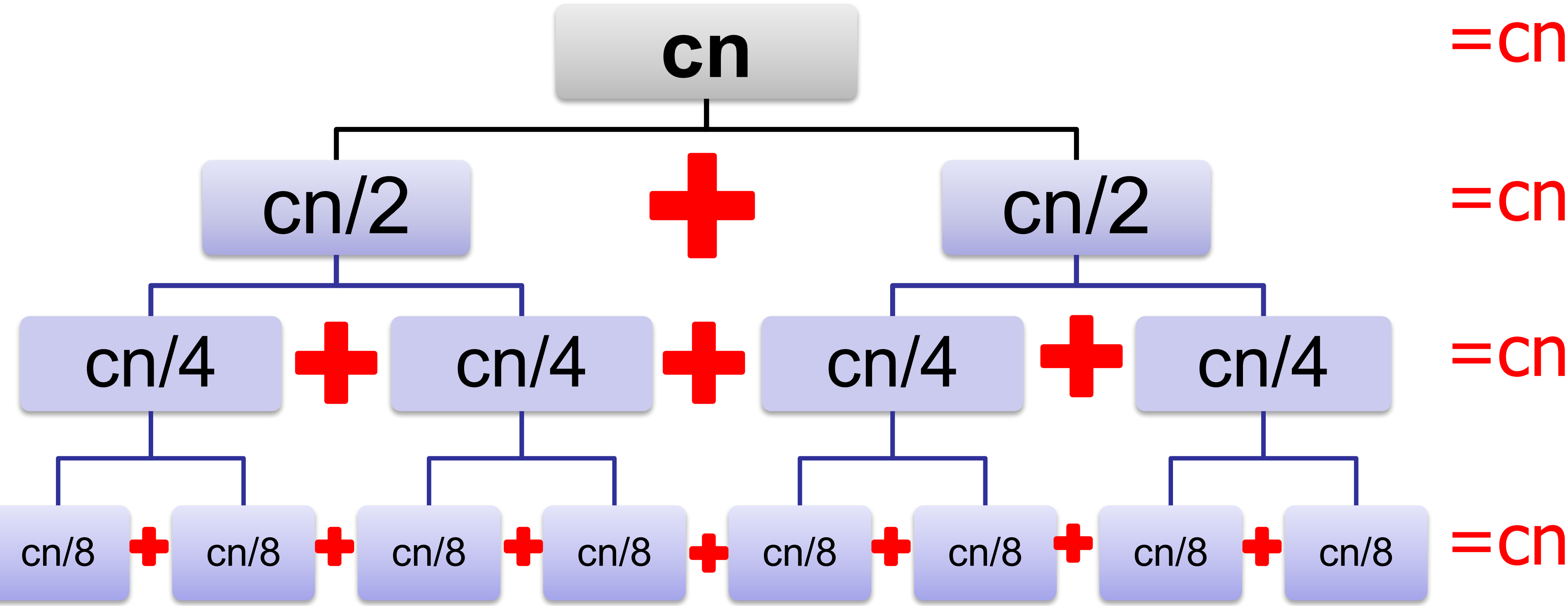
$T(n) = 2T(n/2) + cn$



Since work at each level is same:
(# of levels) x (# total work level 1)

MergeSort Analysis

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



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

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 techniqus.

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.

<u>Recurrence being analyzed:</u>

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

$T(1) = c$

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 $x < n$.

<u>Recurrence being analyzed:</u>

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

$T(1) = c$

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

Induction:
Prove for n.

$T(1) = c$

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

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

Recurrence being analyzed:

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

$T(1) = c$

Induction:
Prove for n.

Guess: T(n) = c·n log n

T(1) = c

T(x) = c·x log x for all x < n.

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

Recurrence being analyzed:

T(n) = 2T(n/2) + c·n

T(1) = c

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

Induction:
Prove for n.

$T(1) = c$

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

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

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.

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

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

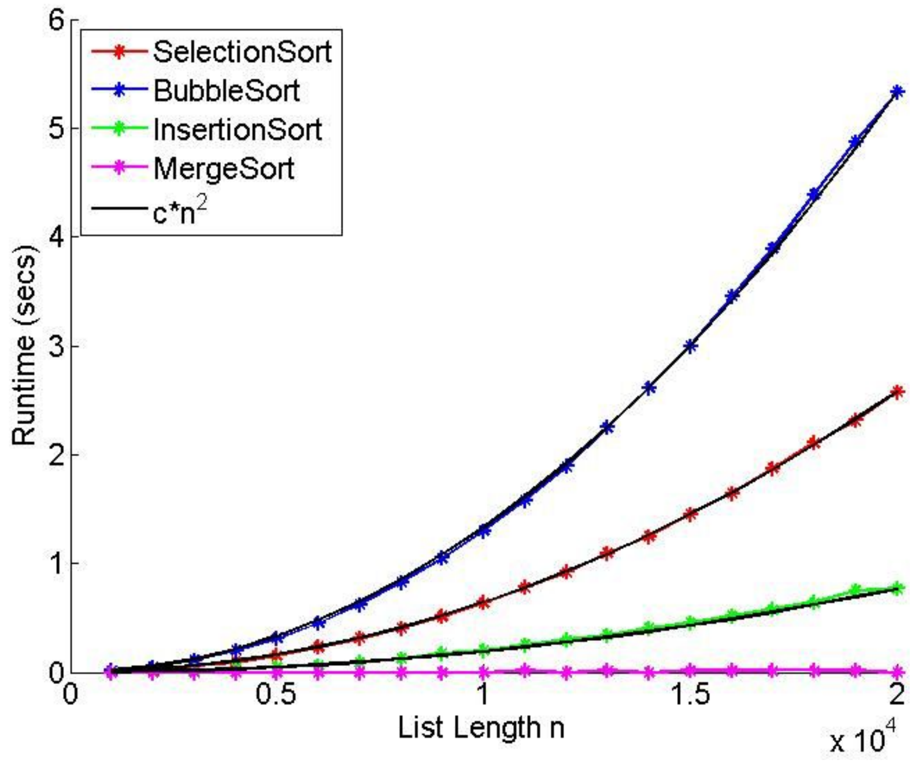
Performance Profiling

(*Dracula* vs. *Lewis & Clark*)

Version	Change	Running Time
Version 1		4,311.00s
Version 2	Better file handling	676.50s
Version 3	Faster sorting	6.59s
Version 4	No sorting!	2.35s

V.2 ⟹ V.3 was using MergeSort instead of SelectionSort.

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?

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?

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.
- How much extra space?

Challenge of the Day 2:

How much space does MergeSort need to sort n items?

(Use the version presented today.)

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

MergeSort

Stability:

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

Sorting Analysis

Summary:

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

Properties: time, space, stability

Also:

The power of divide-and-conquer!

How to solve recurrences…

Slowest Sorting Algorithm?

Step 1:

- Generate all the permutations of the input.

Step 2:

- Sort the permutations (by number of inversions).

Step 3:

- Return the first element in the sorted list of permutations.

Slowest Sorting Algorithm?

Step 1:

– Generate all the permutations of the input.

Step 2:

– Sort the permutations (by number of inversions).

Use BogoSort!

Roughly: $O((n!)!))$

Step 3:

– Return the first element in the sorted list of permutations.

Slowest Sorting Algorithm?

Step 1:

- Generate all the permutations of the input.

Step 2:

- Sort the permutations (by number of inversions).

Recurse!
Recursive instance is larger than original!

Step 3:

- Return the first element in the sorted list of permutations.

Slowest Sorting Algorithm?

Step 1:

– Generate all the permutations of the input.

Step 2:

– Sort the permutations (by number of inversions).

Recurse!
After n! recursions, use QuickSort for the "base case".

Step 3:

– Return the first element in the sorted list of permutations.

Ingrassia-Kurtz Sort

Step 1:

– Generate all the permutations of the input.

Step 2:

– Sort the permutations (by number of inversions).

Recurse!
After n! recursions, use QuickSort for the "base case".

Step 3:

– Return the first element in the sorted list of permutations.

For next time…

Next Monday class:

More sorting!