CS2040S
Data Structures and Algorithms

Welcome!

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.

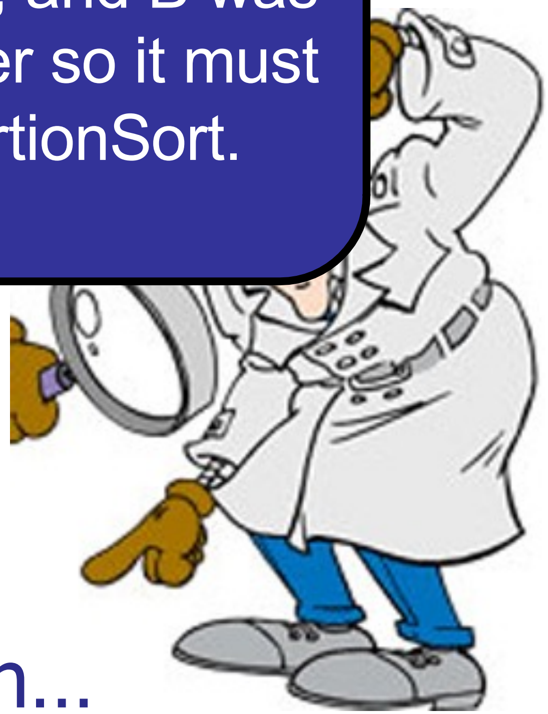
It ran the fastest so it must be QuickSort.

operties:

mance

ability

erformance on special inputs



- Absolute speed is not a good reason…

Problem Set 3

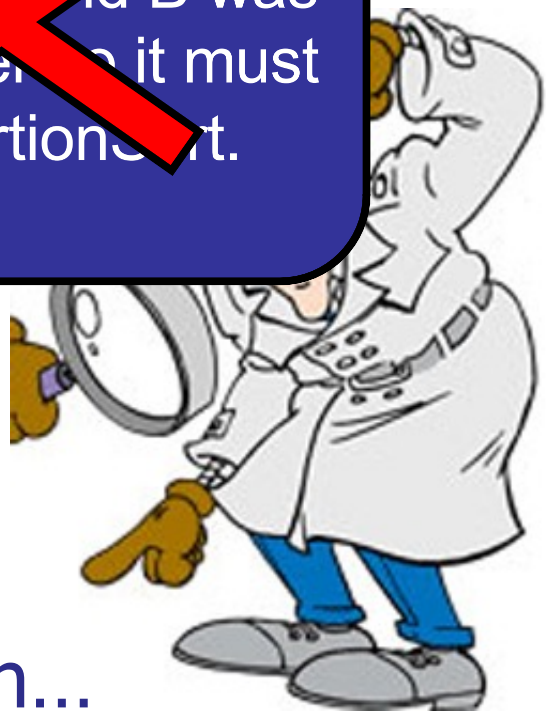
Sorting Detective

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

~~I co ared th peed of A an d B was much f o it must b InsertionS t.~~

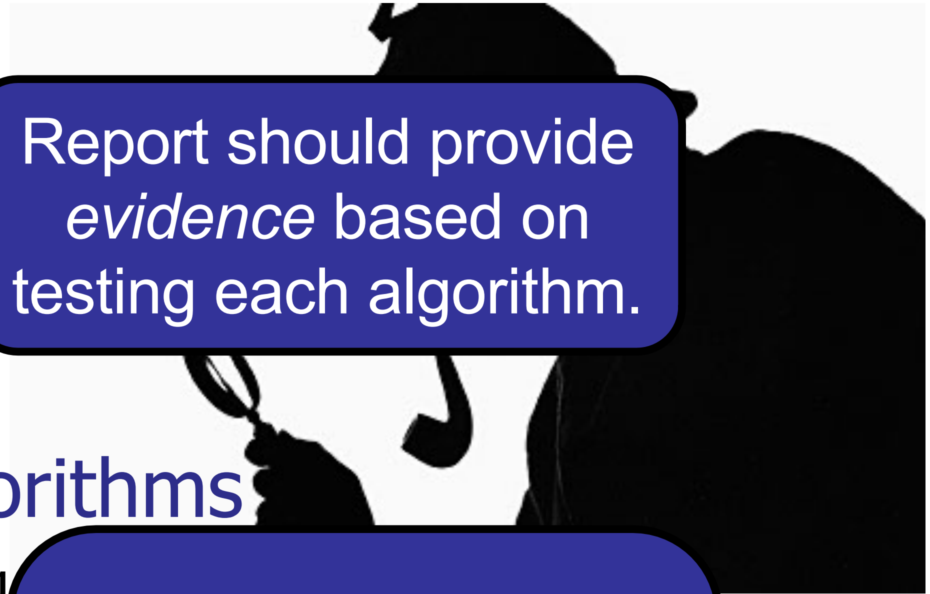
~~It ran t f est so it must b ickSort.~~

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….

Admin

Recitations start this week!

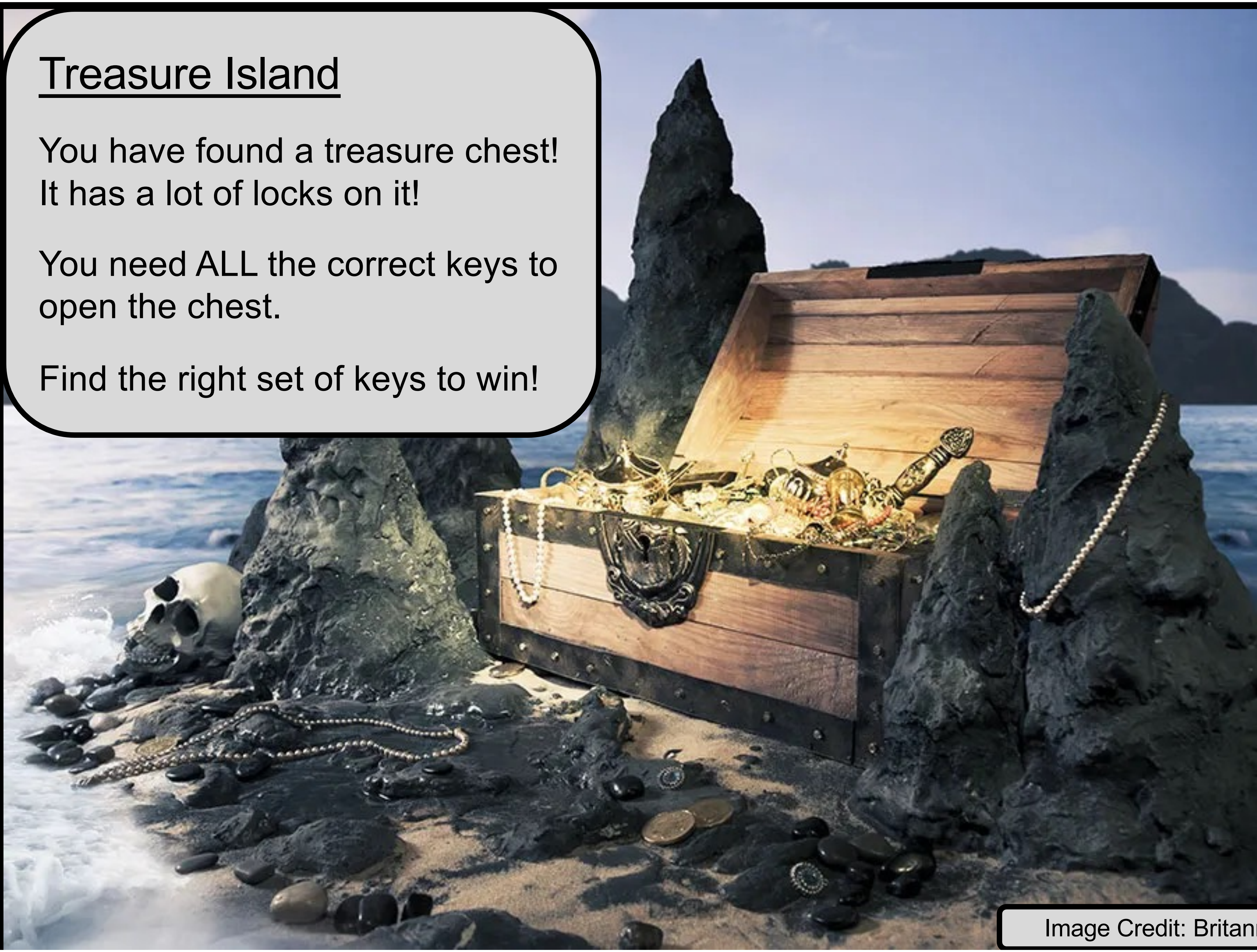
Tutorials start this week!

Part 1: Review (more this week)

Part 2: Harder questions (only one optional this week)

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

Contest closes Friday



Treasure Island

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

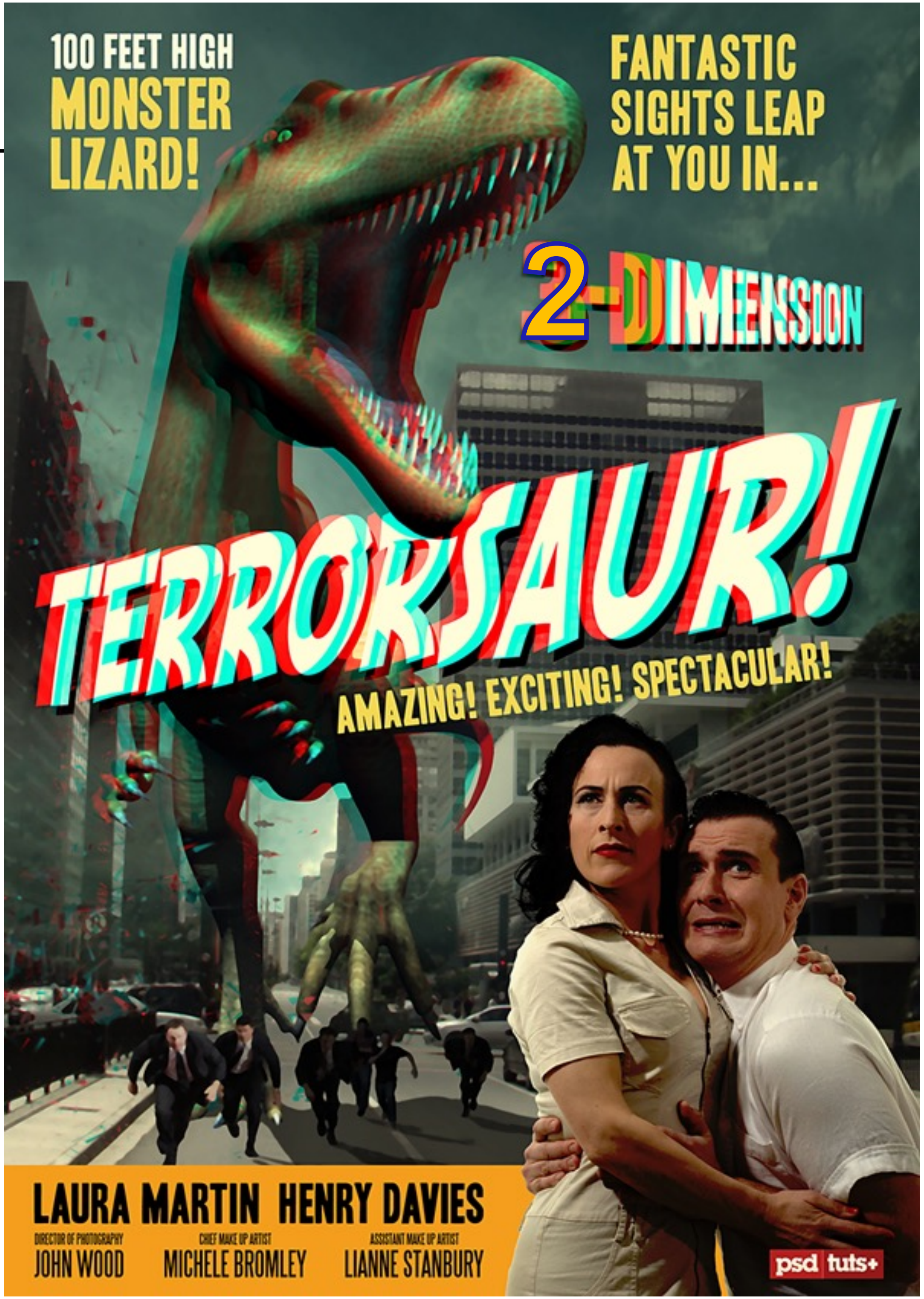
You need ALL the correct keys to open the chest.

Find the right set of keys to win!

Image Credit: Britannica

Postponed…

2D Peak Finding



Peak Finding 2D (the sequel)

Given: 2D array A[1..n, 1..m]

m

n

10	8	5	2	1
3	2	1	5	7
17	5	1	4	1
7	9	4	6	4
8	1	1	2	6

Output: a peak that is not smaller than the (at most) 4 neighbors.

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?

ARCHIPELAGO

is open

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

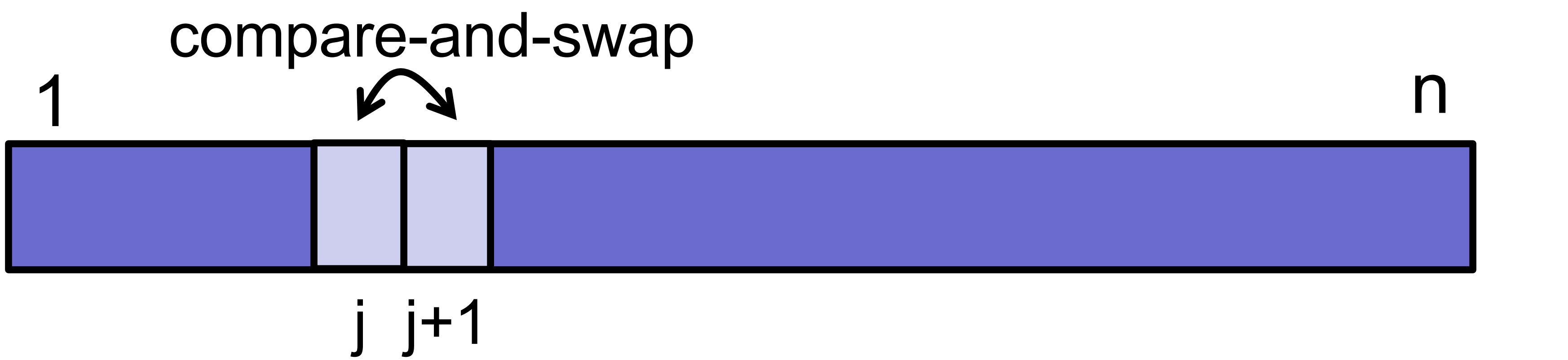
BubbleSort

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:

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

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:

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

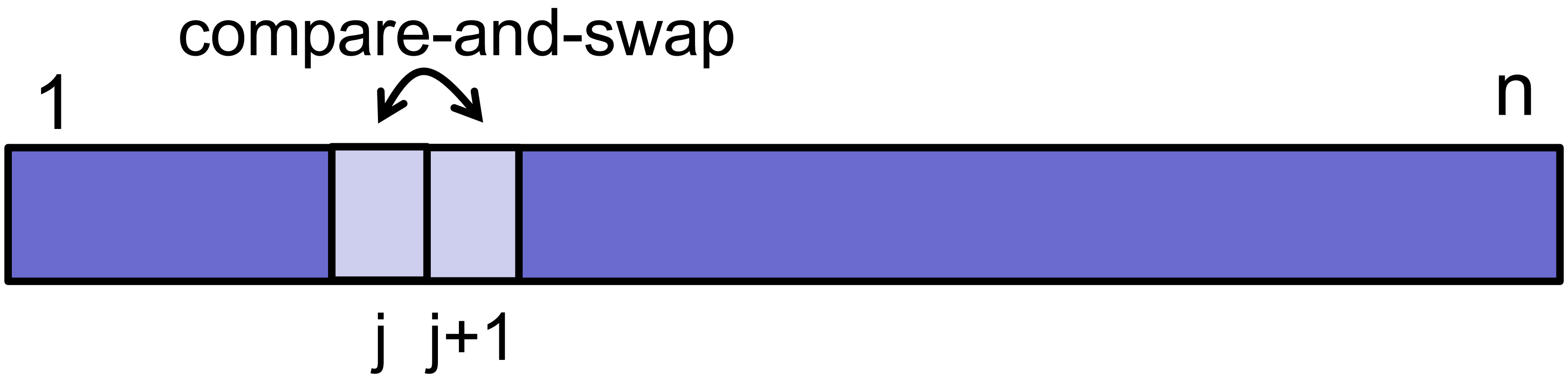
BubbleSort

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])



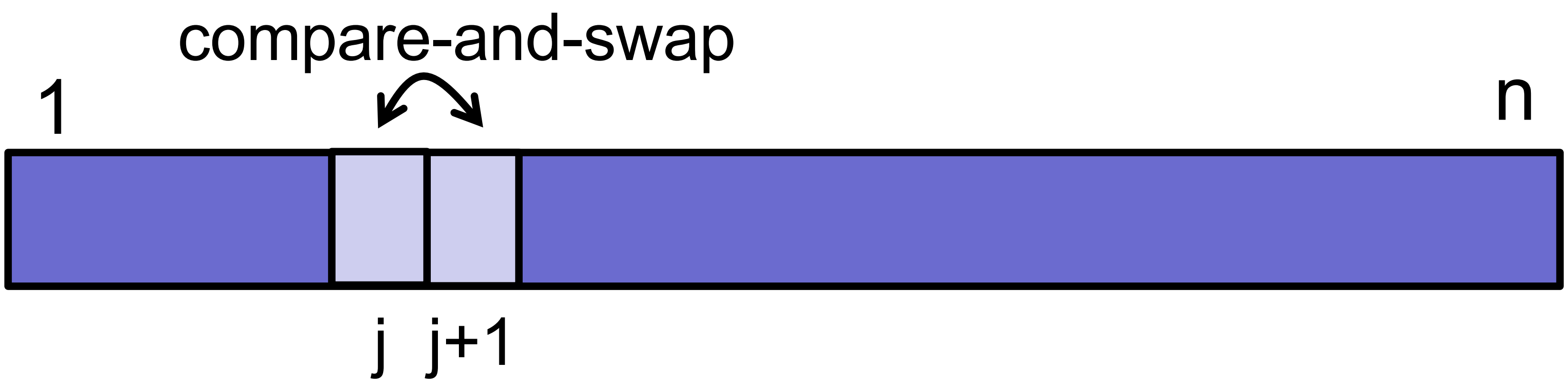
BubbleSort

BubbleSort(A, n)

repeat (until no swaps) **:**

for $j \leftarrow 1$ **to** $n-1$

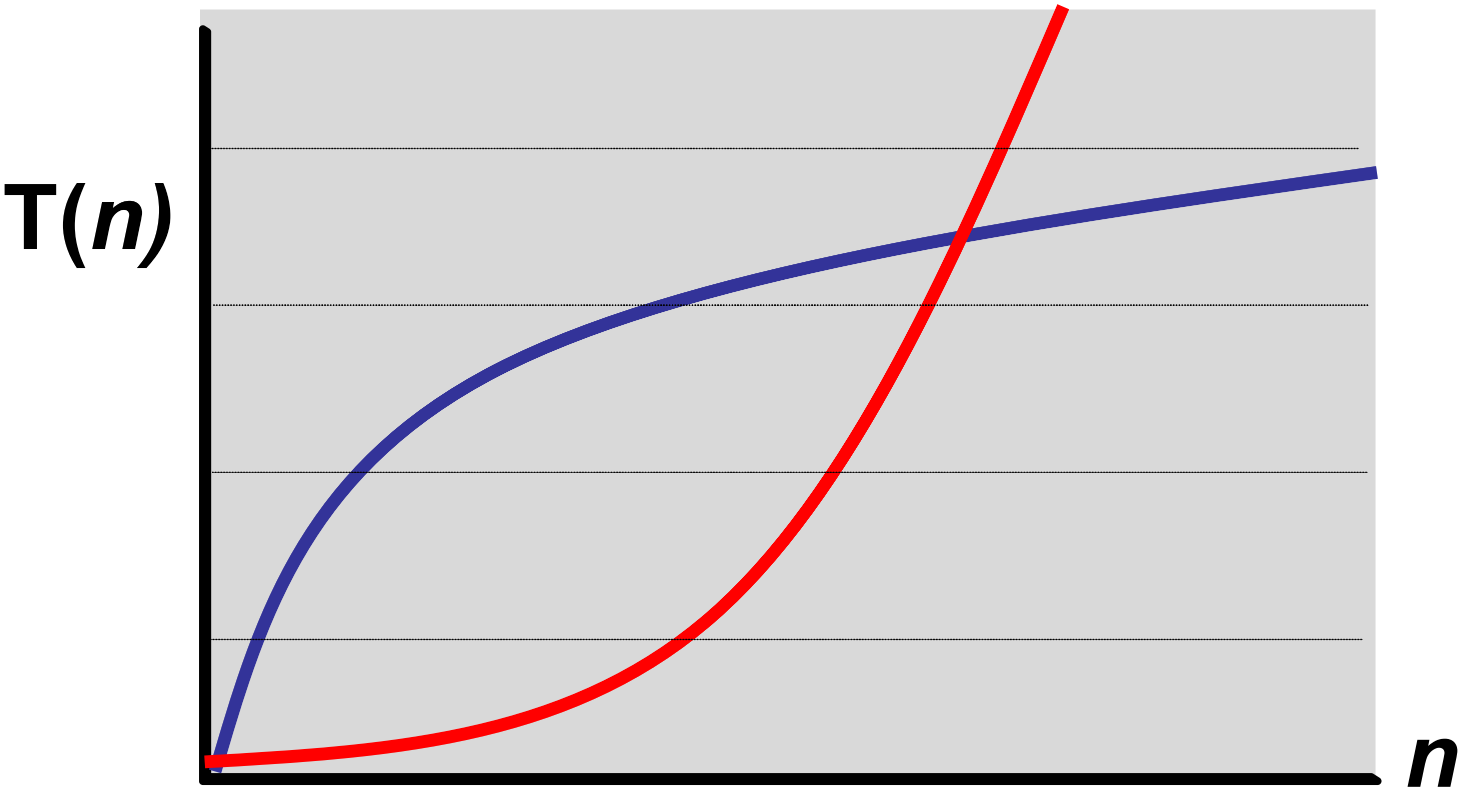
if $A[j] > A[j+1]$ **then** swap(A[j], A[j+1])



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. $O(2^n)$

ARCHIPELAGO
is open

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.

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

Worst-case:

- Max running time over all possible inputs.

BubbleSort Analysis

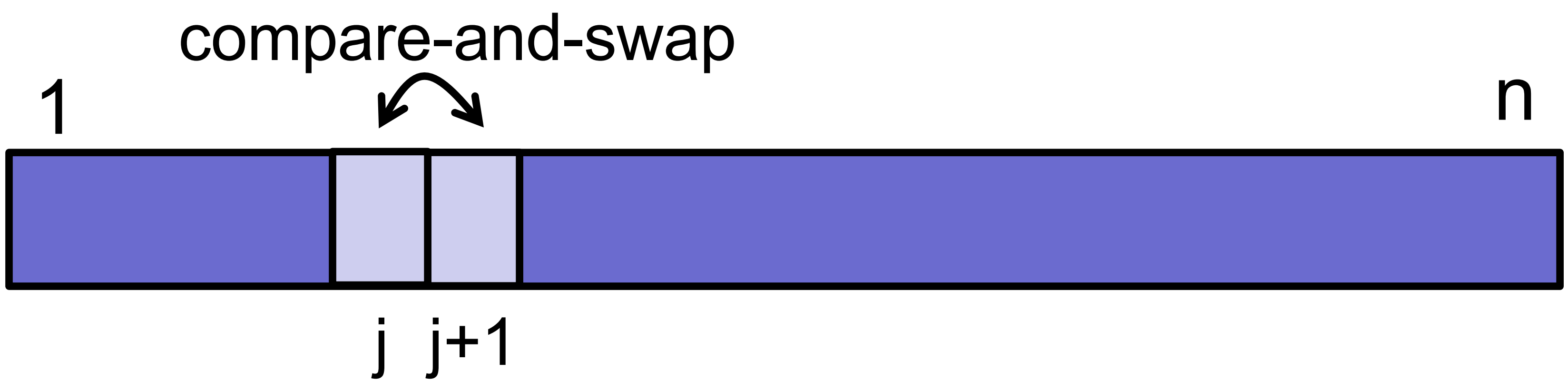
How many iterations do we need?

BubbleSort(A, n)

repeat (until no swaps) **:**

  for j $\leftarrow$ 1 **to** n-1

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



BubbleSort Analysis

ARCHIPELAGO

is open

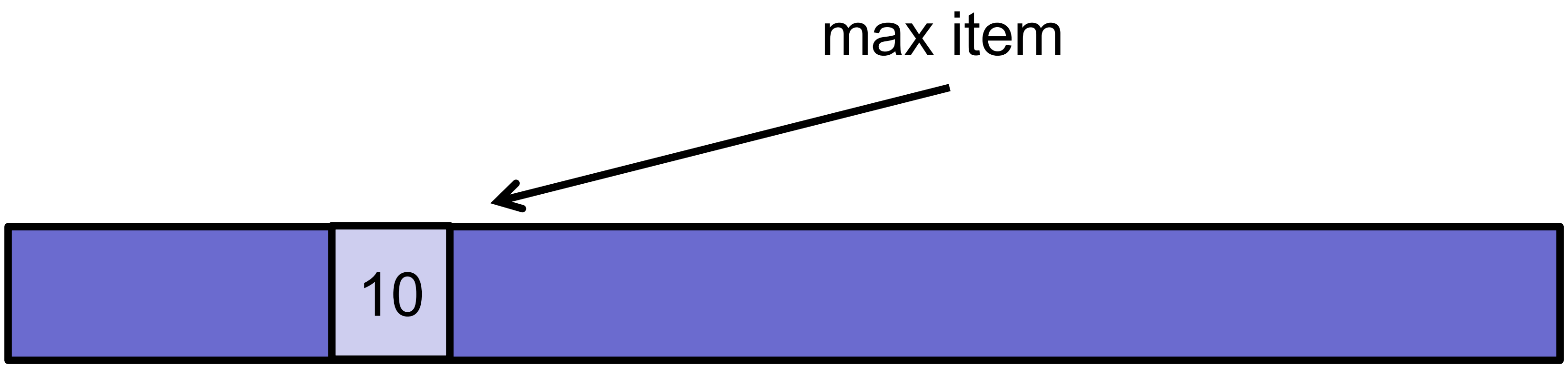
BubbleSort(A, n)

repeat (until no swaps) **:**

for j $\leftarrow$ 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

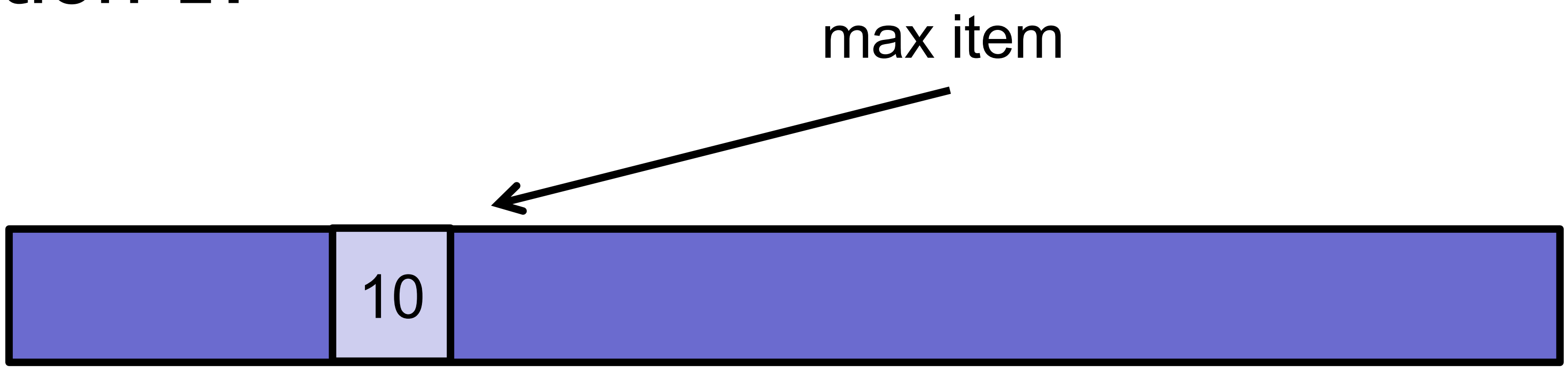
BubbleSort(A, n)

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

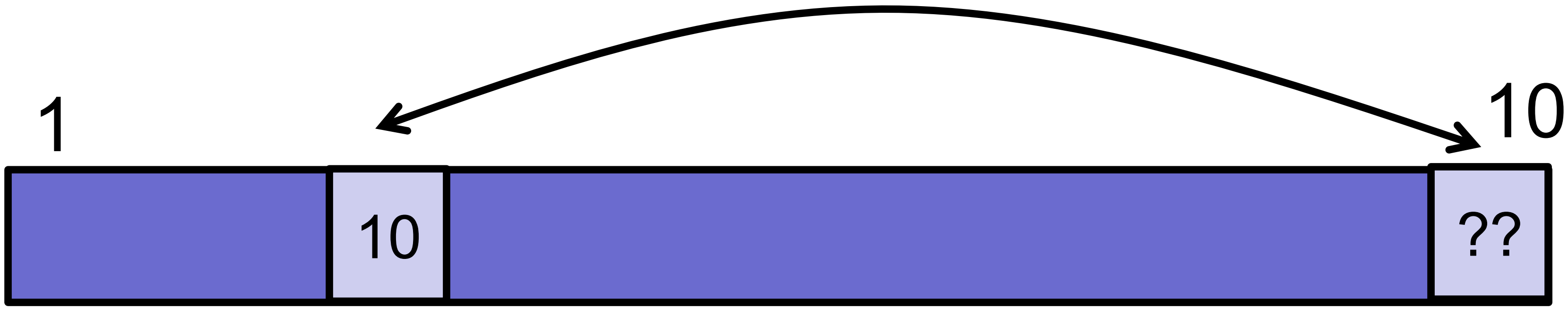
BubbleSort(A, n)

    repeat (until no swaps) **:**

        for j $\leftarrow$ 1 **to** n-1

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

Iteration 1:



BubbleSort Analysis

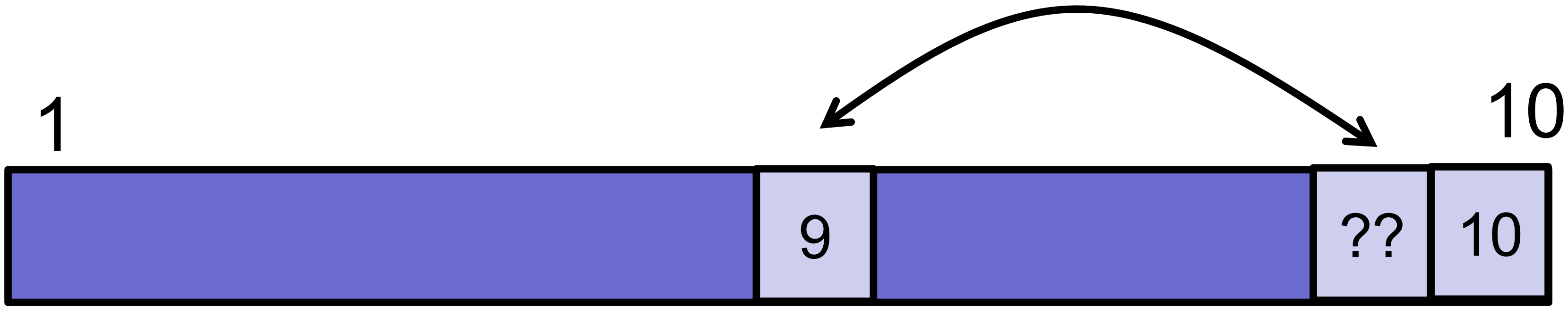
BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

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

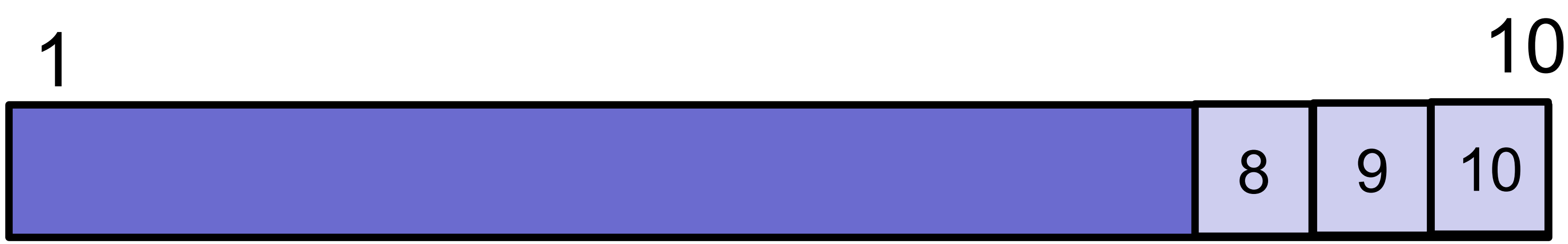
Iteration 2:



BubbleSort Analysis

Loop invariant:

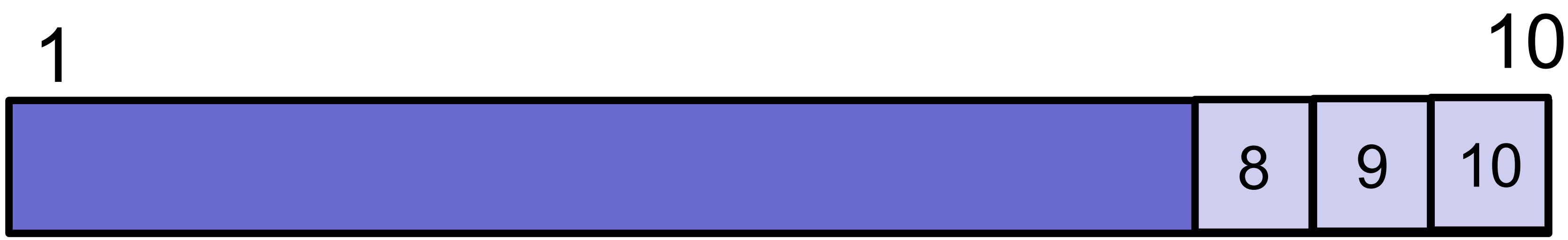
At the end of iteration j: ???



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.

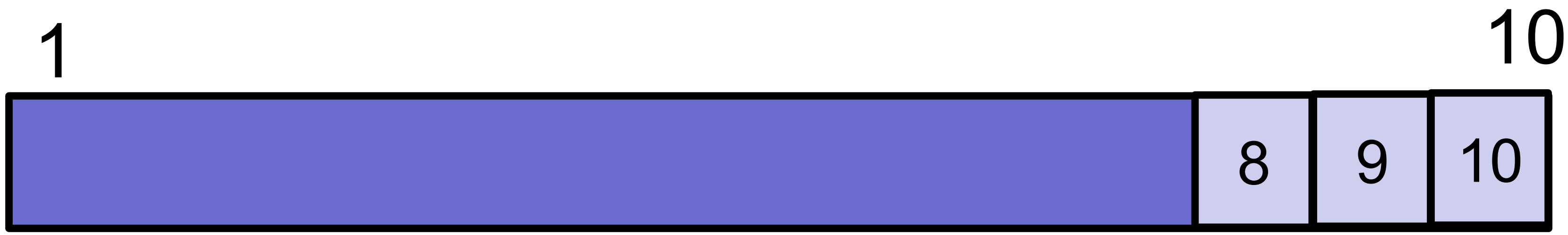


BubbleSort Analysis

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

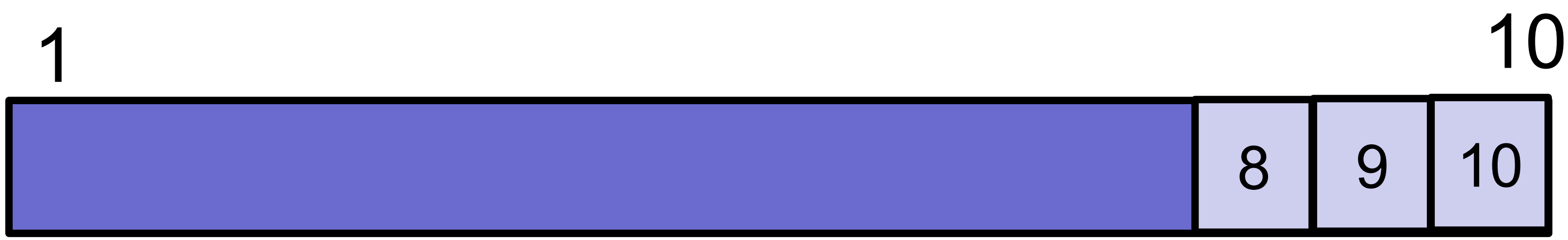


BubbleSort Analysis

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

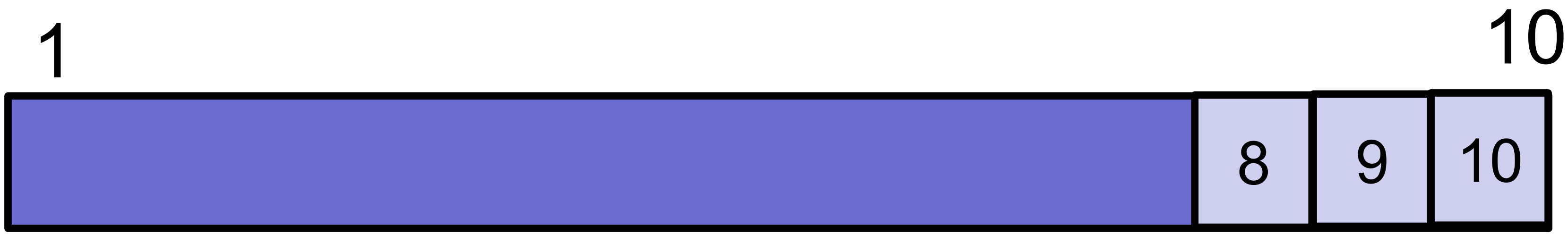


BubbleSort Analysis

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

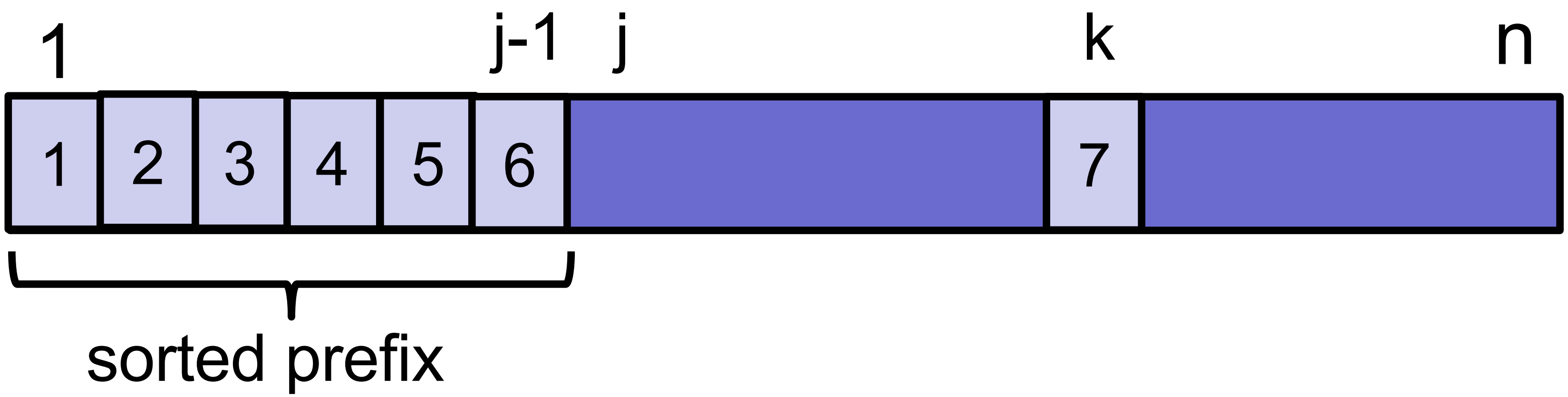
SelectionSort

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)

ARCHIPELAGO
is open

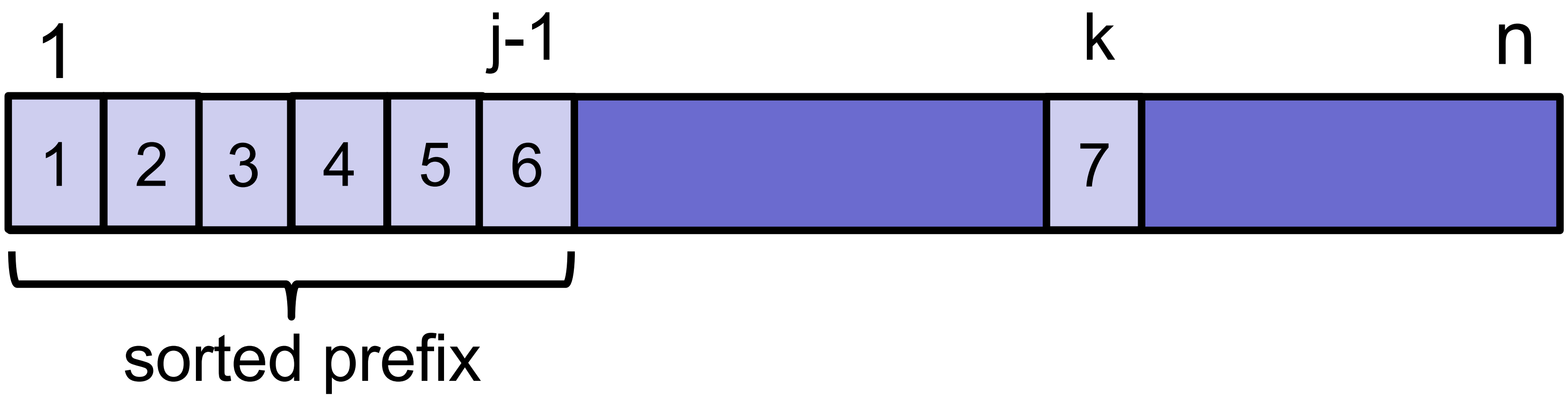
SelectionSort

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: $(n - j)$

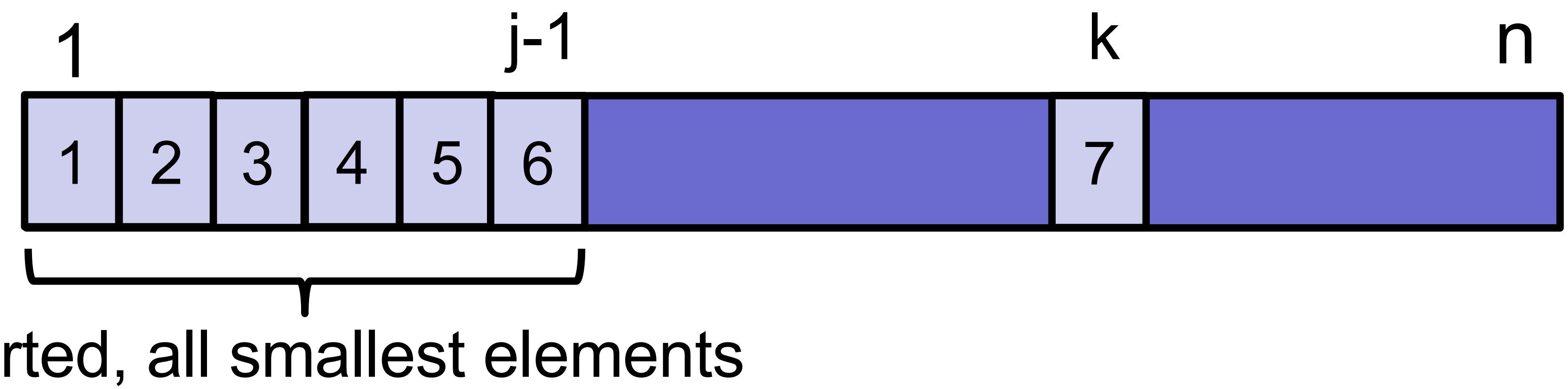
SelectionSort(A, n)

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

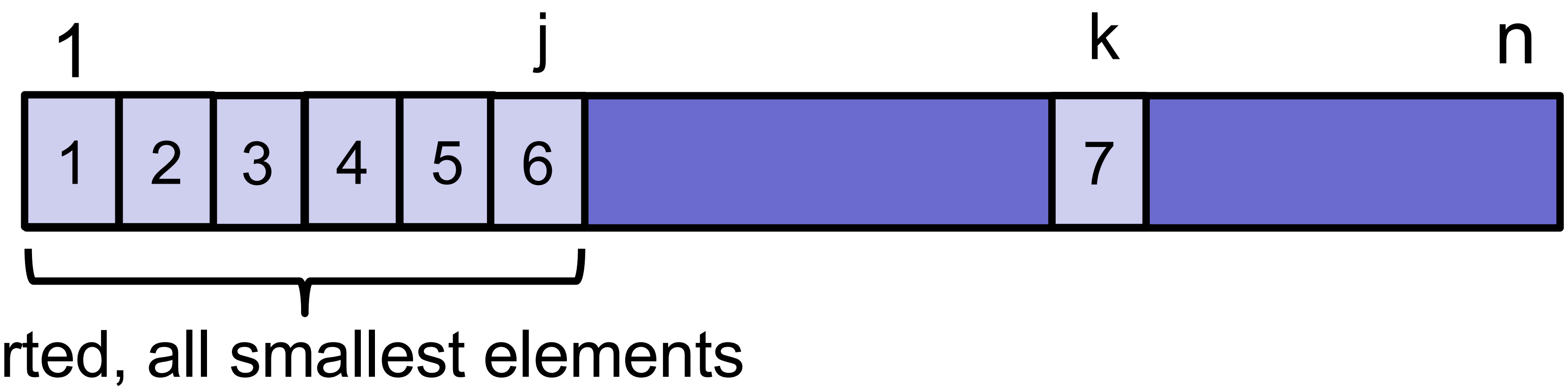
Time: $(n - j)$

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

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

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

SelectionSort

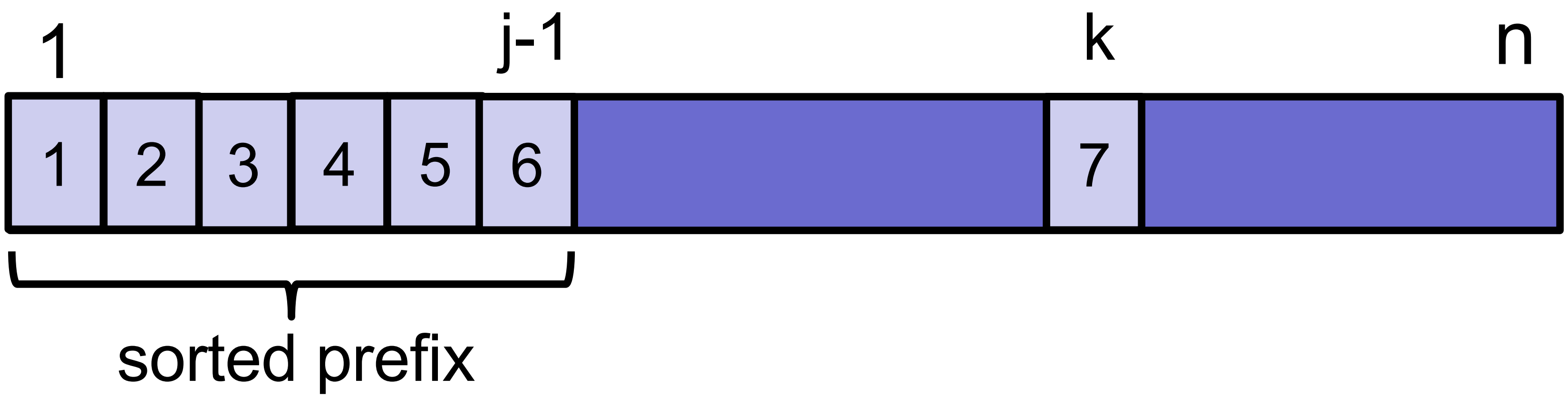
SelectionSort(A, n)

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. $O(2^n)$

ARCHIPELAGO

is open

SelectionSort

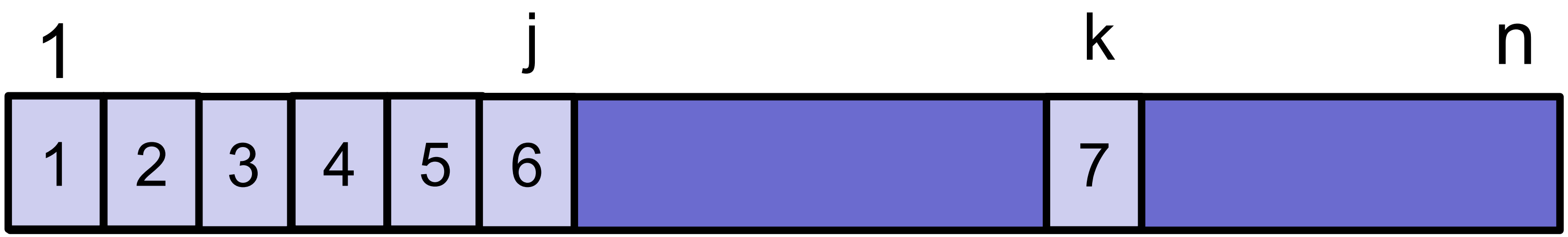
SelectionSort(A, n)

for j ← 1 **to** n-1**:**

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

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

Running time: $O(n^2)$ and $\Omega(n^2)$



SelectionSort

ARCHIPELAGO

is open

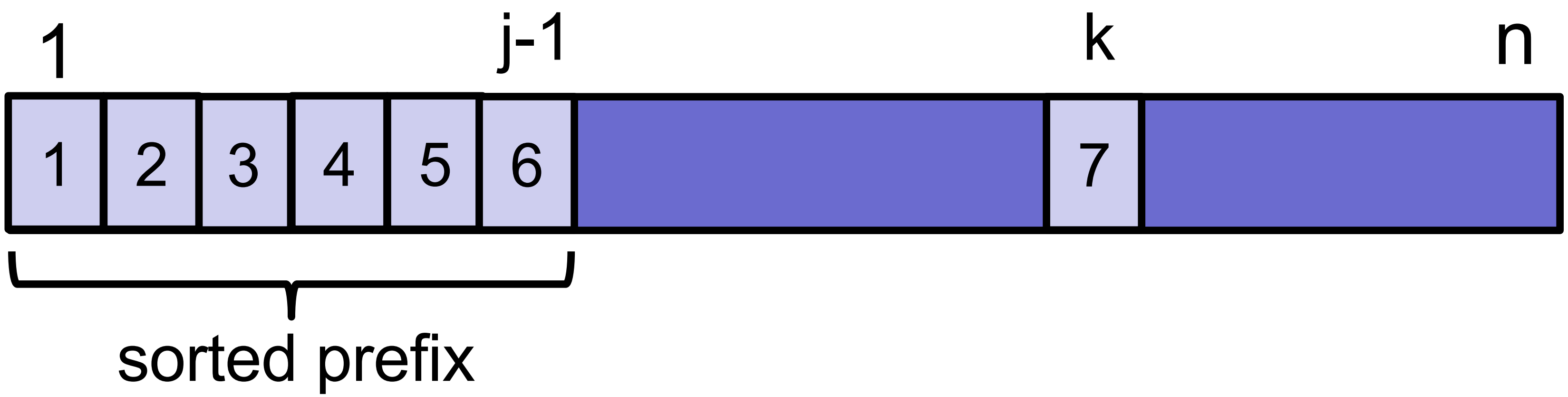
SelectionSort(A, n)

for j ← 1 **to** n-1**:**

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

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

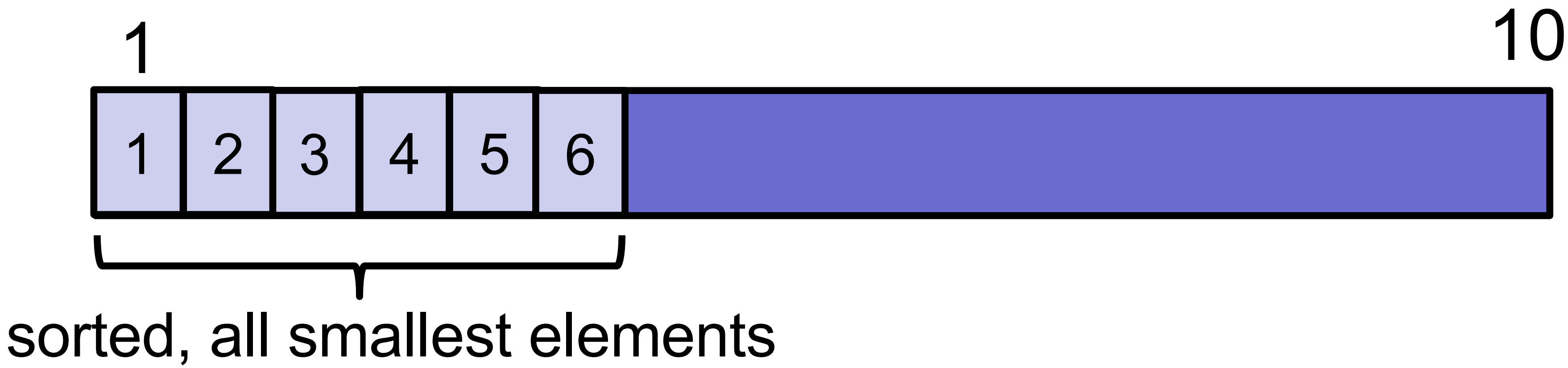
What is a good loop invariant for SelectionSort?



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.



Today: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

Insertion Sort

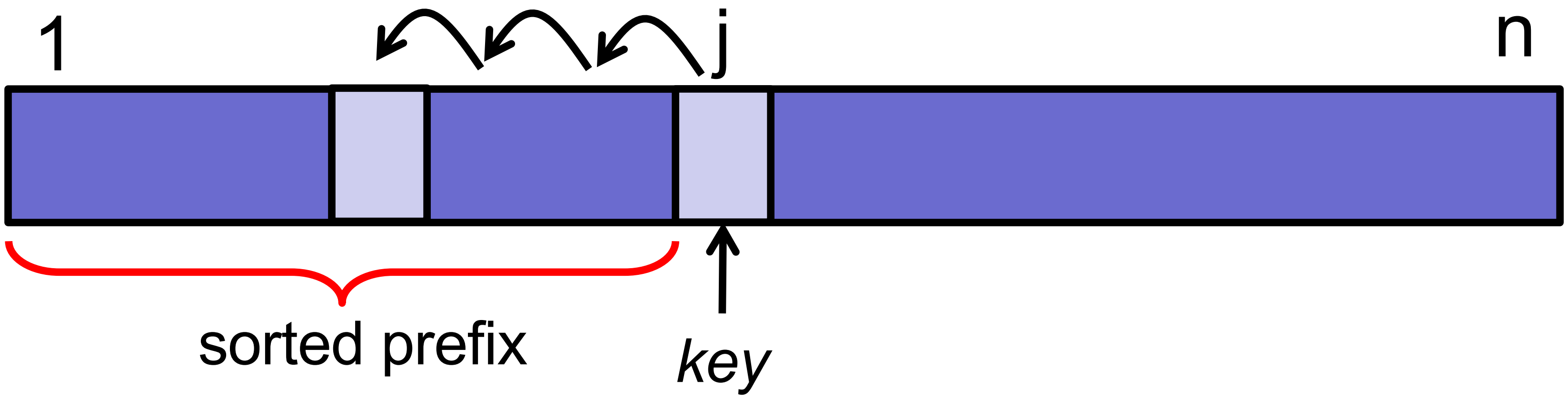
InsertionSort(A, n)

for j ← 2 **to** n

key ← A[j]

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

Illustration:



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

Insertion Sort

Example:

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

Insertion Sort

Example:

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

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. $O(2^n)$

ARCHIPELAGO
is open

Insertion Sort

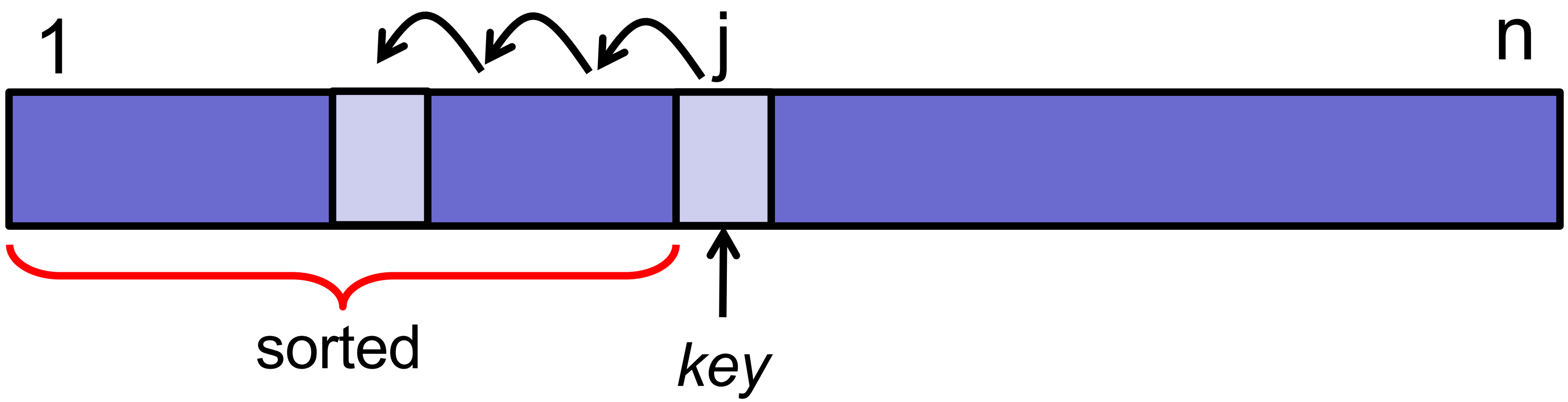
What is the max distance that the key needs to be moved?

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

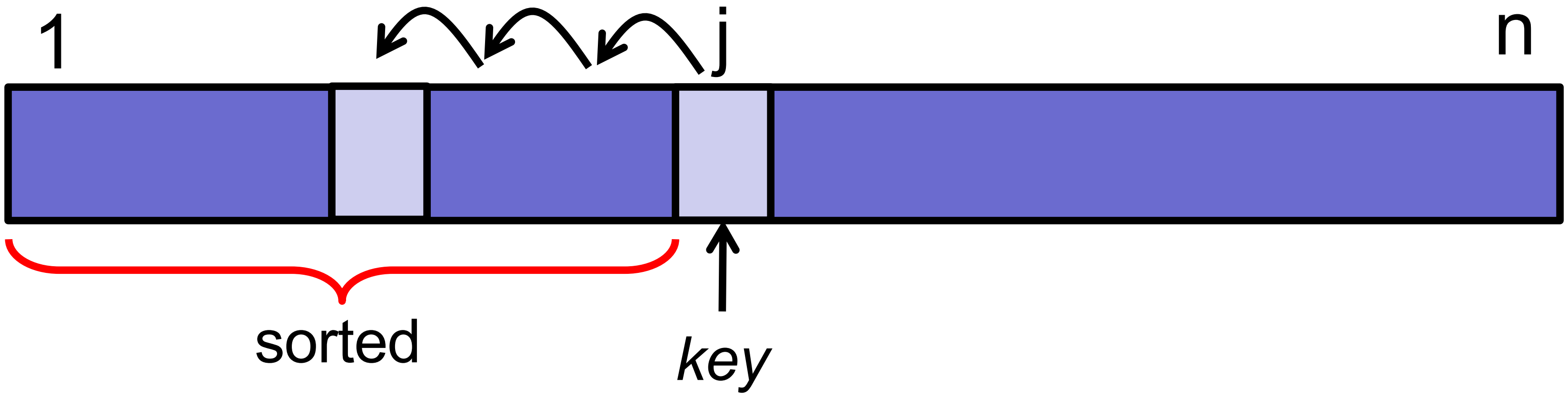
Insertion-Sort(A, n)

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

ARCHIPELAGO

is open

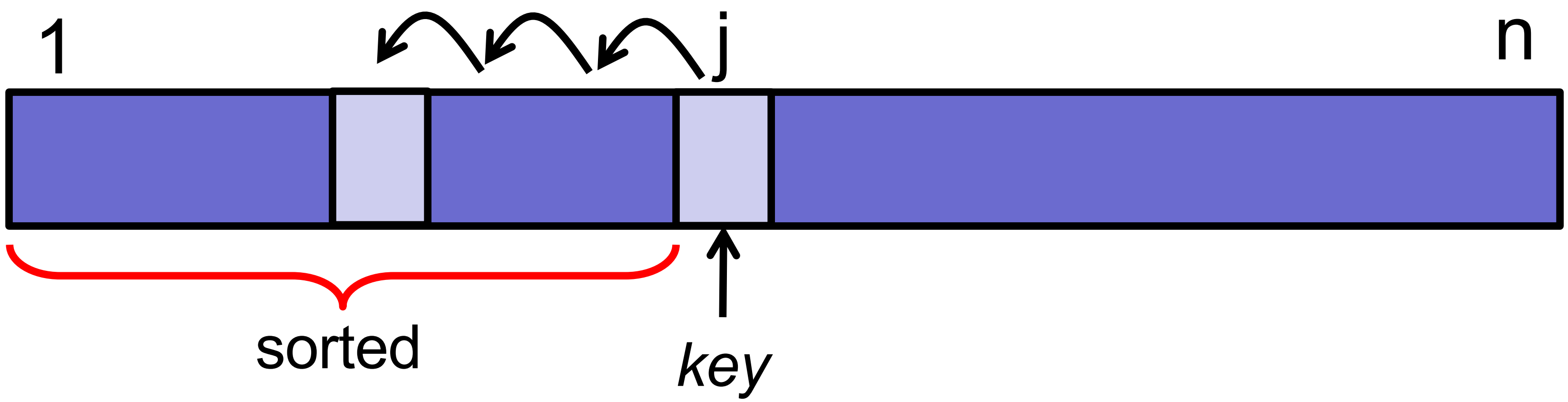
Insertion-Sort(A, n)

for j ← 2 **to** n

key ← A[j]

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

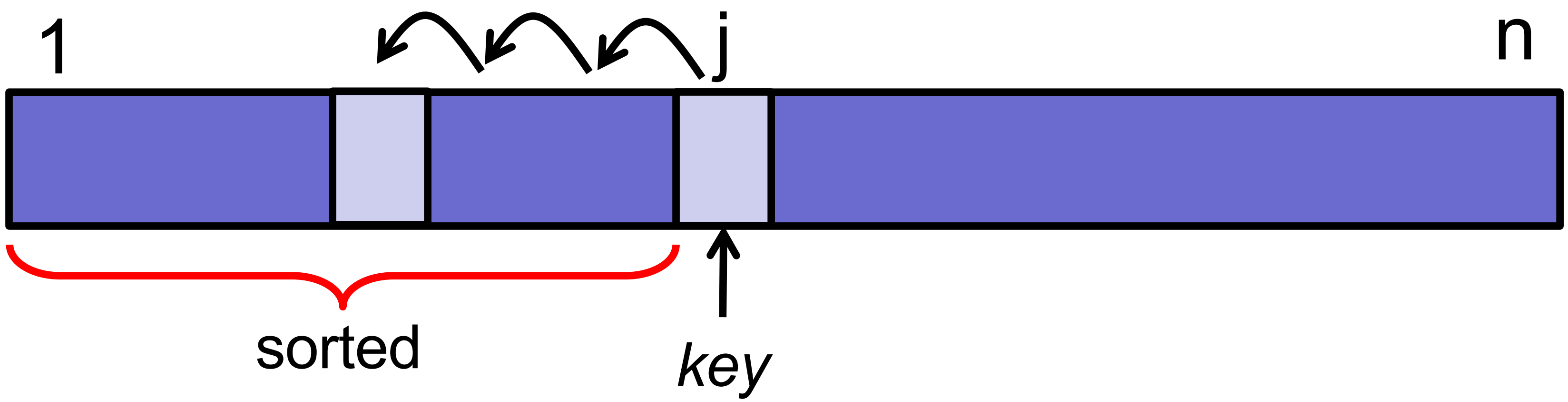
What is a good loop invariant for InsertionSort?



Insertion Sort

Loop invariant:

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



Insertion Sort

ARCHIPELAGO is open

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

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

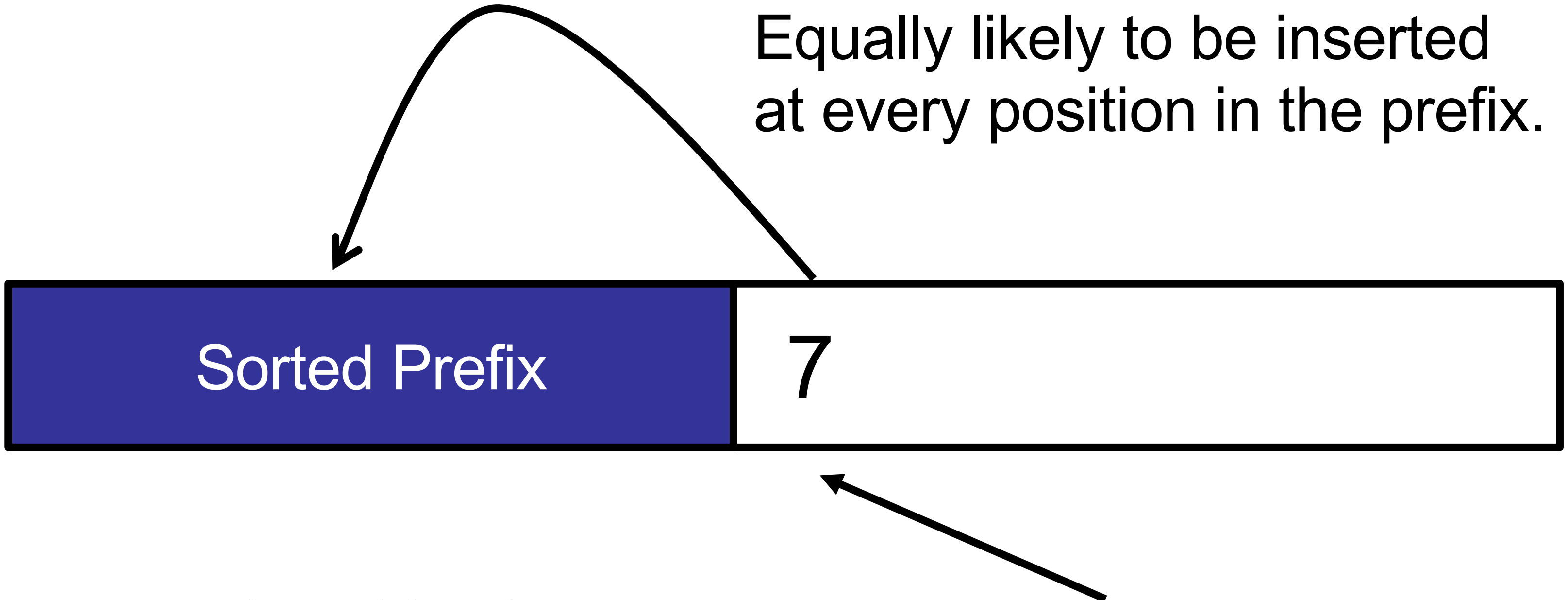
- In expectation, still $\theta(\mathbf{n^2})$

Insertion Sort Analysis

Average-case analysis:

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

Equally likely to be inserted at every position in the prefix.



Thus, expected position is halfway back, i.e., slot j/2.

Next item to insert

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!

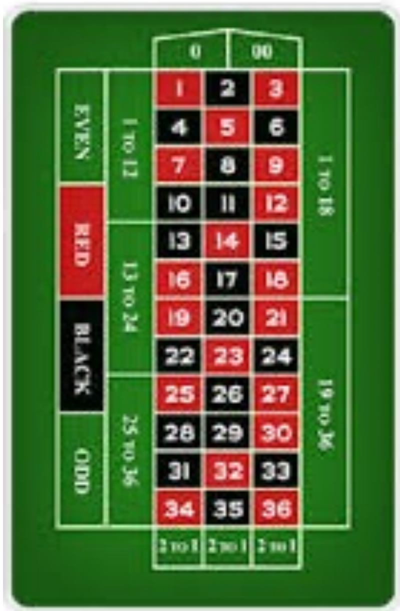
ARCHIPELAGO

is open

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?

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

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.

Another daily challenge:

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)

- 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.

(Exercise: Come up with some examples of where this is obviously the wrong way to measure space!)

(I don’t want to get into memory allocators, garbage collection, stack frames, etc.)

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.

Which are stable?

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

Not stable:
Swap elements while ignoring in between.

SelectionSort

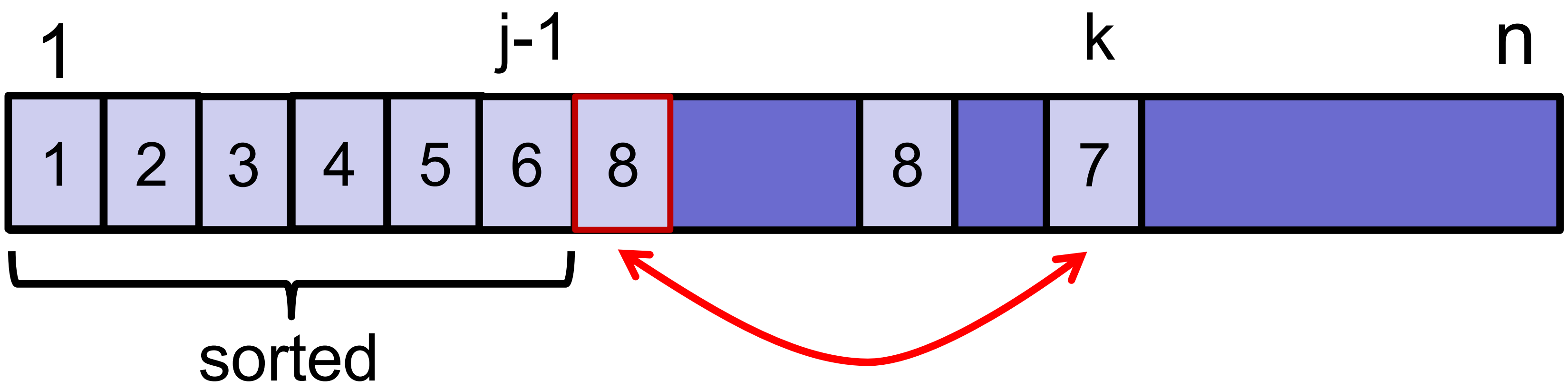
SelectionSort(A, n)

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

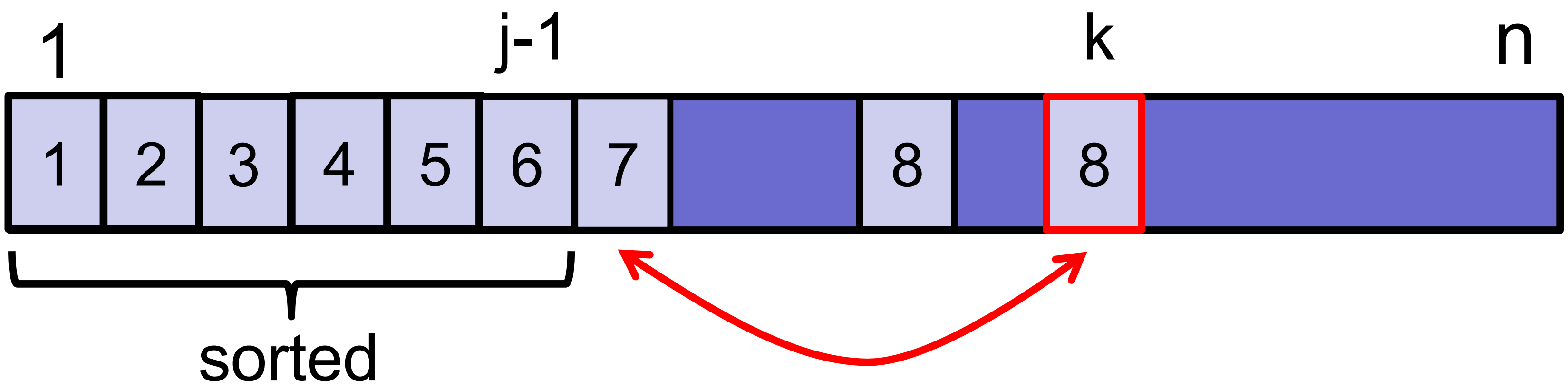
SelectionSort(A, n)

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



Which are stable?

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

Stable:
Do not swap identical elements.

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