CS2040S
Data Structures and Algorithms

Welcome!

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 / rescheduling.
- Do prepare in advance.
- Do have questions.
- Do take advantage of tutorial to get to know your tutor and other students in your class

Admin: Tutorial/Recitations

Please do not modify your tutorial/recitation sections directly on ModReg!

We will undo all such changes.

All changes much go through us.

We are trying hard to keep sections well-balanced.

Puzzle of the Week (Contest)

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!

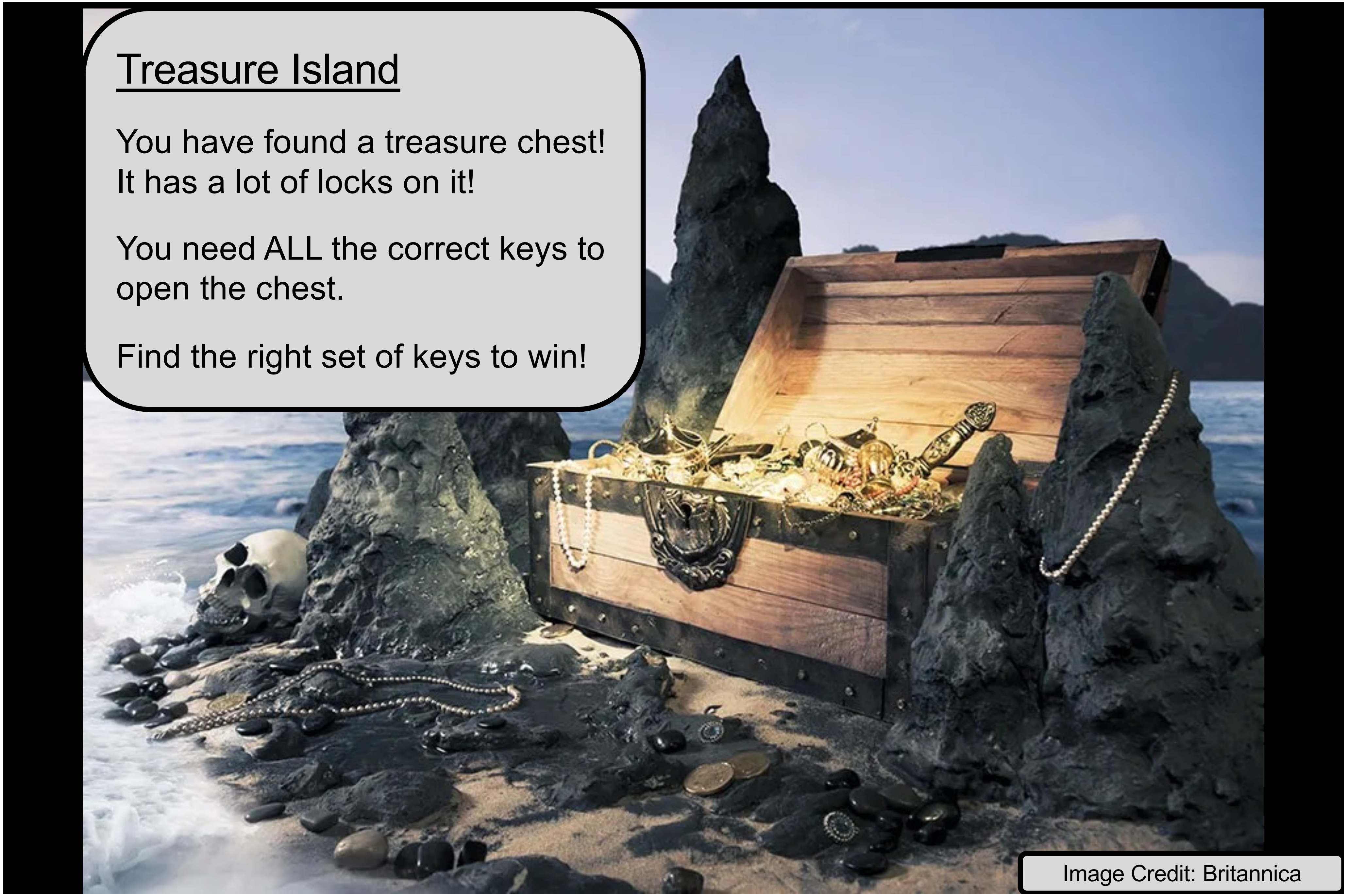


Image Credit: Britannica

Puzzle of the Week (Contest)

Treasure Island

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify *exactly* the good keys.

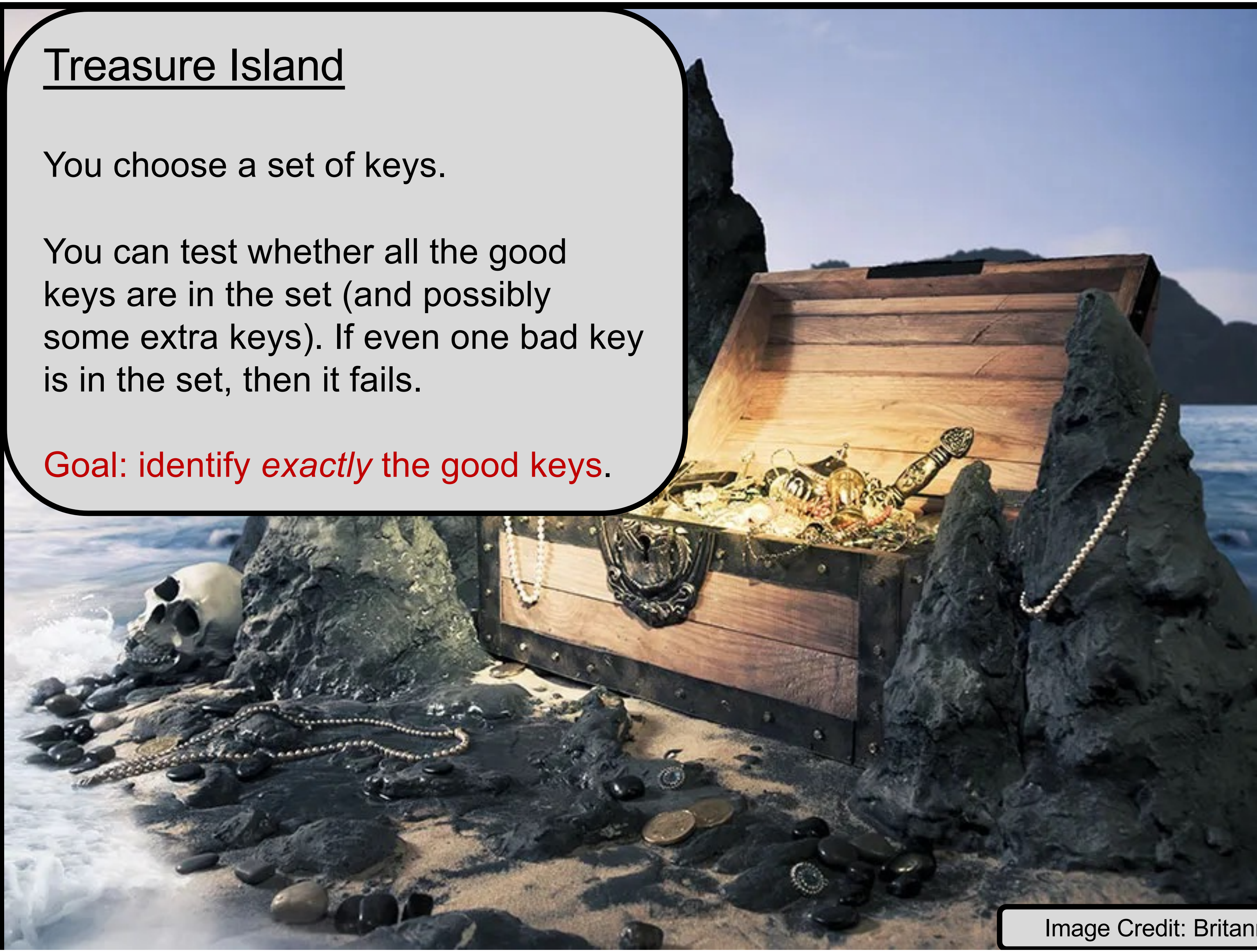


Image Credit: Britannica

Puzzle of the Week (Contest)

Treasure Island

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify exactly the good keys.

Puzzle of the Week (Contest)

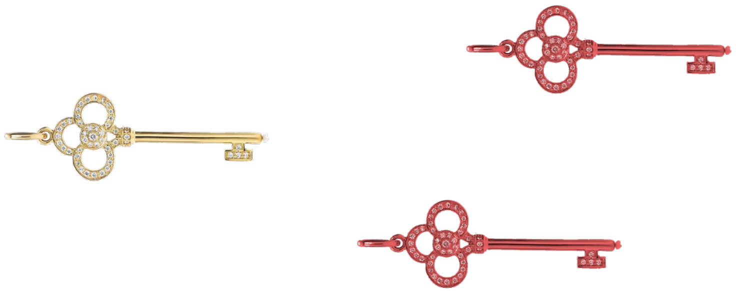
Treasure Island

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify exactly the good keys.

FAIL



Puzzle of the Week (Contest)

Treasure Island

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify exactly the good keys.

SUCCEED

Puzzle of the Week (Contest)

<u>Treasure Island</u>

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify exactly the good keys.

SUCCEED and OPEN

Puzzle of the Week (Contest)

Treasure Island

Two goals:

- Minimize number of tries.
- Minimize total number of keys tried.

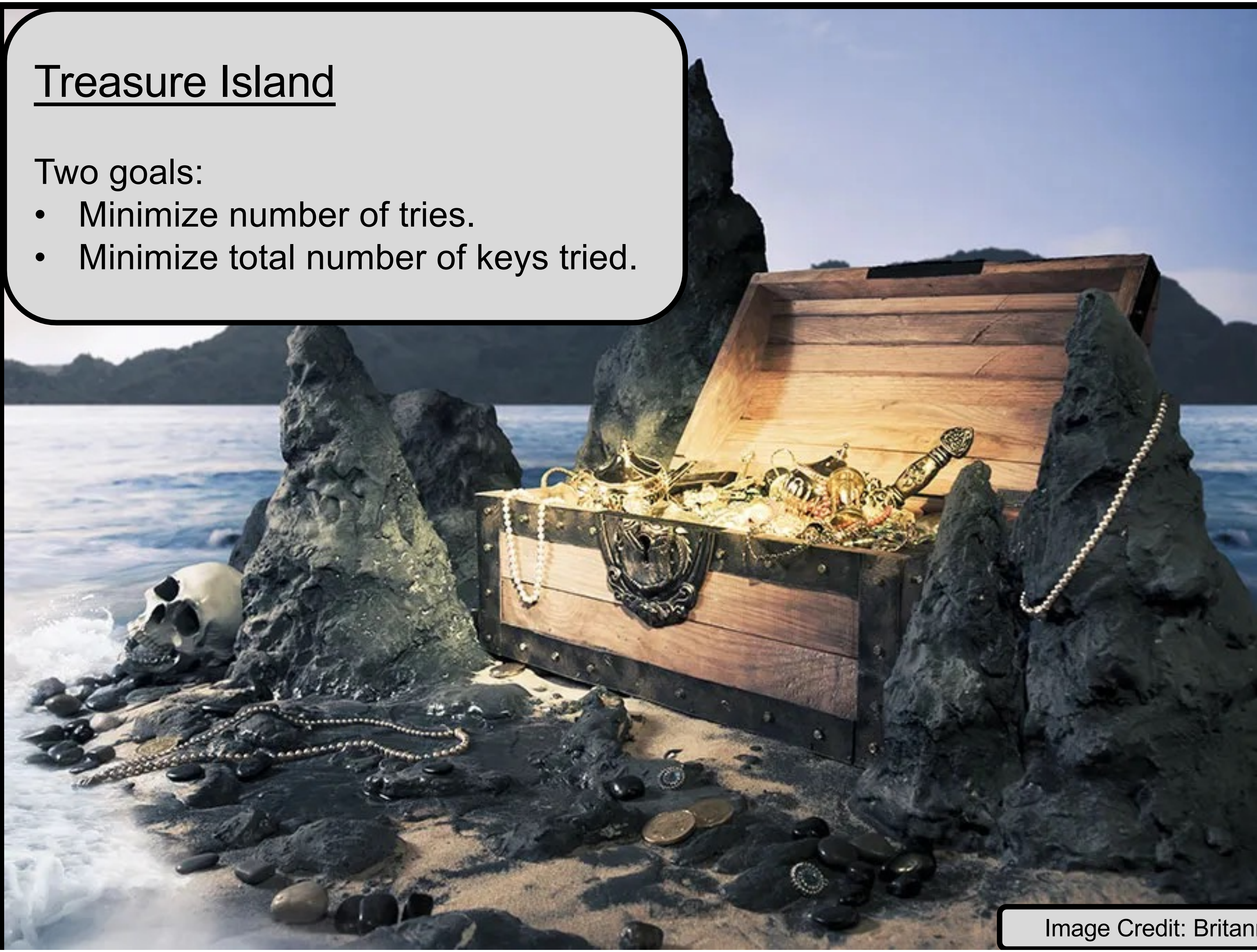
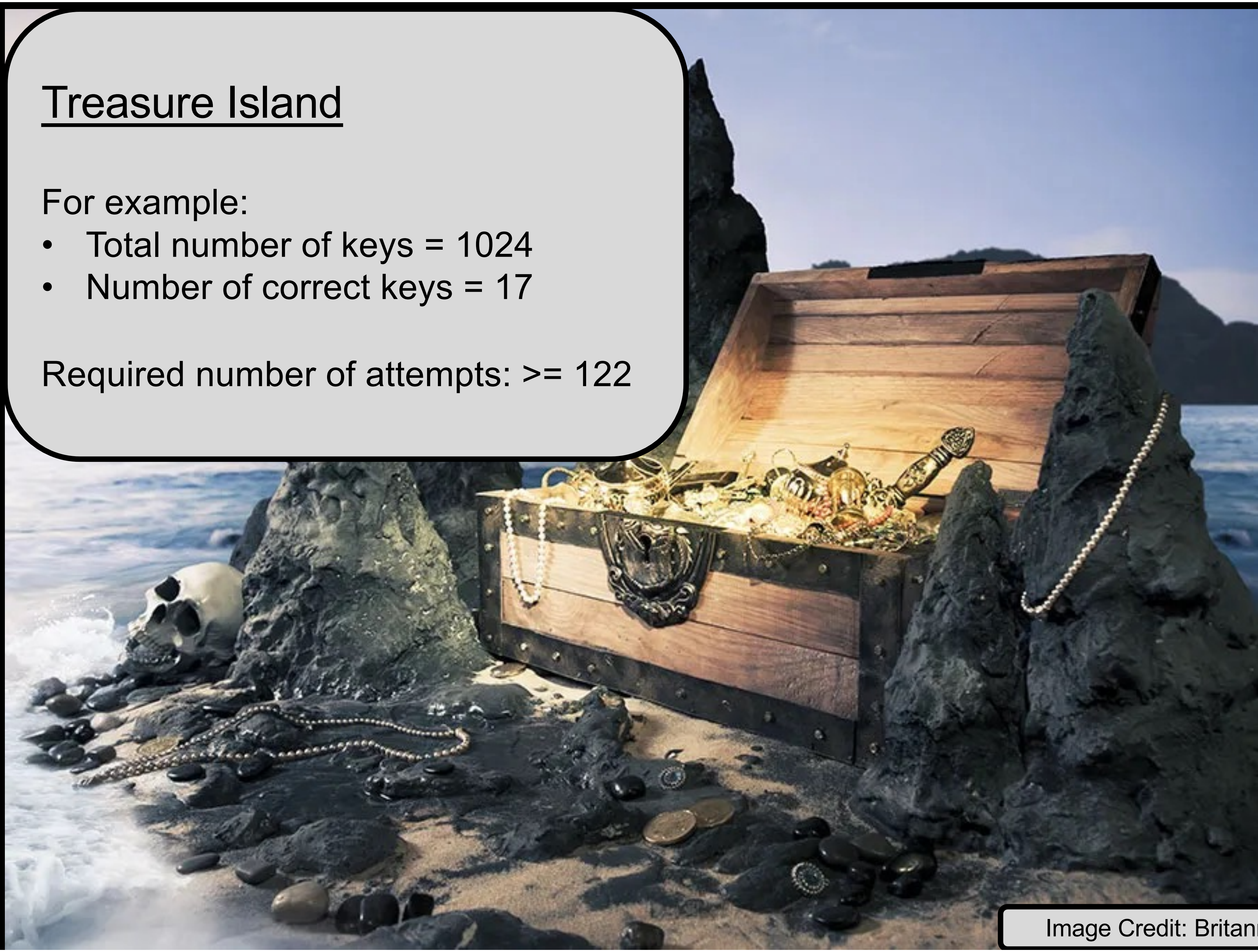


Image Credit: Britannica

Puzzle of the Week (Contest)



Treasure Island

For example:

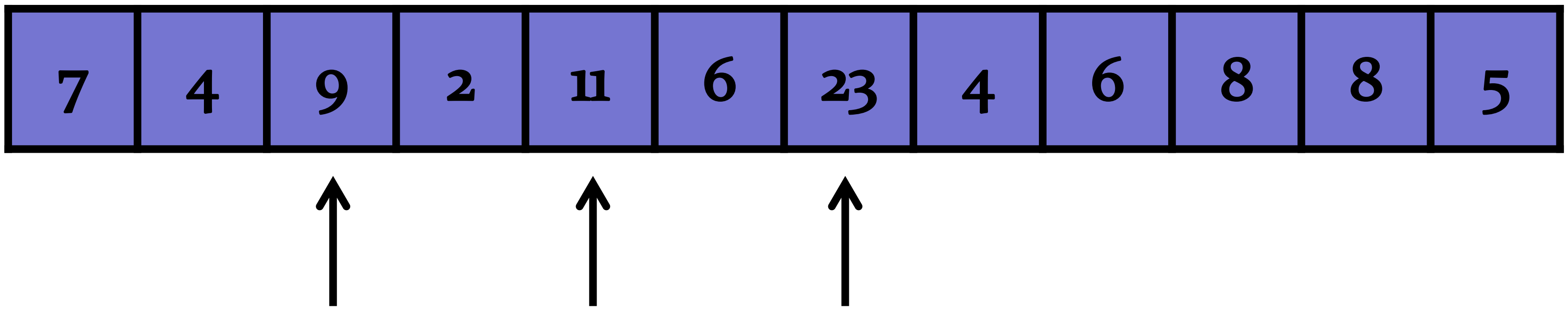
- Total number of keys = 1024
- Number of correct keys = 17

Required number of attempts: >= 122

Image Credit: Britannica

Steep Peaks

Input: Some array A[0..n-1]



Output: a local maximum in A

`A[i-1] < A[i]` **and** `A[i+1] < A[i]`

Assume that

`A[-1] = A[n] = -MAX_INT`

Steep Peaks

Input: Some array A[0..n-1]

7	4	9	2	11	6	23	4	6	8	8	5

Output: a local maximum in A

$$A[i-1] < A[i] \textbf{ and } A[i+1] < A[i]$$

Can we find *steep* peaks efficiently (in O(log n) time) using the same approach?

Steep Peaks

Problematic example:

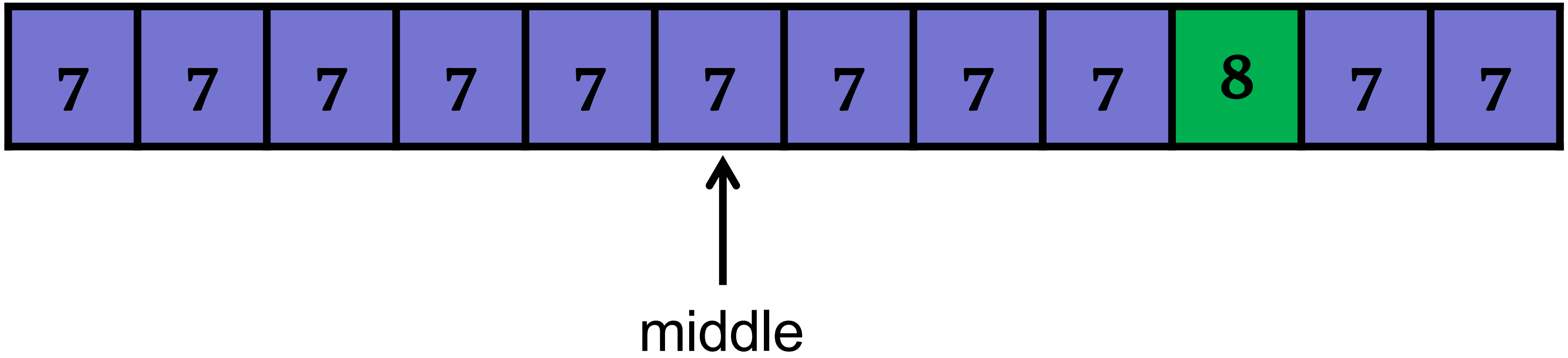
7	7	7	7	7	7	7	7	7	8	7	7

Inuitively:

There are n different positions to search for the steep peak, and no hints as to where it might be found!

Steep Peaks

Problematic example:

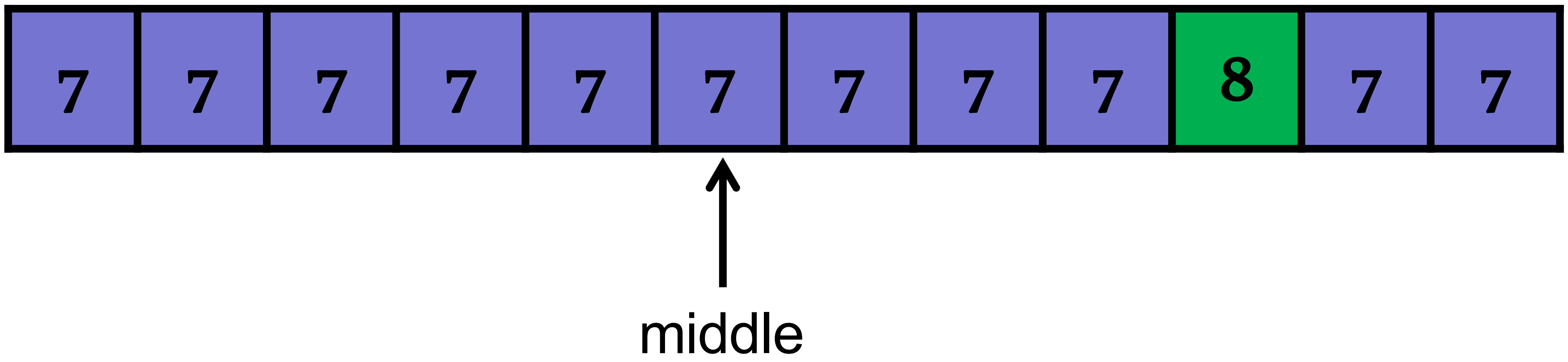


Which side does the algorithm recurse on?

Steep Peaks

ARCHIPELAGO
is open

Problematic example:



What happens if you recurse on both sides?

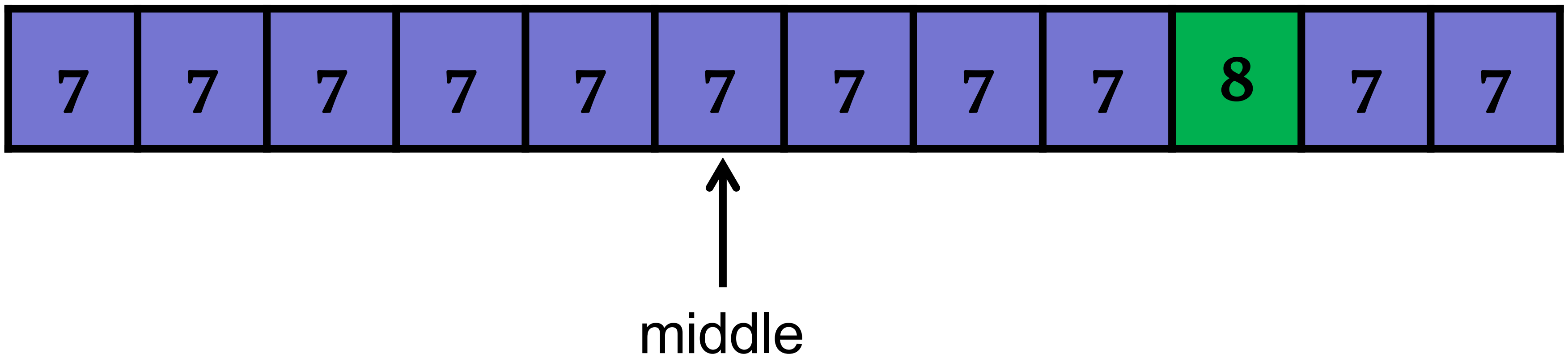
...

if A[n/2-1] == A[n/2] == A[n/2+1] **then**

Recurse on left & right sides

Steep Peaks

Problematic example:



What happens if you recurse on both sides?

Recurrence: $T(n) = 2T(n/2) + O(1)$

Steep Peak Finding

Unrolling the recurrence:

Rule:
$T(X) = 2T(X/2) + 1$

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

$$= 2(2T(n/4) + 1) + 1 = 4T(n/4) + 2 + 1$$

$$= 8T(n/8) + 4 + 2 + 1$$

$$= 16T(n/16) + 8 + 4 + 2 + 1$$

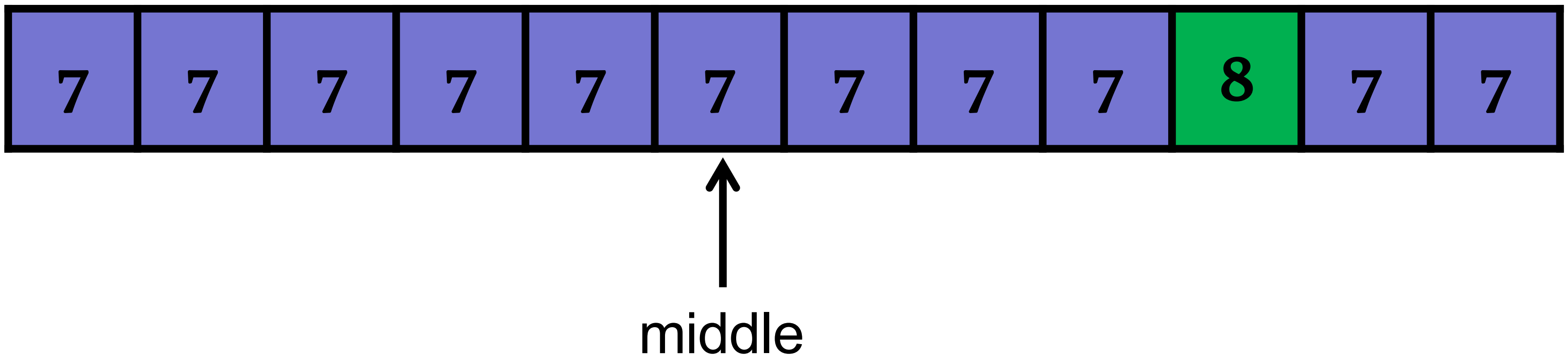
$$\cdots$$

$$= nT(1) + n/2 + n/4 + n/8 + \ldots + 1 =$$

$$= n + n/2 + n/4 + n/8 + \ldots + 1 = \theta(n)$$

Steep Peaks

Problematic example:



What happens if you recurse on both sides?

Recurrence: $T(n) = 2T(n/2) + O(1) = O(n)$

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

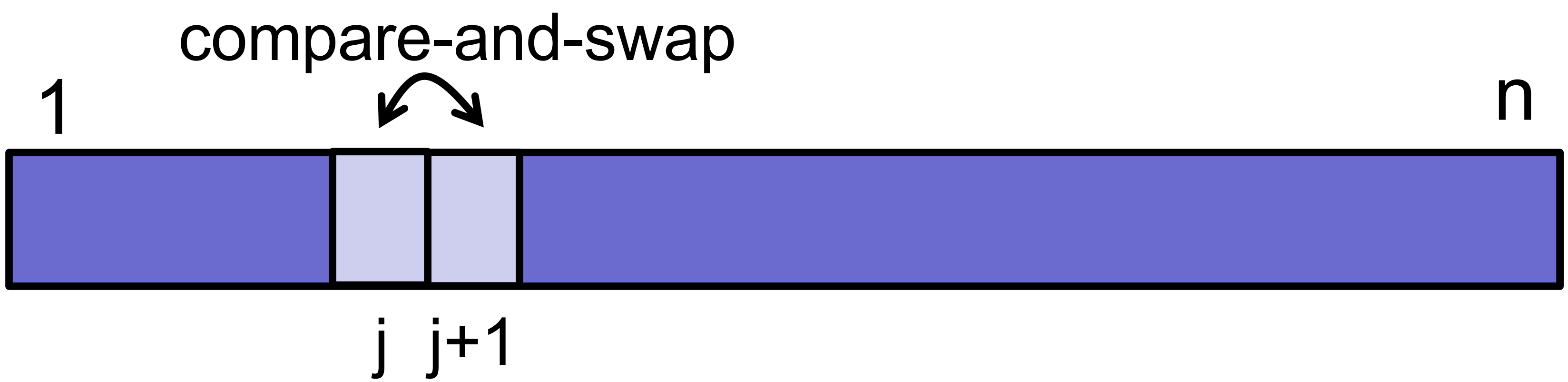
BubbleSort

BubbleSort(A, n)

repeat n **times:**

for j $\leftarrow$ 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:

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

BubbleSort

Pass 3:

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

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**

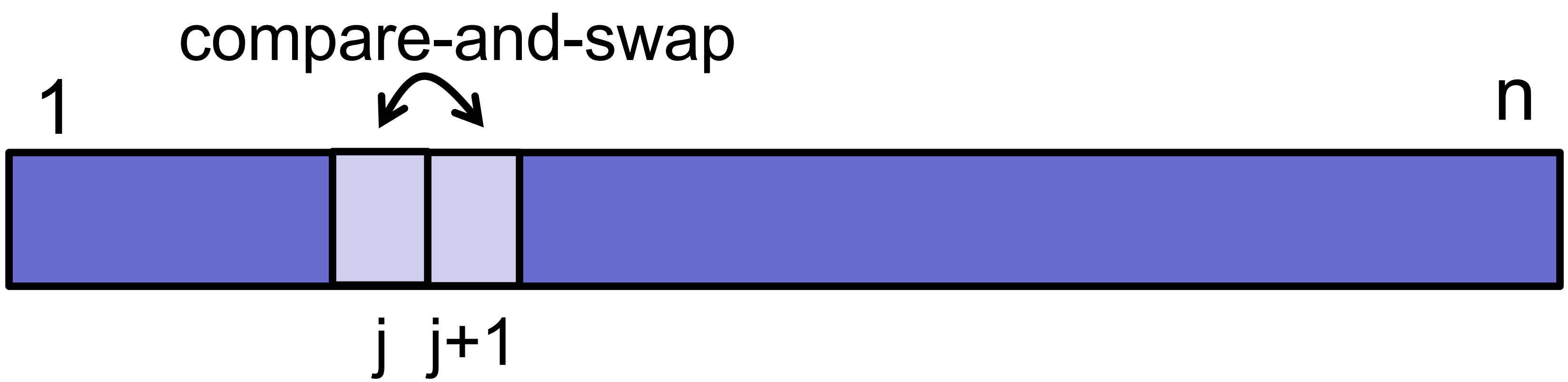
BubbleSort

BubbleSort(A, n)

repeat n **times:**

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

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



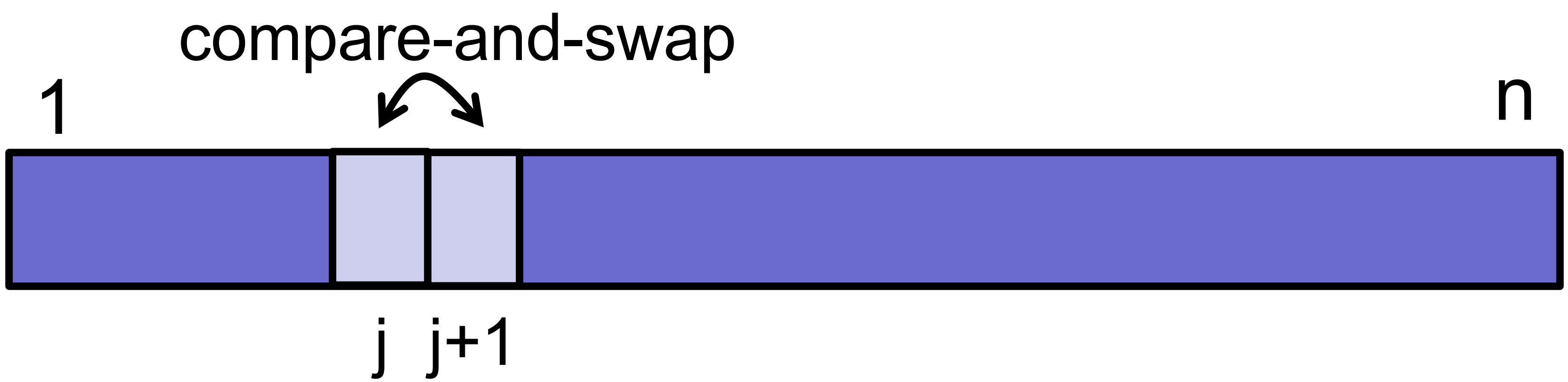
BubbleSort

BubbleSort(A, n)

repeat (until no swaps) **:**

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

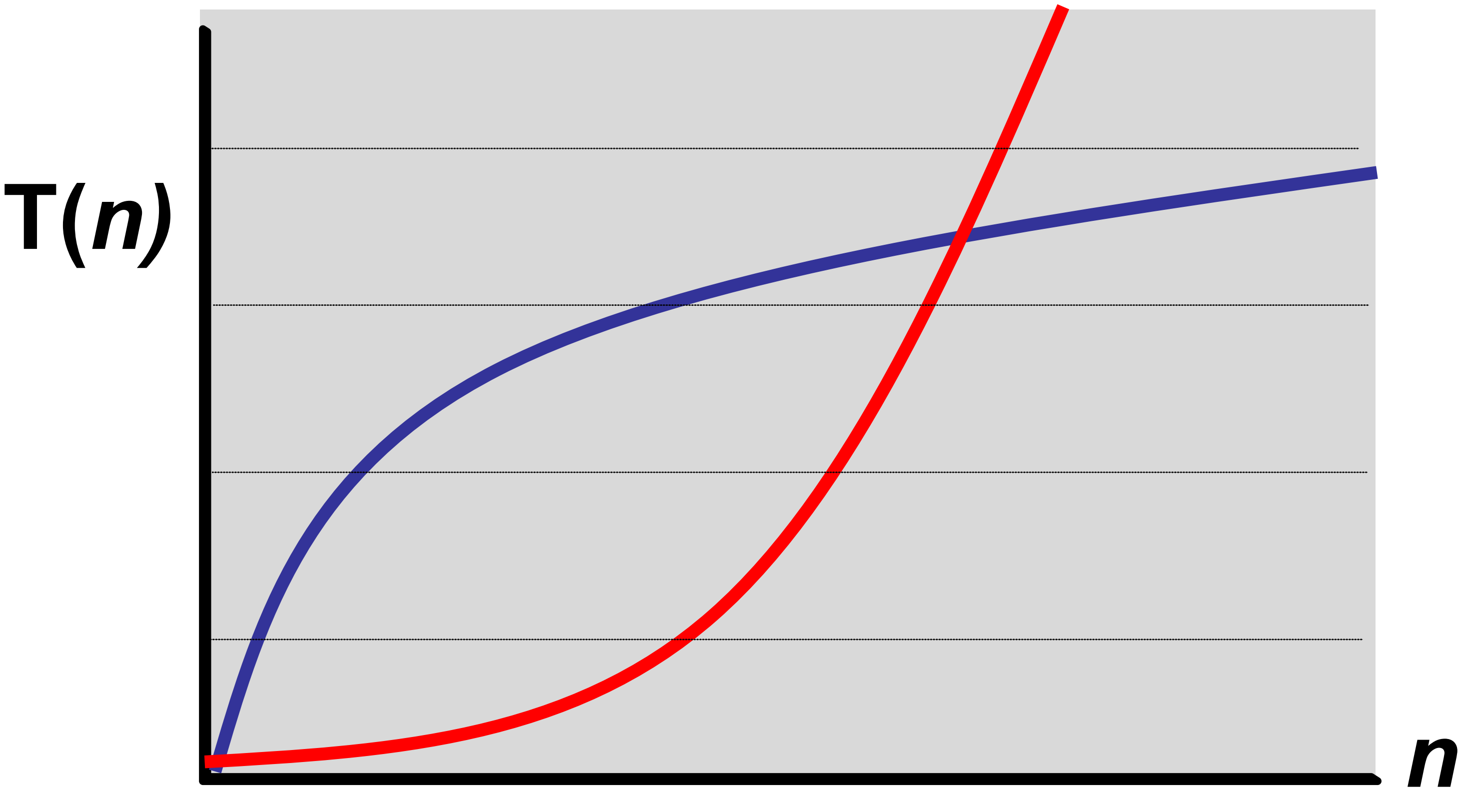
if A[j] > A[j+1] **then** swap(A[j], A[j+1])



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.

Worst-case:

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

- Max running time over all possible inputs.

BubbleSort Analysis

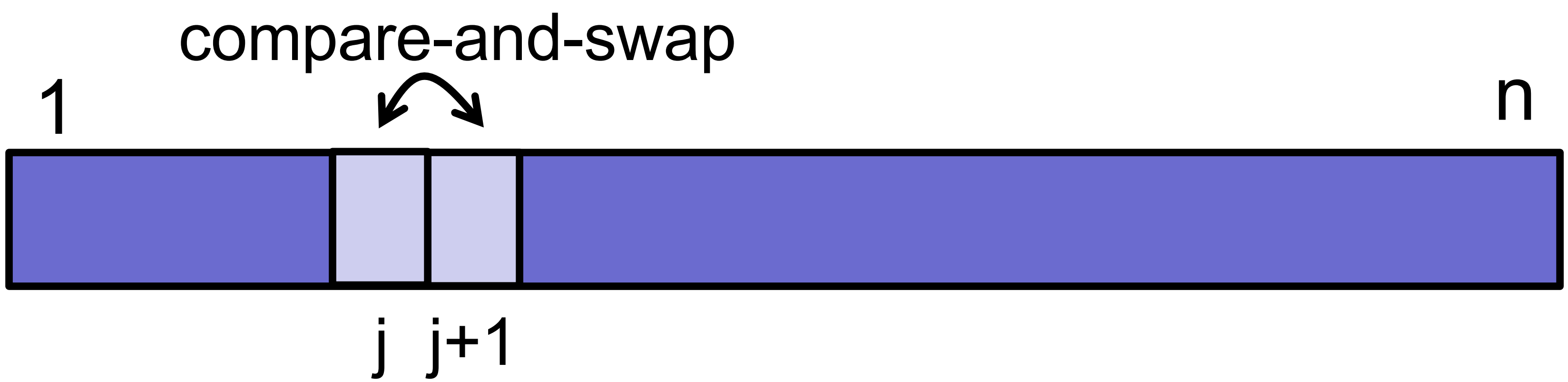
How many iterations do we need?

BubbleSort(A, n)

repeat (until no swaps) **:**

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

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



BubbleSort Analysis

ARCHIPELAGO

is open

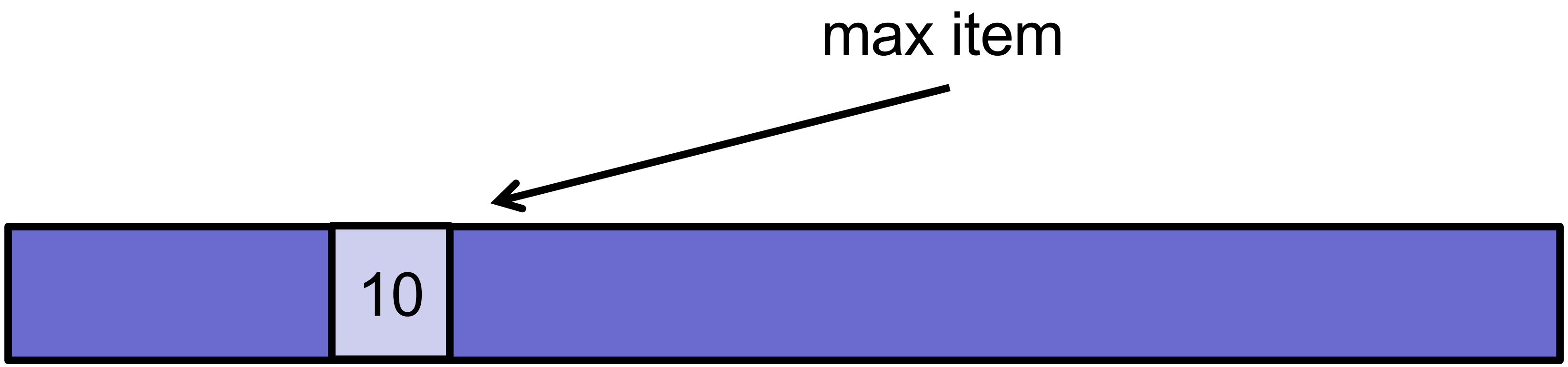
BubbleSort(A, n)

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

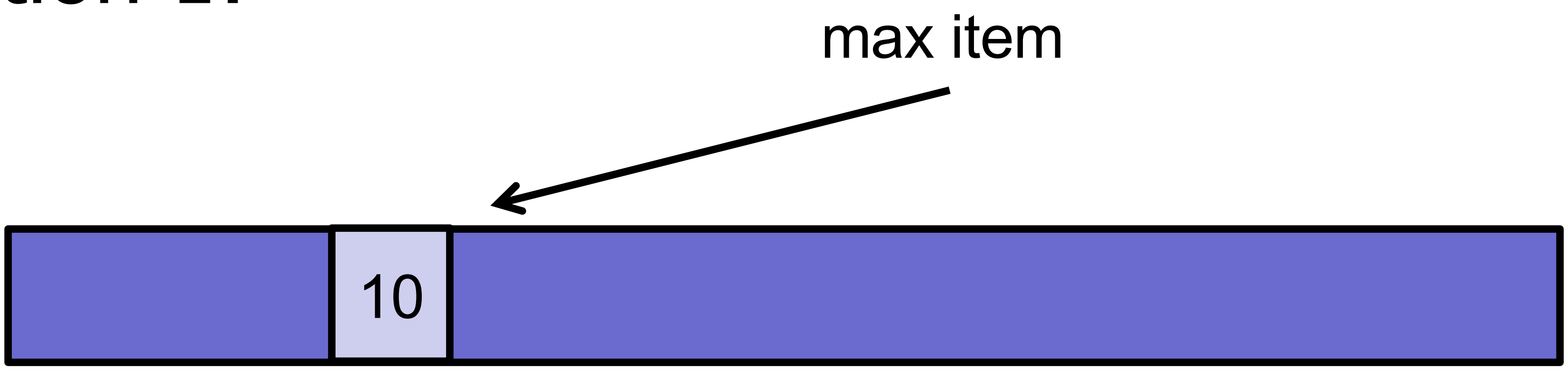
BubbleSort(A, n)

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

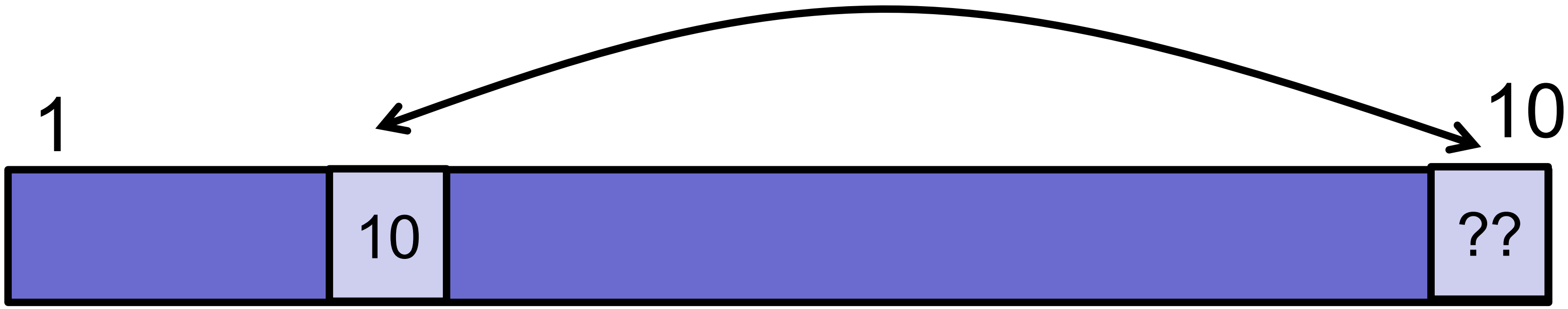
BubbleSort(A, n)

    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

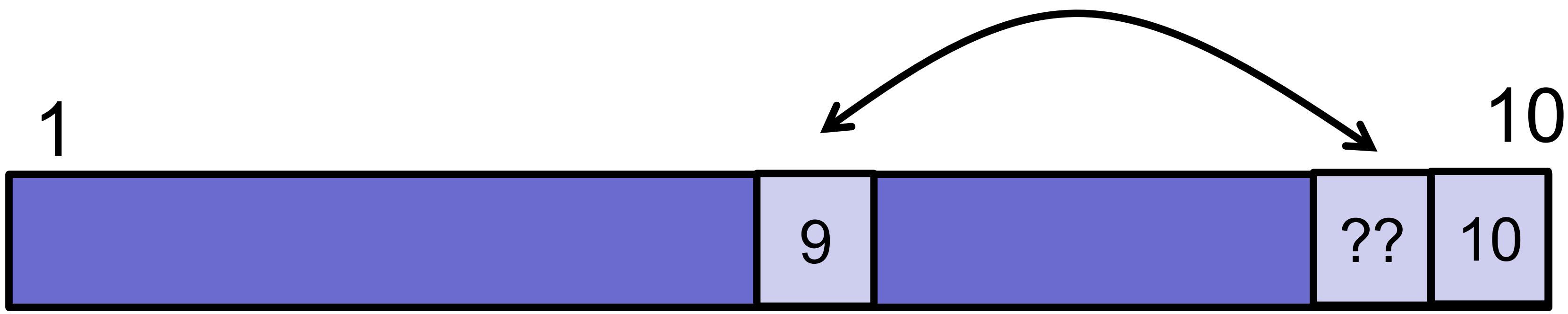
BubbleSort(A, n)

repeat (until no swaps) **:**

for j ← 1 **to** n-1

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

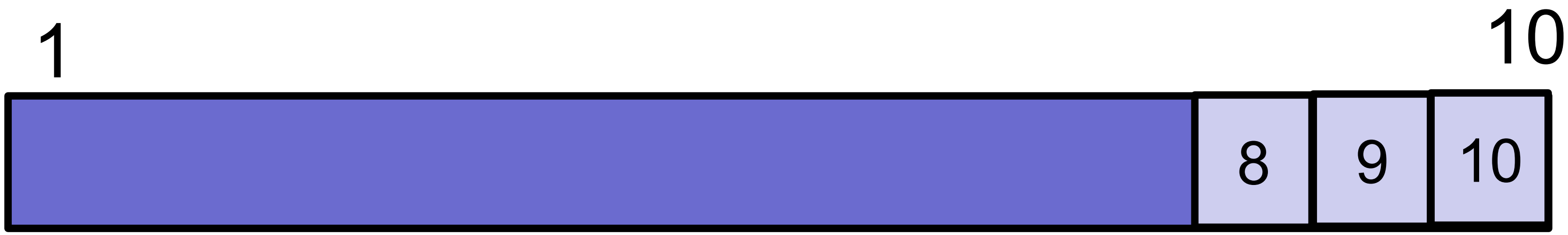
Iteration 2:



BubbleSort Analysis

Loop invariant:

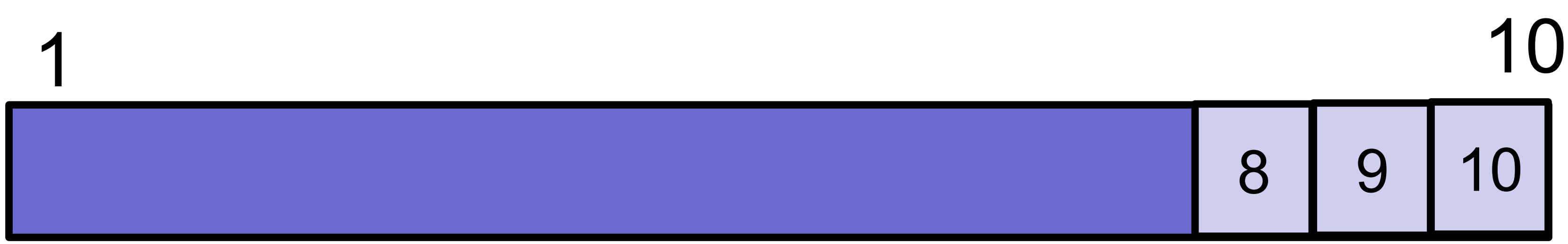
At the end of iteration j: ???



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.

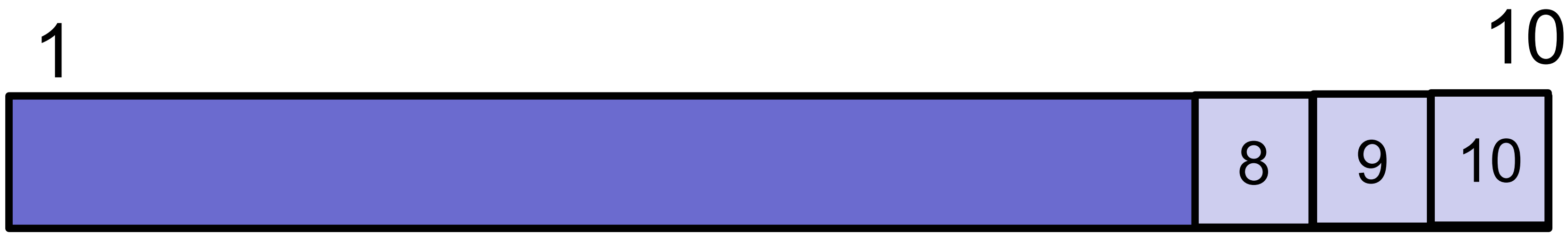


BubbleSort Analysis

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

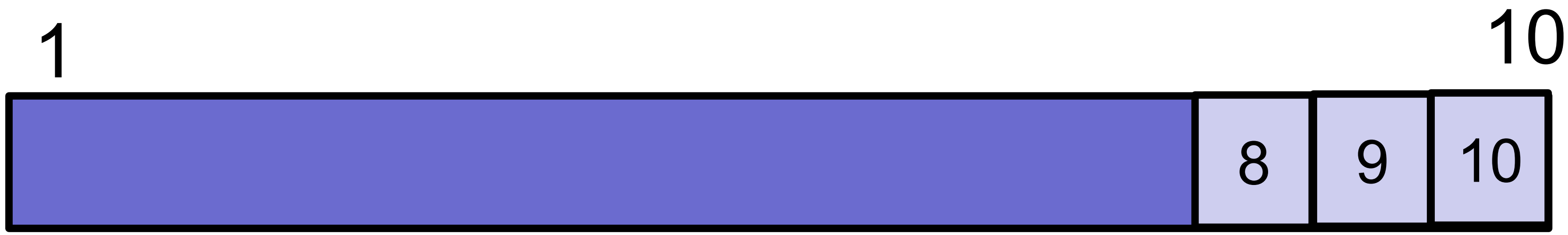


BubbleSort Analysis

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

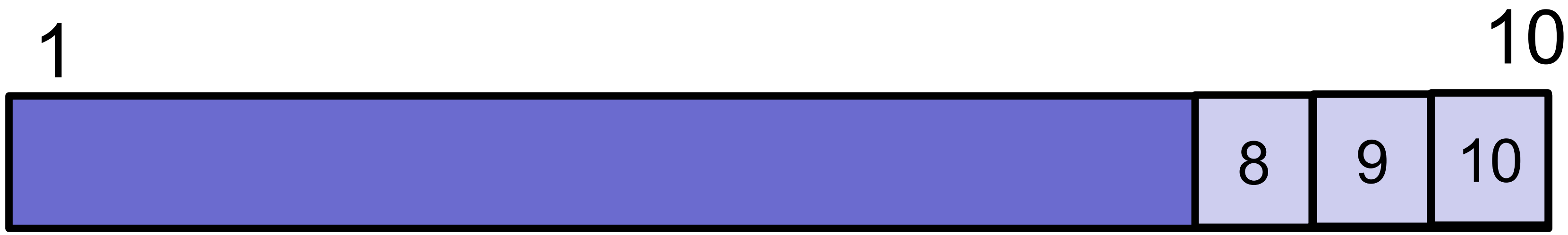


BubbleSort Analysis

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

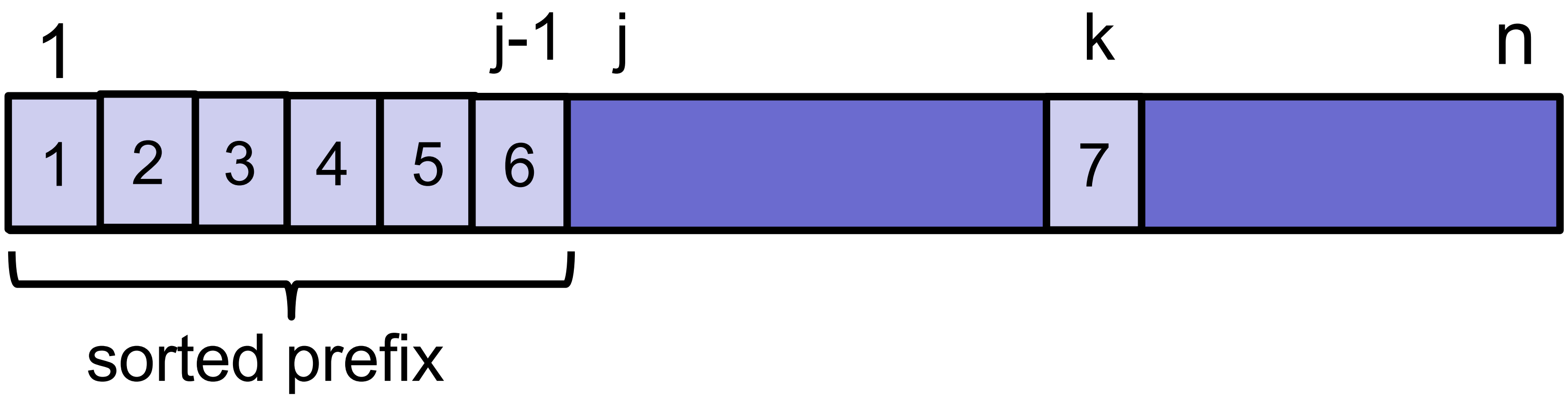
SelectionSort

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

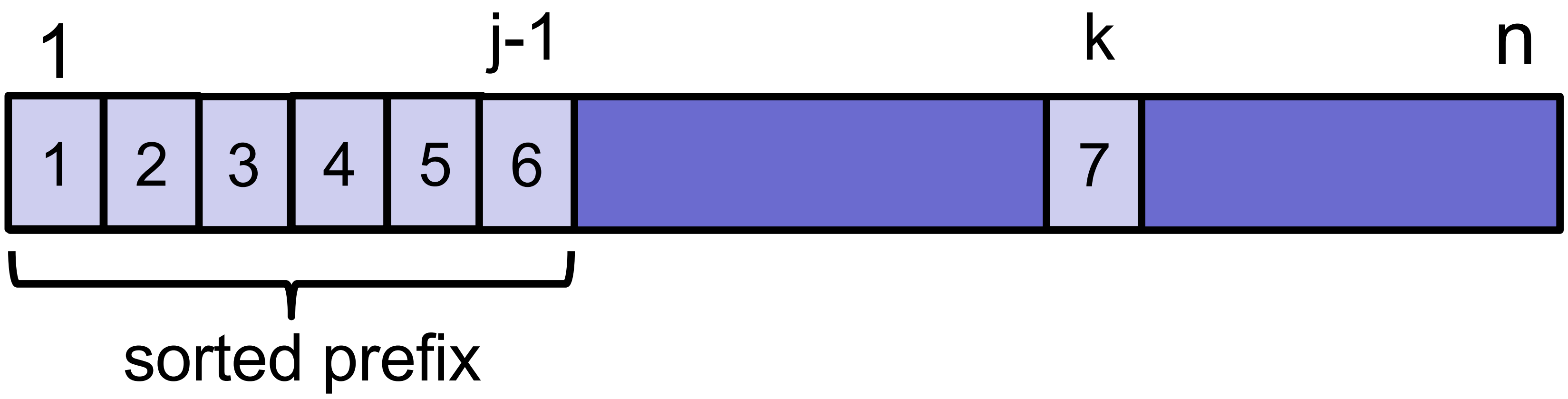
SelectionSort

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: $(n - j)$

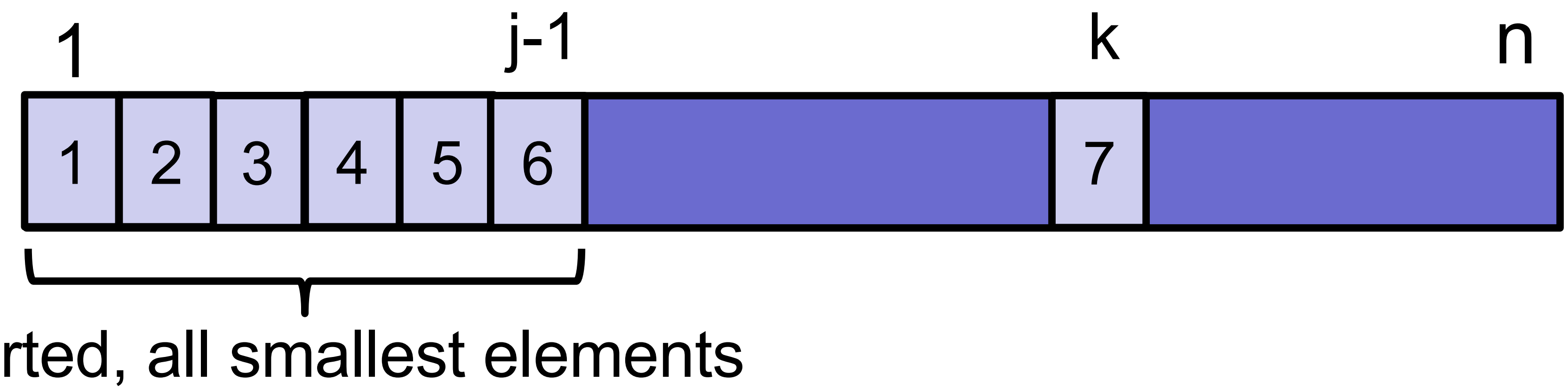
SelectionSort(A, n)

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

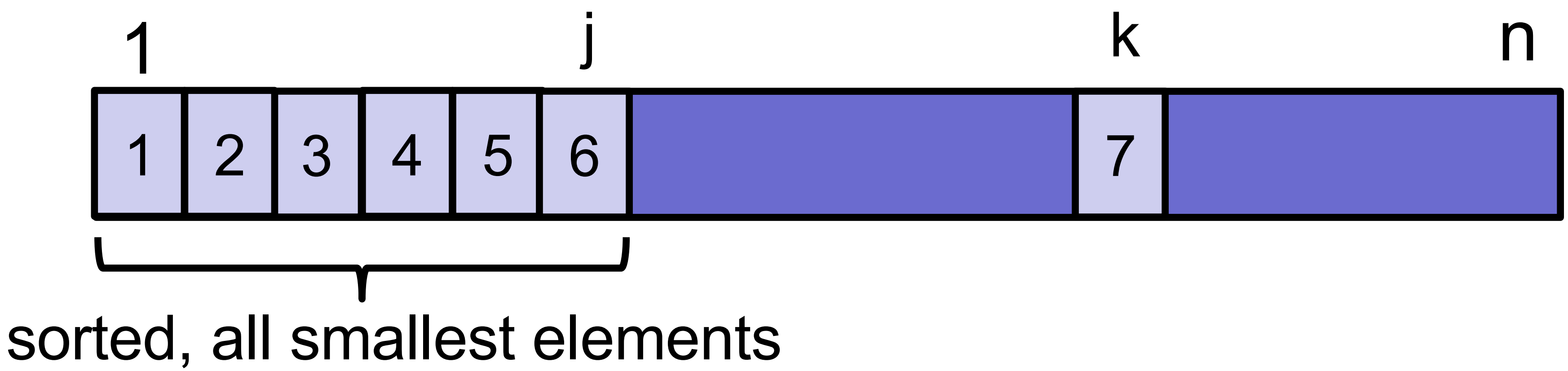
Time: $(n - j)$

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

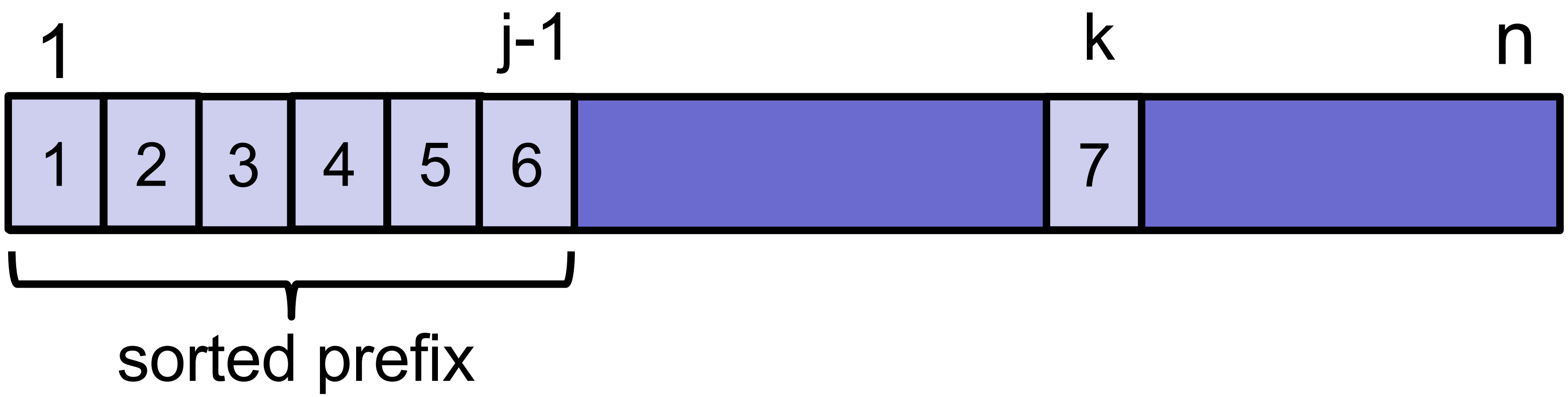
SelectionSort(A, n)

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

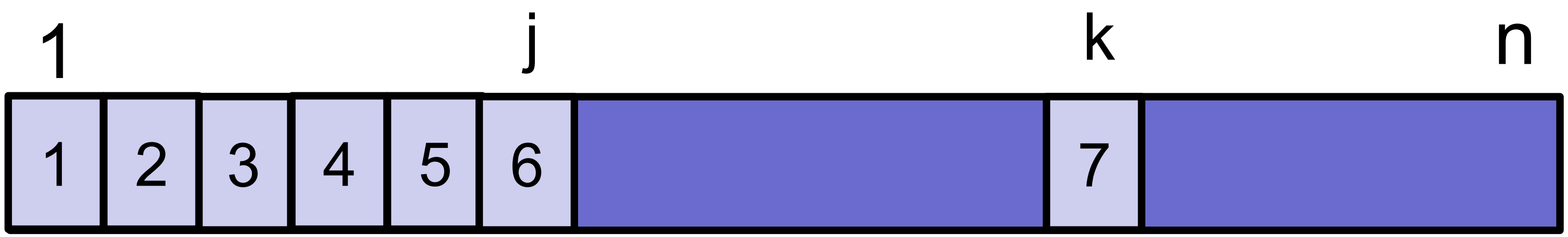
SelectionSort(A, n)

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

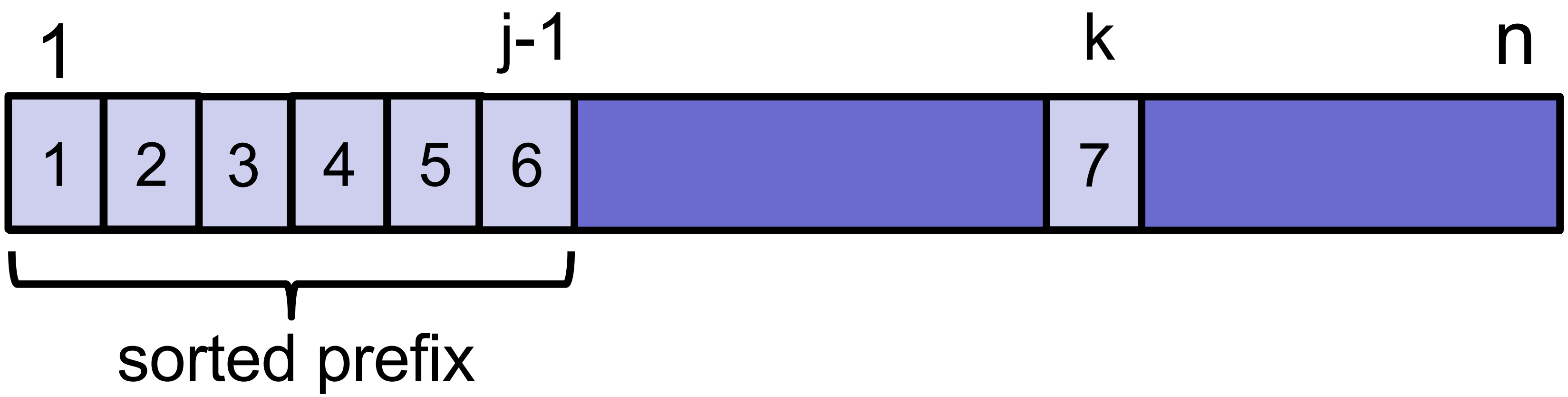
SelectionSort(A, n)

for j← 1 **to** n-1**:**

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

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

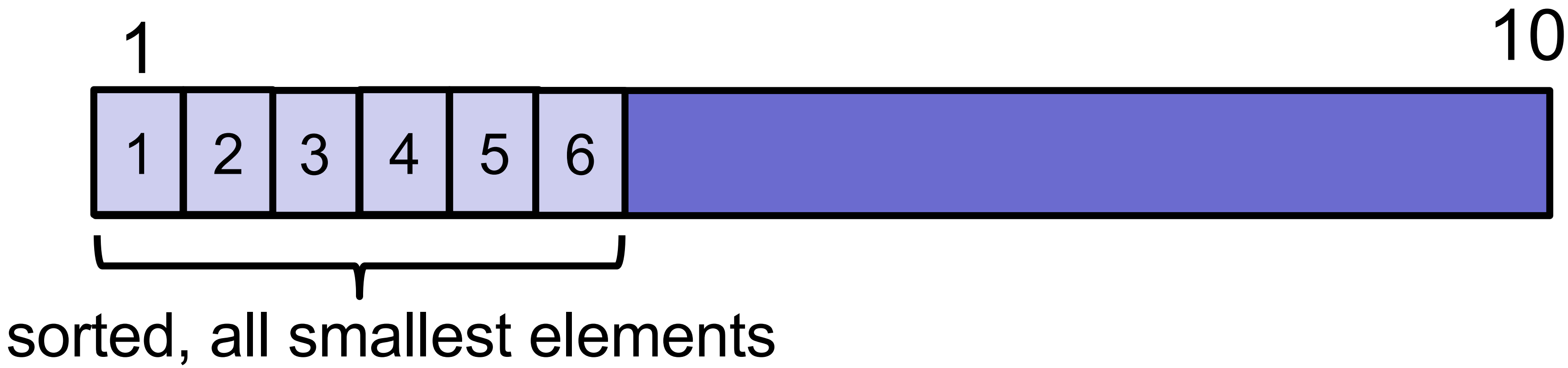
What is a good loop invariant for SelectionSort?



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

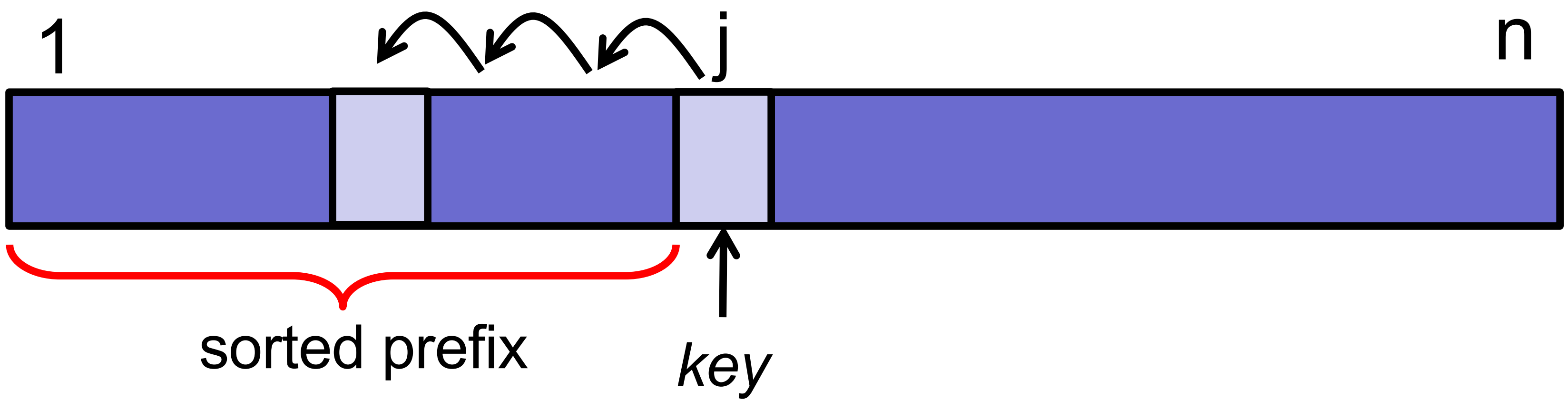
InsertionSort(A, n)

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

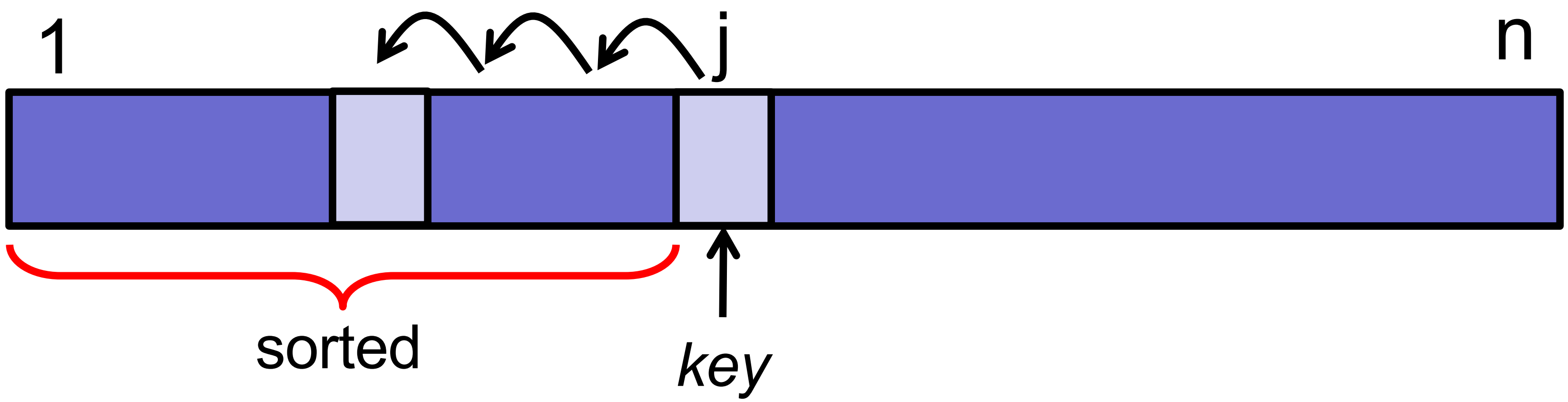
What is the max distance that the key needs to be moved?

Insertion-Sort(A, n)

for $j \leftarrow 2$ **to** n

key $\leftarrow A[j]$

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

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

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

Insertion Sort

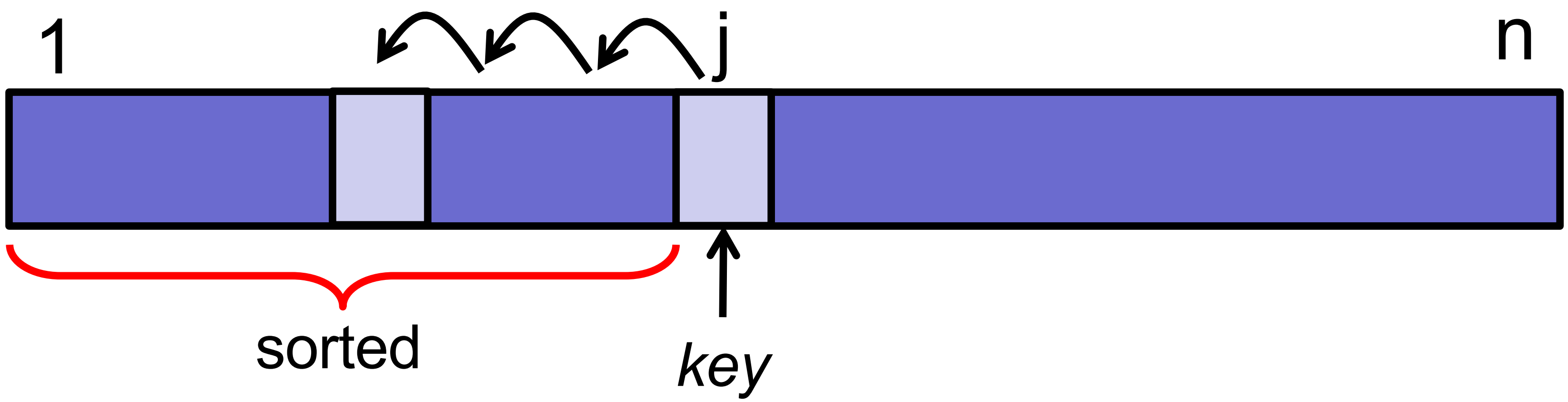
Insertion-Sort(A, n)

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

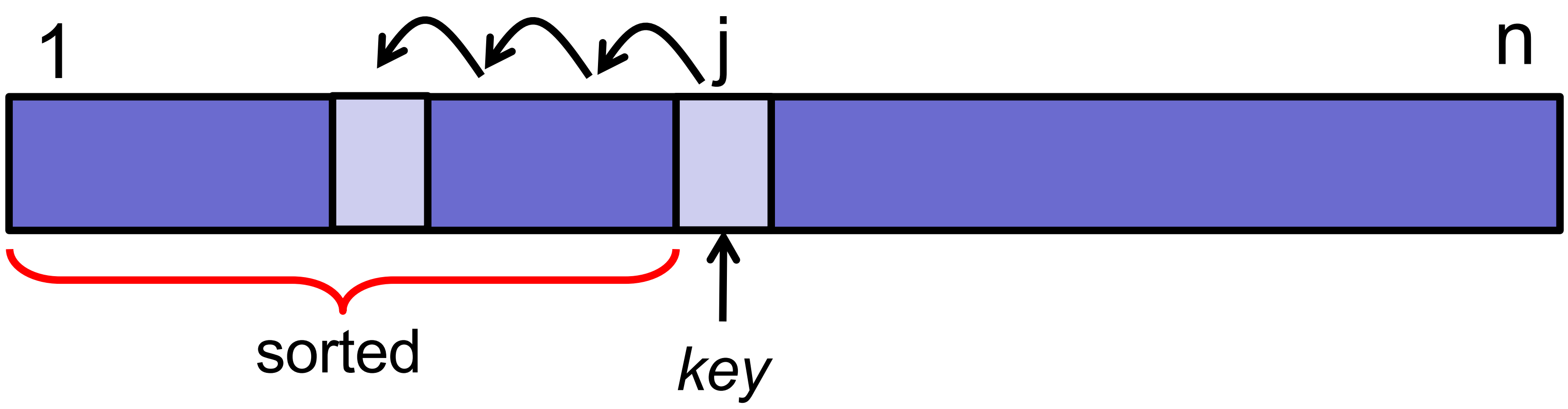
Insertion-Sort(A, n)

for j ← 2 **to** n

key ← A[j]

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

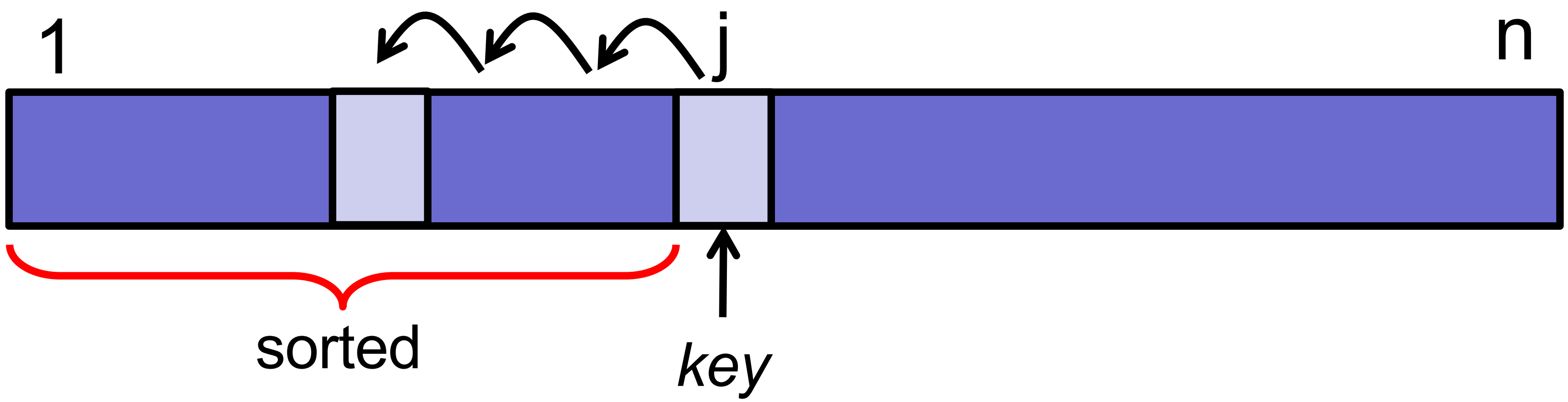
What is a good loop invariant for InsertionSort?



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

Average-case analysis:

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

- Assume all inputs equally likely

$$\sum_{j=2}^{n} \Theta\left(\frac{j}{2}\right) = \Theta\left(n^2\right)$$

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

Today: Sorting

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability